Leasing by Learning Access Modes and Architectures (LLAMA)

Jonathan Waxman

November 2022

Abstract

Lease caching and similar technologies yield high performance in both theoretical [1] and practical [9] caching spaces. This thesis proposes a method of lease caching in fixed-size caches that is robust against thrashing, implemented and evaluated in Rust.

1 Introduction

Caching is the process by which items that may be reused are kept in higher speed memory, known as a cache. While often hierarchical, consisting of many layers of memory of decreasing speed, theoretical analysis of just one layer of a cache has important theoretical implications on caching in general. Since cache sizes are limited in practice, various algorithms exist to maintain the most likely to be reused items in the cache, such as the widely used least recently used (LRU), to various theoretically optimal, but impractical algorithms such as OPT [6], as well as various algorithms that are only useful in special circumstances, like most recently used (MRU). The goal of these algorithms, in general, is to maximize the number of accesses for which the cache contains the item. Note that this analysis does not consider prefetching.

Formally, consider the basic problem of online caching is as follows: Given a sequence S of n pairs $S_i = (k_i, v_i)$ of keys (associated addresses) and values (data to cache), and a cache $C(t)$ with m cache lines, such that at access i the value of $C(i)$ is the set of pairs stored in the cache at that access, maximize the number of accesses in S where $S_i \epsilon C(i)$ (that is, the number of cache hits), where the information to determine $C(i)$ can only rely on $S_1, \ldots S_i$.

Let U be the set of unique keys in S, and let $u = |U|$. For $u \leq m$, the problem is of course trivial, as at any access i, $C(i)$ can simply contain the most recent key-value pair for each key $k_1, \ldots k_i$. For $u > m$, the problem is much more complex. Generally, $C(i-1)$ must, at best, contain all of the keys of $C(i)$ except k_i. As such, in most practical cases the number of maximum possible hits for $m < u$ on any cache is less than the maximum possible hits for any cache with $m \geq u$.

Additional restrictions and relaxations, however, can be applied per algorithm. A restriction common to many practical algorithms is the need to be $O(n)$ time, or at least very close to it, since a cache hit must be significantly faster than a cache miss to be worth the effort. Similarly, additional space is generally restricted, since more memory being used for things other than cache lines means less room for cache lines when the total amount of memory is fixed. As such, a common restriction is the need to be near $O(m)$ space-wise.

The problem of lease caching is similar. Each item that gets stored in the cache is assigned a lease of length of some length t, such that after t accesses it will be evicted. In that case, a related goal is to assign leases such that t for each item closely matches the true reuse interval of the item.

In order to bridge the gap between the theoretical maximum and what is practical in real time, many algorithms seek to approximate theoretically optimal algorithms in a more practical manner, generally by adding assumptions about the input data that are likely to be true in practice, such as lease recently used (LRU) assuming that data that was used more recently is more likely to be used again soon. However, there are trade-offs between the assumptions made and the practical performance of these algorithms. As such, this thesis investigates how a lease caching-based caching algorithm performs in comparison to these existing works in a fixed-size cache as well as how such algorithms can be improved.

1.1 Terminology

As terms vary from paper to paper, defined here are several terms used in the paper.

1.1.1 Item

An item is a data identifier, such as a memory address. Saying that a cache that can store some number of items means that it can store the identifier as well as the data it refers to.

1.1.2 Lease

A number of accesses assigned to an item, after which, if that item has not been accessed, the cache line will be automatically evicted.

1.1.3 Lease Cache

A type of cache that assigns a lease to each item in its cache, and relies primarily on the expiration of leases to evict items [5].

1.1.4 Secondary Eviction Policy

For a lease cache, a policy that is used to evict items with leases when the cache is full and no leases are expired.

1.1.5 Reuse Interval

The reuse interval of an item is the number of accesses to items that occurs between one access to that item and the next access to that item, including the second access.

1.1.6 OPT

An optimal caching algorithm for a fixed size cache of any size. [6]

1.1.7 Cache Size

The number of lines in a cache.

1.2 Contributions

This thesis makes four main contributions. First, it presents Past Access Uniform Leasing (PAUL), an algorithm for caching based only on general frequency trends in reuse intervals based on BOAT/PAUL, my previous unpublished work. Second, it introduces Leasing by Learning Access Modes and Architectures (LLAMA), an extension of PAUL that considers per-key frequency trends as well as incorporating additional methods of identifying frequency trends. Third, it proves the average hit rate for LLAMA for a specific trace. Fourth, it makes several contributions to expanding the general knowledge about various ideas considered, but rejected, during the process from PAUL to LLAMA.

As well, it contains a summary of the behavior of various iterator methods in Rust, many of which were instrumental in the implementation of PAUL and LLAMA.

2 Caching By Learning

While many algorithms, such as LRU, do not attempt to learn any information about the behavior of accesses, those that rely on lease caching generally try to determine the most likely, or at least most profitable, reuse interval for a given access.

2.1 Past Access Uniform Leasing

Past Access Uniform Leasing (PAUL) maintains a histogram of reuse intervals of each access as items in the cache as it gets successfully reused. Its main assumption is that the reuse behavior of a program in the past is a strong predictor of future reuse behavior. From the histogram, it determines a lease assignment based on the smallest lease that would be most likely to generate a hit if the future distribution matches the current distribution. If such a lease would be predicted to overfill the cache (thus causing the secondary eviction policy to activate and thus rendering leases somewhat pointless), it instead

determines a dual lease, a lease that at some known probability will be either the smaller or larger of two leases, such that the average fill predicted from such a lease would fill the cache exactly. That is, on assignment, a dual lease resolves to either the smaller or larger lease at that given probability. This method derives from CARL [8]. That is, this algorithm derived from CARL [8] consists of iterating through the list of possible reuse intervals (based on those observed) and finding the biggest one that doesn't exceed the target size, and then, assuming a lease of equivalent size would underfill the cache, find the smallest lease that does exceed the target size and assigns them in different proportions probabilistically such that the average fill should be acceptable, thus mitigating (but not wholly preventing) underfilling and overfilling of the cache.

It relies on a secondary eviction policy of random eviction, which is resistant to thrashing as recency is not a factory in its choice.

A fundamental problem with PAUL, beyond its assumptions, is the averaging problem. While PAUL approaches OPT's hit rate for traces with constant reuse intervals, any amount of variation in reuse interval can impact performance. For example, consider the case of a trace with a reuse interval distribution of 50% of reuses being of interval 1000, and 50% being of length 10. Since we assume the average cost of the lease, we must assign a dual lease that assigns 10 and 1000 at different proportions of the time, even when it would perhaps be more efficient to simply assign 1000 all the time, and evict leases with a lease of 990 or less remaining once the cache would be full. That is, assigning 1000 all the time may lead to a higher hit rate.

2.2 The Road to LLAMA

On the road to improve PAUL, many different areas were investigated for the potential to make it into PAUL. These investigations are presented here.

2.2.1 Approximate Modeling of PAUL

One of the ways to help find areas PAUL could be improved in would be to create an approximate model of PAUL that could be more easily analyzed.

Jaccard Similarties: The Jaccard similarity of two sets A and B is equal to the cardinality of their intersection divided by the cardinality of their union [3]. Trivially extending it to multisets, it can be used to model the similarity between our distributions. Unfortunately, there does not appear to be any correlation between the Jaccard similarity of the reuse interval distributions at different points in the trace and the hit rate of PAUL.

***p*-norms:** The p-norm of two vectors is a distance metric between them, with a 1-norm being taxicab distance (Manhattan distance) and 2-norm being euclidean distance, and the infinity norm being the limit of p as it approaches infinity [13]. Examining the 1-norm, 2-norm, and infinity norm, of the reuse interval distributions at different points in the trace did not reveal any correlation with PAUL hit rates.

2.2.2 Multi-armed Bandit

In artificial intelligence, the multi-armed bandit problem is a learning problem where a choice among multiple options, with unknown reward outputs randomly distributed across some unknown distribution, are presented, with the goal of maximizing said reward [10, Chapter 2 Section 1]. In the traditional case, the unknown distributions are static, which would mirror the problem of choosing a correct lease length in a highly cyclic trace, where the reward (cache hit or miss) is based on the distribution of reuse intervals. However, there exists a nonstationary variant [10, Chapter 2 Section 4], which would mirror the more general problem of choosing a correct lease length in a trace where the behavior is not fully known. This remains an area of future investigation.

Many existing multi-armed bandit solutions are implementable in $O(n)$, such as greedy, ε-greedy, and upper confidence bound, based on their descriptions in [10, Chapter 2]. Space complexity is generally proportional to the number of actions (based on that same description), which presents a challenge for a space with theoretically infinite action possibilities [10, Chapter 2].

However, the way PAUL tracks reuse distances lowers the amount of actions in a real system to a very small set, specifically, reuse intervals are tracked by the next highest power of 2 [that is, $block = \lceil log_2(ri) \rceil$, and $length = 2^{block}$]. Due to the logarithmic nature of this compression, the block values only span [0, 63] in current real use cases, as the capability of tracking a reuse distance is limited to a length storable in Rust's *usize*, which for 64 bit machines, caps out at $2^{64} - 1$, thus limiting the number of distinct actions to 64. This means that the number of actions can be treated as constant (since it can be bounded by 64) for the purposes of time complexity. Thus, a multi-armed bandit solution would have $O(n)$ time complexity and $O(1)$ space complexity in practice, which is ideal for a caching scenario. However, examining the performance of such an algorithm did not seem to provide good hit rates, at least from rudimentary examples. The issue of reward tracking remains complex—we don't generally know whether a decision will result in a hit or miss until either the item is evicted or it results in a hit. In addition, there remain problems from the cascading impact of multiple actions combined. Thus, multi-armed bandit-based lease caching remains an area of active investigation.

2.2.3 Reservoir Sampling

Rather than attempting to store the whole distribution, or only relying on successful reuses, I experimented with different sampling methods to learn the histograms for PAUL. I tried random sampling at a fixed rate, but was displeased with the still asymptotically linear space requirement involved for an ever-growing sample, and instead decided on reservoir sampling, to allow for a fixed size sample that was less biased by things such as the item's chance of reuse or the item appearing many times in quick succession.

Reservoir sampling [11] is a method for obtaining a random sample of a stream without knowing that stream's length - such as in a program's memory

access pattern. By including a new item at a rate inversely proportional to the number of items seen so far, the chance that any one item is included in the sample is always equal to the number of items that can be held in the sample, divided by the number of items seen so far. This gives equal weight to items regardless of where they appear in the trace, as long as they have an equal number of appearances.

Since this sample is of a fixed size, it has a space requirement proportional to the amount of items allowed to be tracked within it.

2.3 Leasing by Learning Access Modes and Architectures

Rather than tracking only the histogram of reuse intervals overall, Leasing by Learning Access Modes and Architectures (LLAMA) expands on PAUL by implementing per-item tracking in addition to overall reuse interval tracking. That is, whenever an item is reused, the reuse interval is recorded in that item's histogram as well as in a histogram that maintains the reuse interval distribution of all items collectively. When a lease needs to be assigned, if it is for an item with an existing histogram, the lease is based on that data; otherwise, it is based on the data for the trace as a whole so far. This changes its main assumption from PAUL's assumption that past reuse behavior of a program is a predictor of future reuse behavior, to instead assume that past reuse behavior of an item in a program is a predictor of that item's future reuse behavior. While it does maintain PAUL's assumption for items with no known information, it minimizes its reliance on it.

Listing 1 From LLAMA's Lease Assignment

```
let (mut iterator, value_sum, map) = {
    match self.evictions.per_key.get(key) {
        Some(submap) => (submap.map.keys(), submap.value_sum,
        &submap.map),
        _ => (
            self.evictions.overall.map.keys(),
            self.evictions.overall.value_sum,
            &self.evictions.overall.map,
        ),
    }
};
```

As such, for items with known distributions in a repeating cycle, the averaging problem is heavily reduced, as only real possibilities are considered, rather than considering cases that don't actually exist for the item. That is, the average is over more sensible possibilities.

However, to do so with perfect accuracy adds a space requirement proportional to the number of unique keys that would likely be impossible to fulfill in

practice except if the size of the cache line is far greater than the size of a cache line.

In addition, rather than relying simply on items being reused from the cache itself, LLAMA relies on reservoir sampling of keys with insertion times in order to to track reuse intervals across varied interval lengths. Since only the keys of the items are tracked in the reservoir sampling, many keys can be tracked per cache line set aside for this reservoir.

It also maintains PAUL's random secondary eviction policy to maintain robustness against thrashing.

3 Evaluation

Each cache was tested on finite length segments of both sawtooth and cyclic artificial traces of various combinations. LLAMA, LRU, and MRU were implemented in a Rust framework built around a **Cache** trait that abstracts away the basic functions of a cache needed from an outside perspective, as well as ensuring that each implementation was restricted to the same set of information.

3.1 Framework

The **Cache** trait that each framework had to implement had the following trait methods. In each case, the key is the associated address for an item, and the value is the actual data of that item.

Listing 2 The Cache Trait

```
pub trait Cache<Key, Value> {
    fn is_empty(&self) -> bool;
    fn contains(&self, key: &Key) -> bool;
    fn get(&self, key: &Key) -> Option<&Value>;
    fn add(&mut self, key: &Key, value: Value);
    fn clear(&mut self);
}
```

3.1.1 is_empty

A method that returns whether the cache has any items stored within it.

3.1.2 contains

A method that takes a reference to a key, and returns whether the cache has an associated item for that key.

3.1.3 get

A method that take a reference to a key, and returns a reference to the current associated value, if there is one, or reports that there is not one.

3.1.4 add

A method that takes a key and a value, and stores that pair in the cache.

3.1.5 clear

A method that empties the cache.

3.2 Results

For all tests, segments of length 10,000 of each sequence were used, since practical testing on an infinite pattern would be impossible. However, these results approach the expected closed forms for LRU and MRU [7].

The results presented for LLAMA are the average of ten runs, since it is non-deterministic. Also for LLAMA, the number of keys to be tracked in the reservoir was equal to eight times the cache size.

3.2.1 Cyclic Traces

A cyclic pattern is one that traverses some keys in a specific order, and then after having seen every key, repeats that pattern, ad infinitum. MRU is optimal for this pattern, since the one that's reuse is furthest away will always be the one that just entered the cache, since the reuse intervals are identical for every element, and it will be the one that has the lowest amount of that interval already passed. LRU suffers from thrashing in this instance, resulting in a zero hit rate in most of the tested cases.

For MRU, the hit rate for a cache of size c with m unique keys is as follows as the length approaches infinity, per the method described in [7]:

$$h(m, c) = \frac{c - 1}{m - 1}$$

This lines up with the practical results.

3.2.2 Sawtooth Traces

A sawtooth pattern is one that traverses some keys in a specific order, and then traverses those keys in the opposite order, ad infinitum.

For MRU, the hit rate for a cache of size c with m unique keys is as follows as the length approaches infinity, per the method described in [7]:

$$h(m, c) = \frac{c}{m}$$

This lines up with the practical results.

The results are presented below:

Table 1: Results for Cyclic Traces

Cache Size	Unique Keys	LRU Hit Rate	LLAMA Hit Rate	MRU Hit Rate
2	3	0	0.30049	0.4999
2	100	0	0	0.01
9	10	0	0.79473	0.888
9	100	0	0.00338	0.08
20	21	0	0.90014	0.9481
20	30	0	0.55984	0.6536
20	100	0	0.14834	0.19
90	100	0	0.84721	0.89
100	100	0.99	0.99	0.99

Table 2: Results for Sawtooth Traces

Cache Size	Unique Keys	LRU Hit Rate	LLAMA Hit Rate	MRU Hit Rate
2	3	0.6665	0.54361	0.6665
2	100	0.0198	0.01483	0.0198
9	10	0.8991	0.74903	0.8991
9	100	0.0891	0.06212	0.0891
20	21	0.9504	0.88388	0.9504
20	30	0.665	0.53182	0.6641
20	100	0.198	0.16915	0.198
90	100	0.891	0.77002	0.891
100	100	0.99	0.97002	0.99

4 Proof of Average LLAMA Hit Rate

This section forms a proof of the average LLAMA hit rate for the case of a cyclic trace with three unique keys and two cache lines.

Note: this analysis could likely be greatly simplified if it were analyzed as a Markov chain, however I am mostly unfamiliar with them.

Lemma 1. *Given a LLAMA cache with 2 cache lines that begins empty, but with perfect information about the reuse interval distribution (that doesn't change), and an infinite sequence of three unique keys in the same order paired to the same data each time, the cache will enter exactly 19 unique states. Without loss of generality we consider the cache to be holding a set of keys, rather than storing a key value pair, since the value for each key here is unique.*

Proof. Let A, B, and C be the unique keys, in that repeating order. (I.e. the sequence is $A, B, C, A, B, A, B, C, \ldots$) Consider the cache as a set of tuples of keys with their remaining lease. It begins with the empty set (state 1).

On seeing the first unique key A, the cache can be in one of two states, either $\{(A, 1)\}$ or $\{(A, 3)\}$, since each key is assigned a lease of length 1 half of the

time, and a lease of 3 the other half of the time.

Upon seeing the next key B, the cache can be in one of two states for each of the previous two states: from $\{(A,1)\}$ it could enter either $\{(B,1)\}$ or $\{(B,3)\}$ (depending on the lease assigned), since the lease going to 0 would remove A from the cache, and from $\{(A,3)\}$ it could enter $\{(A,2),(B,1)\}$ or $\{(A,2),(B,3)\}$ for similar reasoning.

Upon seeing the next key C, the cache could enter $\{(C,1)\}$ or $\{(C,3)\}$ from $\{(B,1)\}$, $\{(B,2),(C,1)\}$ or $\{(B,2),(C,3)\}$ from $\{(B,3)\}$, $\{(A,1),(C,1)\}$ or $\{(A,1),(C,3)\}$ from
$\{(A,2),(B,1)\}$, and one of four states from $\{(A,2),(B,3)\}$: half of the time, A is evicted, in which case the resultant states, depending on the lease assignment, are $\{(B,2),(C,1)\}$ and $\{(B,2),(C,3)\}$; the other half of the time B is evicted, in which case the resultant states (again dependent on lease assignment), are $\{(A,1),(C,1)\}$ and $\{(A,1),(C,3)\}$.

Upon seeing the next key A, the cache could enter $\{(A,1)\}$ or $\{(A,3)\}$ from $\{(C,1)\}$ and $\{(A,1),(C,1)\}$, $\{(A,1),(B,1)\}$ or $\{(A,3),(B,1)\}$ from
$\{(B,2),(C,1)\}$ and $\{(B,2),(C,3)\}$, the latter of which instead could enter $\{(A,1),(C,2)\}$ or $\{(A,3),(C,2)\}$ if B is evicted instead of C. Finally, $\{(C,3)\}$ and $\{(A,1),(C,3)\}$ could also enter $\{(A,1),(C,2)\}$ or $\{(A,3),(C,2)\}$.

Upon seeing the next key B, the cache could enter $\{(B,1)\}$ or $\{(B,3)\}$ from $\{(A,1)\}$ and $\{(A,1),(B,1)\}$, $\{(B,1),(C,1)\}$ or $\{(B,3),(C,1)\}$ from
$\{(A,1),(C,2)\}$ and $\{(A,3),(C,2)\}$, the latter of which instead could enter $\{(A,2),(B,1)\}$ or $\{(A,2),(B,3)\}$ if C is evicted instead of A. Finally, $\{(A,3)\}$ and $\{(A,3),(B,1)\}$ could also enter $\{(A,2),(B,1)\}$ or $\{(A,2),(B,3)\}$.

Upon seeing the next key C, the cache could enter $\{(C,1)\}$ or $\{(C,3)\}$ from $\{(B,1)\}$ and $\{(B,1),(C,1)\}$, $\{(A,1),(C,1)\}$ or $\{(A,1),(C,3)\}$ from
$\{(A,2),(B,1)\}$ and $\{(A,2),(B,3)\}$, the latter of which instead could enter $\{(B,2),(C,1)\}$ or $\{(B,2),(C,3)\}$ if A is evicted instead of B. Finally, $\{(B,3)\}$ and $\{(B,3),(C,1)\}$ could also enter $\{(B,2),(C,1)\}$ or $\{(B,2),(C,3)\}$. Since the set of states entered here is identical to the set of states entered upon seeing C the first time, and that the set of transitions will also repeat, then there can be no states other than the ones enumerated that could appear during the execution, as they will merely continue to rotate between the three disjoints sets of states entered upon seeing each letter. Specifically, those sets are as follows, each with cardinality 6:

The set of states entered upon seeing a C:

$$\{\{(C,1)\},\{(C,3)\},\{(A,1),(C,1)\},\{(A,1),(C,3)\},\{(B,2),(C,1)\},\{(B,2),(C,3)\}\}$$

The set of states entered upon seeing an A:

$$\{\{(A,1)\},\{(A,3)\},\{(A,1),(B,1)\},\{(A,3),(B,1)\},\{(A,1),(C,2)\},\{(A,3),(C,2)\}\}$$

And the set of states entered upon seeing an B:

$$\{\{(B,1)\},\{(B,3)\},\{(B,1),(C,1)\},\{(B,3),(C,1)\},\{(A,2),(B,1)\},\{(A,2),(B,3)\}\}$$

Thus, the set of states accessible to the LLAMA cache in this case is the repeated states in the above sets plus the empty set. Since the three sets are disjoint, the cardinality of that set is 19. As such, there are exactly 19 unique accessible states. □

Remark. *Let X be some arbitrary key, and Y be the next key after that, and Z the next key after that in a set of three repeating keys, on the LLAMA cache described in the previous lemma. In that case, the repeating set of states for all three sets upon seeing that symbol X in the steady state can be written as:*

$$\{\{(X,1)\},\{(X,3)\},\{(X,1),(Y,1)\},\{(X,3),(Y,1)\},\{(X,1),(Z,2)\},\{(X,3),(Z,2)\}\}$$

Remark. *The chance that a lease for the LLAMA cache described in the previous lemma to be assigned a length 1 is $\frac{1}{2}$, the same as the chance that it is assigned a lease of length 3. (This is because the target cost is 2 (the number of cache lines), and a lease of half 1, the trivial lease, and half 3, the desired lease length, has an average cost of 2.*

Remark. *The chance that any specific item for the LLAMA cache described in the previous lemma to be evicted when the cache is full is $\frac{1}{2}$. (As there are two items in the cache and eviction is random).*

Lemma 2. *Given the same cache and sequence information as with the previous lemma, the proportion of executions that were in each state relative to the proportion of executions that are in each state after seeing some key X relative to the proportion immediately prior can be determined by a set of functions. Let $S_0\prime$ correspond with the proportion of executions that entered $\{(X,1)\}$, $S_1\prime$ correspond with those that entered $\{(X_3)\}$, $S_2\prime$ to $\{(X,1),(Y,1)\}$, $S_3\prime$ to $\{(X,3),(Y,1)\}$, $S_4\prime$ to $\{(X,1),(Z,2)\}$, and $S_5\prime$ to $\{(X,3),(Z,2)\}$. Let $S_0,\ldots,S_5$ correspond with the the number of executions that were in the equivalent states for having seen key Z immediately prior (that is, S_i' corresponds with the proportion of executions that are in an equivalent state as those that correspond with S_i after having seen a new key). As such, the functions are those below:*

$$S_0' = S_1' = \frac{S_0 + S_2}{2}$$

$$S_2' = S_3' = \frac{2 * S_4 + S_5}{4}$$

$$S_4' = S_5' = \frac{2 * S_1 + 2 * S_3 + S_5}{4}$$

Proof. The state associated with S_0, $\{(Z,1)\}$ will enter the state associated with S_0' half of the time ($\{(X,1)\}$), and S_1' the other half of the time $\{(X,3)\}$, as half of the time a lease of 1 will be assigned, and half of the time a lease of 3 will be assigned. The same is true for S_2.

The state associated with S_1 will enter the state associated with S_4' half of the time, and will enter the state associated with S_5' the other half of the time by similar logic as the previous case. The same is true for S_3.

The state associated with S_4 will enter the case associated with S'_2 half of the time, and will enter the the state associated with S'_3 the other half of the time by similar logic to the previous two cases.

Finally, the state associated with S_5, $\{(Y, 2), (Z, 3)\}$ has two main cases: half of the time, Y is evicted, in which case half of that time (a quarter of S_5) it enters the state associated with S'_4 and the other half of that time it enters the state associated with S'_5 with similar logic to the other lease-based split cases; the other half of the time, Z is evicted, in which case it enters the state associated with S'_2 half of the time and the one associated with S'_3 the other half of the time, again by the same lease-based split case logic.

As such, the proportion of each S'_i is as follows:

$$S'_0 = S'_1 = \frac{S_0}{2} + \frac{S_2}{2}$$

$$S'_2 = S'_3 = \frac{S_4}{2} + \frac{S_5}{4}$$

$$S'_4 = S'_5 = \frac{S_1}{2} + \frac{S_3}{2} + \frac{S_5}{4}$$

Which is equivalent to those presented above.

□

Remark. *If the set of functions of the previous lemma, assuming the execution path has already entered from another point of the loop (That is, there exists some $S_i^{(-1)}$), then $S_0 = S_1$, $S_2 = S_3$, and $S_4 = S_5$. In that case, those functions can thus be rendered as:*

$$S'_0 = S'_1 = \frac{S_0}{2} + \frac{S_2}{2}$$

$$S'_2 = S'_3 = \frac{3}{4} * S_4$$

$$S'_4 = S'_5 = \frac{S_0}{2} + \frac{S_2}{2} + \frac{S_4}{4}$$

Remark. *The set of states that correspond with a hit on seeing key K as the next key are exactly the set of states that contain the a tuple with the first element being K. As such, the proportion of accesses that get a hit at any step is exactly equal to $S_2 + S_3$.*

Theorem 3. *The limit of the hit rate of the cache described in the first lemma, as the number of items in the sequence observed approaches infinity, is 0.3.*

Proof. Let $x(n) = S_0^{(n)} = S_1^{(n)}$, $y(n) = S_2^{(n)} = S_3^{(n)}$, and $z(n) = S_4^{(n)} = S_5^{(n)}$ for $n > k$, where k is some constant, and n is the number of items seen, with the assumption that $S_i^{(j)}$ corresponds with the proportion of executions that are in the state associated with the state-archetype S_i on the j^{th} iteration of the loop.

Assume that k is large enough that k was part of a loop iteration (so we can combine the values of S_0 and S_1 and so on.

Thus (per the remark):

$$x(n) = \frac{x(n-1) + y(n-1)}{2}$$

$$y(n) = \frac{3}{4} * z(n-1)$$

$$z(n) = x(n) + \frac{z(n-1)}{4}$$

Since the sum of all the proportions at any iteration must be 1

$$2 * x(n) + 2 * y(n) + 2 * z(n) = 1$$

$$x(n) + y(n) + z(n) = \frac{1}{2}$$

Then, $x(n)$ can be rendered into a closed recurrence relation as follows:

$$z(n) = x(n) + \frac{z(n-1)}{4}$$

$$z(n) = x(n) + \frac{4}{3} * \frac{y(n)}{4}$$

$$z(n) = x(n) + \frac{y(n)}{3}$$

$$y(n) = 3 * (z(n) - x(n))$$

$$x(n) = x(n-1)/2 + y(n-1)/2$$

$$2 * x(n) = x(n-1) + y(n-1)$$

$$2 * x(n) = x(n-1) + (3(z(n-1) - x(n-1)))$$

$$2 * x(n) = x(n-1) + 3 * (z(n-1) - x(n-1))$$

$$2 * x(n) = x(n-1) + 3 * z(n-1) - 3 * x(n-1))$$

$$2 * x(n) = -2 * x(n-1) + 3 * z(n-1)$$

$$x(n) = \frac{3}{2} * z(n-1) - x(n-1)$$

$$z(n) = x(n) + \frac{z(n-1)}{4}$$

$$z(n) = \frac{3}{2} * z(n-1) - x(n-1) + \frac{z(n-1)}{4}$$

$$z(n) = \frac{7}{4} * z(n-1) - x(n-1)$$

$$x(n) + y(n) + z(n) = \frac{1}{2}$$

$$x(n) + 3 * (z(n) - x(n)) + z(n) = \frac{1}{2}$$

$$4 * z(n) - 2 * x(n) = \frac{1}{2}$$

$$4 * z(n) = \frac{1 + 4 * x(n)}{2}$$

$$z(n) = \frac{1 + 4 * x(n)}{8}$$

$$2 * x(n) = x(n-1) + y(n-1)$$

$$2 * x(n) = x(n-1) + 3 * z(n-2)/4$$

$$8 * x(n) = 4 * x(n-1) + 3 * z(n-2)$$

$$8 * x(n) = 4 * x(n-1) + 3 * (\frac{1 + 4 * x(n-2)}{8})$$

$$64 * x(n) = 32 * x(n-1) + 3 * (1 + 4 * x(n-2))$$

$$64 * x(n) = 32 * x(n-1) + 3 + 12 * x(n-2))$$

$$x(n) = \frac{1}{2} * x(n-1) + \frac{3}{16} * x(n-2) + \frac{3}{64}$$

From this linear recurrence relation with constant coefficients, the steady state value can be found, which is the limit of $x(n)$ as n approaches ∞ [12].

$x^* = \frac{\frac{3}{64}}{1-\frac{1}{2}-\frac{3}{16}}$ $x^* = \frac{3}{64-32-12}$ $x^* = \frac{3}{20}$

Since $z(n) = \frac{1+4*x(n)}{8}$, and we know the limit of $x(n)$ as n approaches $infty$, we know that $z^* = \frac{1+4*x^*}{8}$. Thus:

$$z^* = \frac{1 + 4 * \frac{3}{20}}{8} = \frac{1}{5}$$

Since $x(n) + y(n) + z(n) = \frac{1}{2}$, $x^* + y^* + z^* = \frac{1}{2}$. As such

$$y^* = \frac{1}{2} - x^* - z^* = \frac{1}{2} - \frac{3}{20} - \frac{1}{5} = \frac{3}{20}$$

Since y^* is the limit of $y(n)$ as n approaches infinity, and $y(n) = S_2^{(n)} = S_3^{(n)}$, then per the remark that proportion of executions that are a hit at any given access is $S_2^{(n)} + S_3^{(n)} = y(n) + y(n) = 2 * y(n) = 0.3$. Thus, as n approaches ∞, the proportion of executions that get a hit at each access approaches 0.3, and thus, the overall hit rate approaches 0.3 since the proportion of executions that get a hit at each access over the course of the accesses is the hit rate (which will be dominated by the 0.3).

□

5 Related Work

5.1 Protection

A similar concept to lease caching, a cache that uses protection can assign an amount of accesses for which an item is protected [2]. While that protection remains, that item cannot be evicted. This is in contrast to leasing, where the assigned number is a number of accesses after which the item will definitely be evicted, with no guarantee of the item staying until then.

5.2 Compiler Assigned Reference Leasing

Compiler Assigned Reference Leasing (CARL) [1] is an offline method to determine lease assignment at compile time of a program for variable sized caches. Under specific assumptions, many of which are similar to assumptions made in PAUL and LLAMA, it is optimal. PAUL and LLAMA derive much of their theoretical basis from CARL, though in the case of PAUL, the original formulation of PAUL predates CARL's formal publication, and was instead based on the work in progress and CLAM [8] .

5.3 Mockingjay

Mockingjay [9] is a technique based on Hawkeye [4] that, while not claiming to do so, effectively uses a form of lease caching. It uses an access table and set sampling to track reuse intervals and tracks data per program counter, rather than per address, further addressing a similar problem to PAUL and LLAMA's averaging problem. It does not track frequency distributions of reuse intervals directly, but rather tries to learn something similar to the average value. From this, it performs the equivalent of assigning a lease to each item, while also relying on a secondary eviction policy based on $|lease - 4|$.

6 Iterators

LLAMA and PAUL use a variety of iterator functions in their implementation. In order to give a better understanding of how iterators works, they are explained below, with examples from PAUL.

6.1 find

Find allows a condition to be checked on each element of the iterator, consuming the iterator until it finds an item where the condition is met, and returning that item if there is one. It uses $Option < T >$ for iterators over T, with Some(T) for cases where it finds an item and None for cases where it doesn't. Here it is used to find an item in the LRU queue that has the same value as the input.

Listing 3 From PAUL's LRU Simulator

```
fn remove(&mut self, t: &T) -> bool {
        let index = self
                .queue
                .iter()
                .enumerate()
                .find(|(_, u)| t.eq(u))
                .map(|(index, _)| index);
        match index {
                Some(value) => {
                        self.queue.remove(value);
                        true
                }
                None => false,
        }
}
```

6.2 map

Map converts an iterator over T to an iterator over U by applying some provided function from T to U to each element. This transformation must be valid for all possible values of T. Here it is used to transform an Option type of a tuple to an Option of one of its elements, discarding the other if it were the Some variant. Note that T and U do not have to be distinct.

6.3 enumerate

Enumerate is perhaps one of the simplest operation on an iterator, numbering each item in the iterator. In more technical terms, it maps an iterator over T to an iterator over (usize, T), where the usize is the number of items iterated through before seeing that item. For example, given an iterator over the letters of the alphabet, enumeration would yield (0, a), (1, b), ... (24, y), (25, z). Here it is used to keep track of the index of each item in the LRU queue, so that when find returns a value in the queue that matches the one we're trying to remove, we'll be able to know its index.

6.4 filter_map

This transforms an iterator over some type T to an iterator over some type U, while allowing for the possibility of ignoring some options, such as options that can't be transformed easily.

Here, it is used to unwrap a *Result* type while silently ignoring errors, by mapping all valid entries to their unwrapped forms, while filtering out all invalid entries.

Listing 4 From PAUL Main Loop

```
let (...) = read_lines(in_file)
        .unwrap()
        .into_iter()
        .filter_map(|x| match x {
                Ok(y) => Some(y),
                Err(_) => None,
        })
        .fold(
                (0, 0, ..., 0),
                |(total, hits,  ..., space), entry|
                {...}
        );
```

As with map, note that T and U do not have to be different types, and that it takes a closure with input type T and output $Option < U >$, where None will be filtered out and the rest are put into the iterators unwrapped.

This is akin to using the ordinary map method and then filtering out the result (for which there is an aptly name filter method not discussed here).

6.5 fold

Fold allows you to repeatedly apply a closure to each element in an iterator while accounting for some data in between iterations with a variable (known as the accumulator). Given some initial value for that variable, and a closure taking the variable and the iterator item and outputting the next value of the variable, it consumes the iterator by repeatedly evaluating the closure for each element in order, returning the final value of the variable.

Here, it's used to track the growing number of each type of outcome of a cache access by summing it with the previous and performing the simulation of the cache at each step within the body.

In general, fold allows for accumulations (such as summation) of values over an iterator without needing mutable variables. While some mutable variables are used in PAUL for convenience (such as the cache simulator itself being a mutable struct), in theory fold allows for a functional style that uses no mutable variables.

7 Summary

Lease caching is a promising technology that, while still seeming to have room to grow, demonstrates practical results in my testing, with LLAMA beating LRU in its handling of cyclic traces, though still somewhat off from the optimal MRU. For sawtooth traces, LLAMA lags behind, but not by a lot. As such,

LLAMA may be useful in situations where cache patterns vary heavily and are prone to issues such as thrashing that plague LRU. While many experimental ideas along the way failed to provide good results, they nonetheless expanded the understanding of how to improve and characterize lease caching algorithms. The proof of LLAMA's hit rate on the the cyclic example of size three with two cache lines also extends the general knowledge of how such caches behave. In addition, the use and understanding of Rust iterator methods and Rust traits shows the power of good abstractions.

8 Acknowledgements

A thank you to Professor Chen Ding for guiding me through the thesis process as my thesis advisor as well as teaching me what I know about lease caching and introducing me to Rust. In addition, thank you to Professor Michael L. Scott, for serving on my thesis committee. As well, thank you to Professor Hangfeng He for serving as the third faculty member for my thesis examination. Finally, I would like to thank all of those who attended my thesis defense.

The original research that established the idea of PAUL in the Summer of 2021 (BOAT/PAUL), was supported in part by the National Science Foundation's Research Experiences for Undergraduates program (Grant No. 1717877, 1909099). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

[1] Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. Carl: Compiler assigned reference leasing. *ACM Trans. Archit. Code Optim.*, 19(1), mar 2022.

[2] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *MICRO*, pages 389–400, 2012.

[3] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, 1912.

[4] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89, 2016.

[5] Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. Beating opt with statistical clairvoyance and variable size caching. In *Proceedings of the Twenty-Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 243–256, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[8] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. CLAM: compiler lease of cache memory. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*, pages 281–296. ACM, 2020.

[9] Ishan Shah, Akanksha Jain, and Calvin Lin. Effective mimicry of belady's min policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–572, 2022.

[10] Richard S Sutton and Andrew G Barto. *Reinforcement Learning*. Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, 2 edition, November 2018.

[11] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, mar 1985.

[12] Wikipedia contributors. Linear recurrence with constant coefficients — Wikipedia, the free encyclopedia, 2022. [Online; accessed 10-December-2022].

[13] Wikipedia contributors. Norm (mathematics) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 20-November-2022].