Uniform Lease vs. LRU Cache: Analysis and Evaluation

Dong Chen*
jameschennerd@gmail.com
National University of Defense Technology
Changsha, China

Chen Ding*
Fangzhou Liu*
Benjamin Reber*
Wesley Smith*
cding@cs.rochester.edu
fliu14@cs.rochester.edu
breber@cs.rochester.edu
wsmith6@cs.rochester.edu
University of Rochester
Rochester, NY, USA

Pengcheng Li
landy0220@gmail.com
Alibaba Group
San Mateo, CA, USA

Abstract

Lease caching is a new technique that provides greater control of the cache than what is allowed in conventional caches. The simplest control is uniform lease (UL), which means that all leases are identical in length. The UL cache is prescriptive and based on allocation. In comparison, a conventional cache is reactive and based on replacement. They represent two fundamentally different approaches to cache management.

This paper shows two results. First, it proves that a previous model of the LRU cache called Higher-Order Theory of Locality (HOTL) computes the miss ratio of the UL cache. Second, it shows how UL and LRU behave the same and differently through contrived examples and in the 30 benchmarks of PolyBench.

CCS Concepts: • **Computing methodologies** → **Modeling methodologies**.

Keywords: Lease cache, Uniform lease, LRU, Locality metrics, Locality modeling

ACM Reference Format:
Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform Lease vs. LRU Cache: Analysis and Evaluation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3459898.3463908

*The first five authors are ordered alphabetically. Dong Chen was a faculty member at the National University of Defense Technology, China, during the latter period of this study.

ISMM '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8448-3/21/06…$15.00
https://doi.org/10.1145/3459898.3463908

1 Introduction

All modern machines use cache memory to reduce the latency and energy cost of memory accesses. As the number of cores increases, the typical size of cache memory has increased to tens of megabytes in the last level cache and tens of gigabytes of on-package near memory such as MCDRAM on Intel's many-core processor Knights Landing. In some cases, DRAM now can act as a cache for non-volatile memory, i.e., Intel Optane, which functions as a secondary storage [20]. Optimizations that reduce the amount of data movement are valuable. Due to the limited capacity, cache design requires a mechanism for cache replacement; when a new item is cached and the cache is full, there must be some way to select an item to remove. The choice of a cache replacement policy can have a high impact on the total data movement.

Recently, Li et al. [21] proposed a new caching paradigm called a *Lease Cache*. In a lease cache, each cache block is assigned a non-negative integer *lease*, which represents the amount of logical time that it should reside in cache. If a cache block is accessed before its lease is up, then its lease is refreshed. Otherwise, the cache block is evicted at the end of its lease. This concept is growing in relevance, e.g. Twitter's *Time-To-Live Cache* [35].

Unlike conventional cache designs, lease caching is prescriptive. In a traditional cache, the decision of what to replace is deferred until the cache is full and a new piece of data needs to be stored. However, in a lease cache, this replacement decision is replaced by a decision that is made as soon as the data is first accessed. The resident time in cache of a data block is prescribed by its lease. As a direct consequence, the amount of resident data in a lease cache changes dynamically.

Lease caching requires a method for assigning leases to memory accesses, i.e., OSL in [21] and CARL in [26]. Both designs require the program knowledge beforehand, which is usually a heavy-weight step. The simplest possible lease assignment policy that address this problem is *uniform lease (UL) policy*, where each access receives the same lease. We call the cache managed by the UL policy the *uniform lease (UL) cache*.

In this paper, we study the UL cache policy and compare its performance with the conventional LRU cache. As we will explain in Section 3.2.1, we found that an UL cache uses the same recency information as a LRU cache. An UL cache has variable size, while a LRU cache has fixed size. This difference raises the question of how the two compare in performance. The following lists the main contribution of this paper:

- *Theory*: We give the theoretical conditions under which UL and LRU perform the same, and we prove their equivalence. (Section 4).
- *Performance*: We compare the cache performance of the UL cache and the LRU cache, first using contrived examples (Section 3.2) and then the 30 benchmarks in the PolyBench benchmark suite(Section 5).

We assume unrestricted dynamic occupancy in the UL cache, which cannot be always valid in an actual system. Still, the modeling of the ideal UL cache in this paper has three uses in practice.

First, as we will prove in Section 4.1, the recent fast techniques to predict the LRU cache performance in particular HOTL (Section 2.3), are in fact a model of the UL cache. As a result, whether HOTL predicts LRU accurately for a program depends on whether UL and LRU perform the same for that program. This paper represents a new method to analyze HOTL accuracy, which is essential in efficiently predicting the LRU cache performance (Section 4.4).

Second, the lease cache is prescriptive and fundamentally different from conventional caches which are reactive. Prescriptive caches are variable in size and based on allocation. Reactive caches can be fixed sized since they are based on eviction. As explained in Section 3.2.1, we consider the LRU cache as the baseline policy for reactive caching, when no program information is required. As the baseline policy, LRU gives the lower bound of attainable performance. For prescriptive caching, the corresponding baseline is the UL cache. The relation between UL and LRU shows how these two caching approaches, prescriptive and reactive, compare in their worst attainable performance.

The last use concerns with how viable the lease cache is in practice. While the paper does not aim to justify the lease cache paradigm, such studies exist. Li et al. [21] showed that a lease cache policy called Optimal Steadystate Lease (OSL) can potentially obtain significant performance improvement over conventional caching techniques in the case where program information is available. Prechtl et al. [26] recently showed that, with program information, the lease cache outperforms LRU in an actual (fixed-size) cache implemented on a FPGA board. These recent studies have shown that the lease cache is superior in performance and feasible to implement.

However, not all programs have predictable behavior. It is imperative in practice that the lease cache have robust baseline performance in situations where program information is unavailable. The UL-LRU equivalence, if established, would show that ideal lease caching is performance safe, that is, when there is no program information to assign leases, the UL policy still provides similar performance to LRU.

2 Background

2.1 The Reuse Interval

One crucial notion used throughout this paper is the *reuse interval (RI)*. Reuse Interval is defined as the change in logical time between a datum's use and its reuse. Suppose we have a trace *abccba*. In this case, the reuse interval of the datum *a* is 5. Given a trace, we may construct a distribution of all RIs. We use the notation $P(ri = y)$ to refer to the portion of all reuses with RI y in the trace. In the example above, there are three different reuse intervals; 1, 3 and 5, and each accounts for 1/3 of all reuses. Reuse interval plays an important role in modeling both lease cache and LRU cache, and we will describe its usage in both cache models separately.

2.2 The Lease Cache

Recently, Li et al. [21] proposed a new type of cache called a *lease cache*. In a lease cache, at each data access, a length of time called a *lease* is assigned to the cache line. The lease instructs the cache to store the data block for the duration of the lease. A lease is measured by the number of accesses rather than physical time, so a lease of 1000 means that the data block is stored until 1000 accesses later. If an access is a miss, a new data block is loaded into the cache and given a lease. If the access is a hit, the lease of the data block is renewed. In either case, a lease is given at *every* data access.

2.3 Higher Order Theory of Locality (HOTL)

Reuse interval is central to the higher-order theory of locality (HOTL) [33], a model that hinges on the *footprint* function, or $fp(x)$, to model LRU cache performance. The concept of the footprint is simple: the quantity $fp(x)$ denotes the average number of *unique* elements in a random length-x window from the execution trace. Footprint is computed from a reuse interval distribution ($P(ri = y)$), data size (m), and trace length (n) as follows:

$$fp(x) = m - \sum_{y=x+1}^{n-1} (y - x)P(ri = y) \tag{1}$$

Given an execution trace, HOTL first computes footprint $fp(x)$. Then, HOTL computes the approximate LRU miss ratio as the portion of RIs greater than x, at the point where

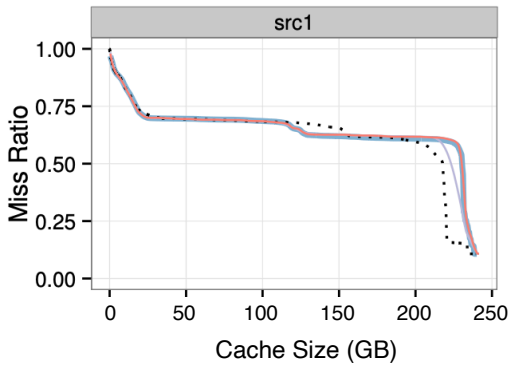


Figure 1. HOTL accuracy on a storage trace as reported by Wires et al. [32] The dotted line is the miss ratio curve predicted by HOTL and the blue line is the actual miss ratios (computed from the reuse distance).

footprint equals cache size (c). Mathematically, for infinitely long traces:[1]

$$mr_{HOTL}(c) = P(ri > x)\Big|_{fp(x)=c} \tag{2}$$

HOTL is the fastest way to model the LRU cache. [19, Table 6] examines the time and space complexity of 7 different cache miss ratio curve analysis techniques based on two metrics, reuse interval (RI) and reuse distance (RD) [23]. With proper implementation, reuse interval collection requires $O(N)$, much lower than that for reuse distance, $O(N \log M)$, where N is the trace-length and M is the entire working set size. The time cost comparison was also made in Xiang et al. [33, Table 2]: among 29 benchmarks from SPEC2006, the average slowdown is 153 times for reuse distance analysis, 23 times for HOTL, and 38 times for simulation for a single cache configuration. Hu et al. [18, Figure 4] also reported that the average overhead per 16 SPEC2006 program is 11 hours for reuse distance, and 1.6 hour for HOTL.

HOTL, however, as observed by [18, Figure 4] and [32, Figure 4], may over- or under-predict the miss ratio occasionally. In Figure 1, we show one example, `src1`, a file access trace recorded by one of MSR's source code control servers [24]. Unfortunately, those studies did not characterize common access patterns that cause this mis-prediction.

Reasoning out these errors of HOTL is necessary, as we will emphasize in Section 4.4, since it had been widely adopted to address many research problems. Such study could give more insight to researchers who use HOTL.

Next, we will discuss how the lease is chosen for the uniform lease cache, using locality theory, and its relation with the conventional LRU cache.

[1]Yuan et al. [37] showed that Eq. 1 is mathematically equivalent to the Denning recursion and discussed the boundary effects in finite-length traces as well as the relation with other locality models.

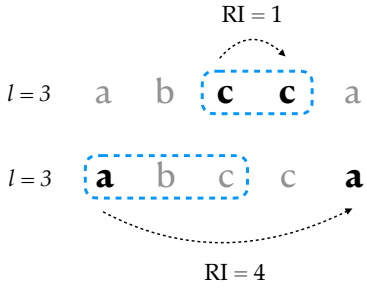


Figure 2. Two examples that demonstrate the relation between the RI and the length of time that lease is active. We assign lease $l = 3$ to each datum. The blue box encloses the time interval where the lease of datum c (RI=1) and a (RI=4) is active.

3 The Uniform Lease (UL) Cache

This section derives the formula for UL cache performance and shows its relation with the LRU cache using contrived examples and with two other caching policies.

3.1 Cache Size and Miss Ratio

Recall from Section 1 that the size of the lease cache changes dynamically. The cache size is given by the number of accesses whose lease is active. The length of time that an access's lease is active depends on the reuse interval of that access and we demonstrate this relation in Figure 2: 1) An access whose reuse interval y is less than or equal to the lease l will have an active lease until y accesses later. At this point, the cache block's lease is renewed by a new access, so we say the lease of the original access is no longer active, even though the data is still present in cache; 2) An access whose reuse interval is greater than l has an active lease until its lease expires. Thus the average cache size given lease l can be computed by the sum of these two values over the total RI distribution for the trace, shown in Eq. 3.

$$s(l) = l \cdot P(ri > l) + \sum_{y \le l} y \cdot P(ri = y) \tag{3}$$

The above calculation ignores the effect of last resident times at the end of program execution, and so this formula over counts at most $\frac{l^2}{2}$ units[2]. The actual average size of the lease cache is between $s(l) - \frac{l^2}{2n}$ and $s(l)$, where n is the length of the trace. The effect of over counting can be ignored if $n \gg l$.

[2]The overcount comes from accesses who have no reuses at the end of execution. In the worst-case scenario, the last l accesses in the trace have no reuses. In this case, the oldest cacheline will have a remaining lease of 1, the next oldest of 2, then 3 and so on up to l. The overcount is the sum of these remaining leases.

The miss ratio of the UL cache $mr_{UL}(c)$ at the cache size c, is the portion of RIs greater than the lease l, as shown below:

$$mr_{UL}(c) = P(ri > l)\Big|_{s(l)=c} \tag{4}$$

As an example, consider the infinite trace *abc abc* ... with the uniform lease l. All accesses in the trace have the reuse interval 3, and so the RI distribution is:

$$P(ri = y) = \begin{cases} 1 & y = 3 \\ 0 & y \neq 3 \end{cases}$$

When $0 \leq l \leq 3$, each access has an active lease until l accesses later, and so the cache size $s(l) = l$. When $l > 4$, each access has an active lease until 3 accesses later, so $s(l) = 3$. When $l < 3$, we have $mr_{UL}(s(l)) = mr_{UL}(l) = P(ri > l) = 1$; otherwise, $mr_{UL}(s(l)) = mr_{UL}(3) = 0$.

3.2 UL vs. LRU

3.2.1 Similarities and Differences. Both UL and LRU caches use the same recency information: in implementation, UL caches make use of the expiration time of each resident data block. The expiration time is the last access time plus the lease. In comparison, LRU caches make use of the last access time. Therefore, we say that the two caches use the same recency information.

UL and LRU differ in their eviction logic. The UL eviction is prescriptive. It evicts a block whenever its lease expires. LRU maintains a global ranking of all data and evicts the least ranked block whenever space is needed. If there is no new data in the cache, no data block is ever evicted. In the LRU cache of size c, an access is a miss if its reuse distance rd, i.e., LRU stack distance, is greater than the cache size:

$$mr_{LRU}(c) = P(rd > c) \tag{5}$$

Comparing Eq. 4 and Eq. 5, we see that UL eviction depends entirely on RI, while LRU eviction depends on the number of distinct data items accessed in between. In this sense, we may say that UL eviction is individual, while LRU eviction is collective and communal.

Both UL and LRU can be used to cache any program without program information. They use different parameters: the uniform lease in UL (l in Eq. 4) the cache size in LRU (c in Eq. 5). In practice, both parameters can be given a priori without knowing anything about the workload being cached. If we consider the LRU cache as the baseline policy for a fixed-size cache, then the UL cache can be treated as the baseline policy for a prescriptive lease cache.

3.2.2 Three Contrived Examples. We use three contrived examples to compare UL and LRU caches: one where UL and LRU would perform the same, one where UL outperforms LRU, and one where LRU outperforms UL. They are easy to understand but still instructive since they are representative of common patterns in program execution.

Cyclic The UL and LRU caches behave the same for cyclic accesses. In this pattern, the reuses recur regularly. They have the same-length reuse windows and same-size working sets (WS). With the uniform lease l, the cache stores all d blocks accessed in the last l accesses. The content of the LRU cache of size d has the exact same content. As an example, consider the infinite trace *abc abc* .... All have same RIs, i.e., $P(ri = 3) = 1$. When the lease $0 \leq l \leq 3$, the WS size of each window is l. The WS in each lease window is the same as the content of the LRU cache of the same size.

Phased If the same lease has a different WS size in different phases, the UL cache size varies, which can achieve the same miss ratio with LRU but with a smaller average cache size. Consider *abab xxxx abab xxxx* ... and the comparison between UL and LRU in Figure 3a.

l	$s(l)$	mr_{UL}	c	mr_{LRU}
1	1	5/8	1	5/8
2	13/8	3/8	2	3/8
6	3	0	3	0

(a) Phased pattern, for example *abab xxxx abab xxxx* ....

l	$s(l)$	mr_{UL}	c	mr_{LRU}
1	1	2/3	1	2/3
4	3	1/2	2	1/2
5	10/3	1/3	3	1/3
7	23/6	1/6	4	0
8	4	0		

(b) Sawtooth pattern, for example *abc xxx cba xxx* ...

Figure 3. UL and LRU cache performance comparison. The table lists the lease chosen by the UL cache for each cache size, and their corresponding miss ratio under the UL and the LRU policy.

Sawtooth The sawtooth pattern of reuse happens when the last accessed is first reused (as opposed to cyclic when the first accessed is first reused). The reuses are "nested", where the length of reuse intervals increases from inner reuses to outer reuses. In the UL cache, a lease of a middling length would capture inner reuses, but the same lease is used for outer reuses and wastes cache space. The LRU cache, due to the space constraint, evicts the data of these outer reuses earlier than the UL cache and hence performs better. An example is *abc xxx cba xxx* ... shown in Figure 3b.

3.3 UL vs. Working Set and Protecting Distance

UL has a direct relation with the working set (WS) defined by Denning [9] in that the UL cache with uniform lease l is equivalent to an ideal working-set cache with parameter $x = l$, i.e., the content of the UL cache at each access is the working set of the last length-l window. The UL cache may

also be called an ideal WS cache. The size of the ideal WS cache is the average WS size of all length-l windows. This computes the same average as Eq. 6.

In practice, the working-set policy is reactive. An execution is divided into periods. At the end of each period, the data blocks accessed in the last period are kept while others are evicted. In comparison, UL is prescriptive. Duong et al. [14] developed the *Protecting Distance-based Policy (PDP)*, which "prevents replacing a cache line until a certain number of accesses to its cache set". To choose the best protecting distance, PDP included additional hardware to sample the reuse interval (called reuse distance in the paper) and adaptively found the protecting distance that maximized the hit ratio. Having a fixed size, PDP is partly reactive. In comparison, UL is variable sized and entirely prescriptive.

4 UL-HOTL Equivalence and Implications

4.1 UL-HOTL Equivalence

We first show that the UL cache size has a form similar to the footprint definition in HOTL. The following lemma gives a new formula to compute the UL cache size and shows that it is equivalent to Eq. 3 in Section 3.1.

Lemma 4.1. *The average size of the UL cache with lease l is*

$$s(l) = l - \sum_{y<l}(l-y)P(ri=y). \tag{6}$$

Proof. We denote the portion of accesses with reuse interval y as $P(ri=y)$. Eq. 3 from Section 3.1 calculates $s(l)$ directly. Manipulating this formula gives us:

$$\begin{aligned} s(l) &= l \cdot P(ri > l) + \sum_{y\le l} y \cdot P(ri=y) \\ &= l \cdot (1 - P(ri \le l)) + \sum_{y\le l} y \cdot P(ri=y) \\ &= l - l \cdot P(ri \le l) + \sum_{y\le l} y \cdot P(ri=y) \\ &= l - \sum_{y\le l} l \cdot P(ri=y) + \sum_{y\le l} y \cdot P(ri=y) \\ &= l - \sum_{y<l}(l-y) \cdot P(ri=y) \end{aligned}$$

□

The two miss ratios, mr_{HOTL} and mr_{UL}, use the same equation except for the cache size calculation, where one uses the lease function and the other the footprint. The two functions have a strong resemblance, e.g. between $(l-x)P(ri=x)$ in Eq. 6 and $(y-x)P(ri=y)$ in Eq. 1, but they clearly differ. In particular, the footprint is bounded by the data size m, and m is absent in the lease function $s(l)$. To see the deeper relation, we define a useful condition.

Definition 4.2. (*Maximal RI Sum*) In a program execution with n accesses to m data items, it has the *maximal RI sum* if

$$\sum_{x=1}^{n-1} xP(ri=x) = m \tag{7}$$

The name of the definition states that the condition means that the sum of all RIs is the largest possible. This is proved in the following lemma.

Lemma 4.3. *(*Maximal Expected RI*) In any program execution, the maximal value of the expectation of all finite RIs is bounded by the data size:*

$$\sum_{x=1}^{n-1} xP(ri=x) \le m \tag{8}$$

where n, m are the trace length and the data size respectively.

Proof. Let the sum of all finite-length RIs be S. Let $N(ri) = P(ri)\cdot n$. The sum of RI is equal to the total lifetime of all data. For each data item i, its lifetime lf_i is the time from its first access to its last, or the sum of all its RIs. Since each lifetime is at most the length of the trace, i.e. $lf_i \le n$, the total lifetime of all data is at most the maximal lifetime times the data size. Hence, we have

$$S = \sum_{x=1}^{n-1} xN(ri=x) = \sum_{\text{all data } i} lf_i \le mn$$

The lemma is proved by dividing both sides of the inequality by n.

$$\frac{S}{n} = \sum_{x=1}^{n-1} xP(ri=x) \le m$$

□

In the proof, the two sums of RIs are different groupings of the same RIs: the first by data, i.e. the lifetime sum, and the second by the RI value, i.e. the mean RI.

Using the Maximal RI Sum condition, we can rewrite the HOTL formula by replacing m with the expected RI. Yuan et al. [38] used this condition to show a number of formulas for computing the working-set size. Similarly, we next assume the condition and prove HOTL-UL equivalence.

Theorem 4.4. *(*UL-HOTL Equivalence*) Assuming Maximal RI Sum, the HOTL model computes the miss ratio of the UL cache for all cache sizes, that is,*

$$mr_{HOTL}(c) = mr_{UL}(c) \qquad \forall c \ge 0.$$

Proof. We show that the footprint and the lease cache size are mathematically identical, that is, the footprint $fp(x)$ is the average size of the cache with uniform lease x.

$$
\begin{aligned}
fp(x) &= m - \sum_{ri>x}(ri - x)P(ri) && \text{definition}[3] \\
&= m + \sum_{ri>x}(x - ri)P(ri) \\
&= m + \sum_{\text{all } ri}(x - ri)P(ri) - \sum_{ri<x}(x - ri)P(ri) \\
&= m + x - m - \sum_{ri<x}(x - ri)P(ri) && \textit{Lemma 4.3} \\
&= s(x) && \text{Eq. 6}
\end{aligned}
$$

Since $fp(x) = s(x)$ for all $x \geq 0$, $mr_{HOTL}(c) = P(ri > x)|_{fp(x)=c} = P(ri > x)|_{s(x)=c} = mr_{UL}(c)$ for all $c \geq 0$. □

Consider the infinite cyclic trace *abc abc* ... again with a uniform lease l. We have computed the miss ratio of the UL cache earlier. For comparison, let's now compute the HOTL miss ratio. Recall that the footprint $fp(x)$ is the number of distinct data items in a length-x window. For this example, it is easy to see that $fp(x) = x$ for $0 \leq x \leq 2$ and $fp(x) = 3$ for $x \geq 3$.[4] The footprint is exactly the same as the UL cache size, i.e. $fp(x) = s(x)$. The HOTL miss ratio is therefore identical to the UL cache miss ratio: $mr_{HOTL}(c) = 1$ for $c = 0, 1, 2$, and $mr_{HOTL}(3) = 0$.

4.2 Empirical UL-LRU Comparison

From UL-HOTL equivalence (Theorem 4.4), HOTL computes the miss ratio of the UL cache. Hence, we may take HOTL results in previous studies as the performance of the UL cache. The following is a partial list of publications that use HOTL to predict the LRU cache performance:

- Xiang et al. [33] tested all 2-program co-runs of the first 20 programs of SPEC 2006 CPU benchmarks (by the smallest benchmark id) and compared the model prediction with the miss ratio measured by hardware counters. Of the 190 pairs and 380 miss ratios, just two miss ratios had significant errors when seen in both logarithmic- and in linear-scale plots. These tests validated on a real system but for a single cache size. Ye et al. [36] tested a machine with hardware cache partitioning support and (in Figure. 12) compared the solo-run prediction accuracy for 20 cache sizes for (the solo-run of) 28 SPEC 2006 CPU benchmarks.
- Wang et al. [30] tested 16 program traces and (in Figure 5) compared the model prediction with the fully-associative LRU caches of all sizes between 256KB and 8MB.
- Wires et al. [32] tested 14 disk I/O traces on storage caches and (in Figure 4) compared the prediction with the fully-associative LRU caches of sizes up to 1TB.

These studies show that HOTL is largely accurate in predicting the LRU miss ratio. This means that the UL cache and the LRU cache have a similar performance in most cases.

Since the UL cache has a variable size, it can outperform LRU. This is confirmed by the previous experimental results, which show that when HOTL mis-predicts the LRU miss ratio, especially in Wang et al. [30] and Wires et al. [32] who evaluate on large ranges of cache sizes, HOTL makes under-prediction more often than over-prediction. On average, HOTL miss ratio is lower than LRU. This means that when the UL cache performs differently from the LRU cache, its performance is usually better than that of LRU. While this is expected, the past studies show that such difference does not happen often. Furthermore, it can happen that the UL cache performance is worse than that of LRU.

4.3 A Sufficient Condition of UL-LRU Equivalence

A sufficient condition of UL-LRU equivalence is that the UL cache have a constant occupancy, so it is effectively a fixed-size cache. We give an informal proof here. Both the UL and the LRU cache evict the data block with the oldest last access. In the UL cache, a data block is evicted when its lease terminates. At each eviction, all other data blocks in the cache must have a more recent last access time. Hence, the LRU cache selects the same block for eviction. The content of the UL cache is the same as the LRU cache of the same size. Their miss ratio must be the same.

We call this condition *Constant UL Cache Size*. Section 3.2 shows three examples. The first example, cyclic accesses, has a constant UL cache size (for any lease) and has identical UL and LRU cache performance. The other two examples, phased and sawtooth, do not have a constant UL cache size, and UL and LRU differ in performance.

From the UL-HOTL equivalence (Theorem 4.4) and by transitivity, UL-LRU equivalence implies HOTL-LRU equivalence. The HOTL-LRU equivalence means that HOTL predicts LRU cache locality correctly, i.e. HOTL correctness. Previously, Xiang et al. [33] gave a condition for HOTL correctness called the *reuse window hypothesis*: the distribution of the working-set size in all windows is the same as it is in all reuse windows.

Constant UL cache size is a new condition of HOTL correctness. Note that the condition depends on the UL-HOTL equivalence, which assumes

4.4 Implications in LRU Caching

As mentioned in Section 2.3, HOTL is the fastest method to predict LRU performance. In the past, experiments (mentioned in Section 4.2) have shown that HOTL is largely accurate but can be occasionally wrong. When it is wrong, the

[3] In this proof, we ignore infinite RIs and assume $P(ri = \infty) = 0$. This is implied by the Maximal RI Sum condition. In practice, it means that the data size is negligible compared to the trace length, i.e. $n \gg m$.

[4] An interested reader may use Eq. 1, with the RI distribution and $m = 3$, and verify that it produces the same footprint.

predicted miss ratio may be higher or lower. This paper has proved UL-HOTL equivalence. Therefore, HOTL prediction error is the result of the difference between UL and LRU. Understanding this difference is important: a model that can quickly and accurately predict the LRU cache performance can be widely adopted to address many problems. Here we list three of them:

Cache partitioning. cache partition, which is supported on commodity processors, has been studied to manage the shared cache for co-run applications [15]. Stone et al. [28] showed that optimal cache partitioning can be solved by a greedy solution if programs have convex miss ratios, which may be accomplished with *cliff* removal for the LRU cache [1] and general cache types [29]. Without convex miss ratios, Brock et al. [4] showed that optimal partitioning of the LRU cache is solved by dynamic programming, and the solution requires that HOTL prediction be correct, which is established by UL-HOTL equivalence.

Memory management on non-volatile memory (NVM). Yang et al. [34] and Gugnani et al. [17] both reported the asymmetric read-write latency of the recently released Intel Optane DIMM. Write locality [7], which shares the same rationale with HOTL, predicts write-back ratio for the LRU cache instead of miss ratio. Quantified write-backs can help to guide data movements and memory allocations to reduce writes to NVMs [13]. Combined with the quantified miss ratio by HOTL, write-backs can trade-off with misses for a hybrid system.

Efficient Prediction of LRU cache locality. UL-LRU equivalence provides a new method to analyze and optimize LRU cache performance. In particular, HOTL accuracy can be now analyzed by how UL cache performs differently than LRU cache. This paper includes two such analysis. The first uses contrived examples in Section 3.2.2 to show what access patterns cause HOTL to over- and under-predict and why. The explanation is aided by a new way of analyzing HOTL, i.e. not just as a formula but as a caching algorithm. The next section shows the second analysis which examines UL-LRU equivalence in a set of benchmark programs.

5 Evaluation

We measure experimentally how UL caching compares with LRU caching in performance.

5.1 Experimental Setup

We run our experiments on an AWS cloud server with Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz and 1GB memory.

For the study, we use the PolyBenchC-4.2.1 benchmark suite, which contains about 30 dense array kernels [22]. It is representative of scientific computing applications in linear algebra, statistics, computational sciences, and tensor operations in deep learning. PolyBench programs are short (usually around a hundred lines of code). They are not full-size applications. Owing to their code size, all programs can be clearly and completely understood, carefully examined, and easily changed through manual effort. This makes PolyBench an ideal benchmark suite for our purposes, which involves thoroughly analyzing code-level patterns that correspond to specific cache behaviors.

Figure 4 shows the data size and the trace length in a 2-D plot. Two programs on the lower-left corner have less than 1000 blocks (64KB) and around 10 million accesses. The data size is more than 640KB in other 28 tests. Among these, 20 programs have over 6MB data, and one of them, `heat-3d`, has over 64MB. Eight programs have more than 2 billion accesses. The number of references is shown in the lower graph in Figure 4. The largest programs, `adi` and `correlation`, have 34 and 32 references respectively.

Figure 4 also shows the number of distinct RI values. Matrix multiplication, `gemm`, has six. In general, the data reuse is more complex with more distinct RI values. In this measure, PolyBench has non-trivial complexity in its data access: there are over 1,000 distinct RI values in over a half of the programs and 3/4 of these programs have over 10,000 distinct RI values.

5.2 Lease Selection

In the UL lease cache model, all references are assigned the same lease. The length of the lease is determined by the cumulative RI histogram of all references, as mentioned in Section 3.1. Given a RI distribution, we assign the largest lease to all references such that the average cache use, described in Eq. 6, is less than or equal to the target cache size, C. This maximizes the number of cache hits during execution. The whole process can be represented as Eq. 9.

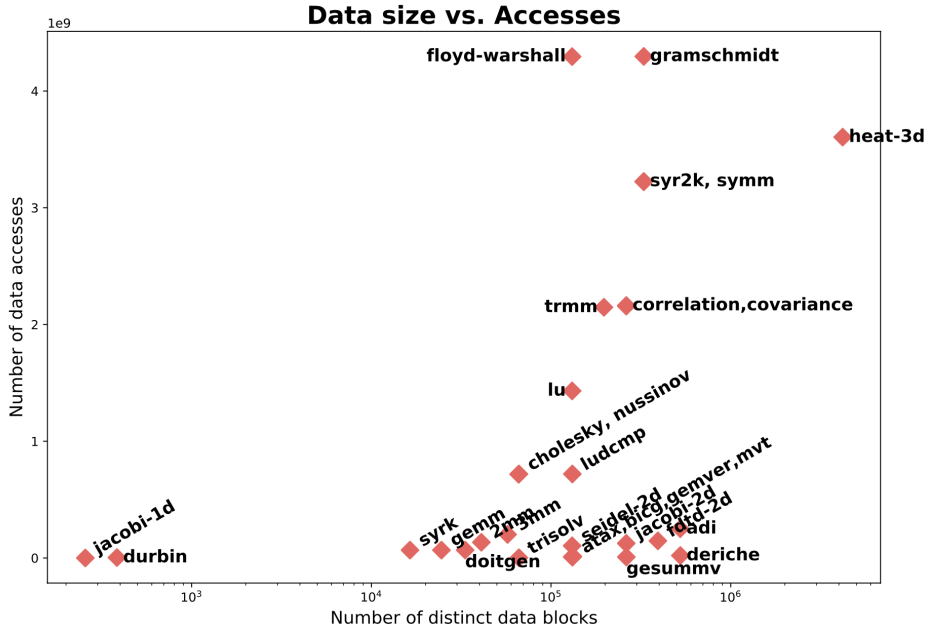
$$UL = \max(l \in P(ri) | s(l) \leq C) \tag{9}$$

Where $P(ri)$ is the RI distribution, $s(l)$ is the average cache size given the lease l (Eq. 6). In Table 1, we shows the uniform leases value assigned for each benchmark when the target cache has 256 cache blocks (16KB). In this experiment, the RI histograms is collected by code instrumentation. More efficient approaches can be applied, e.g. SPS [6].

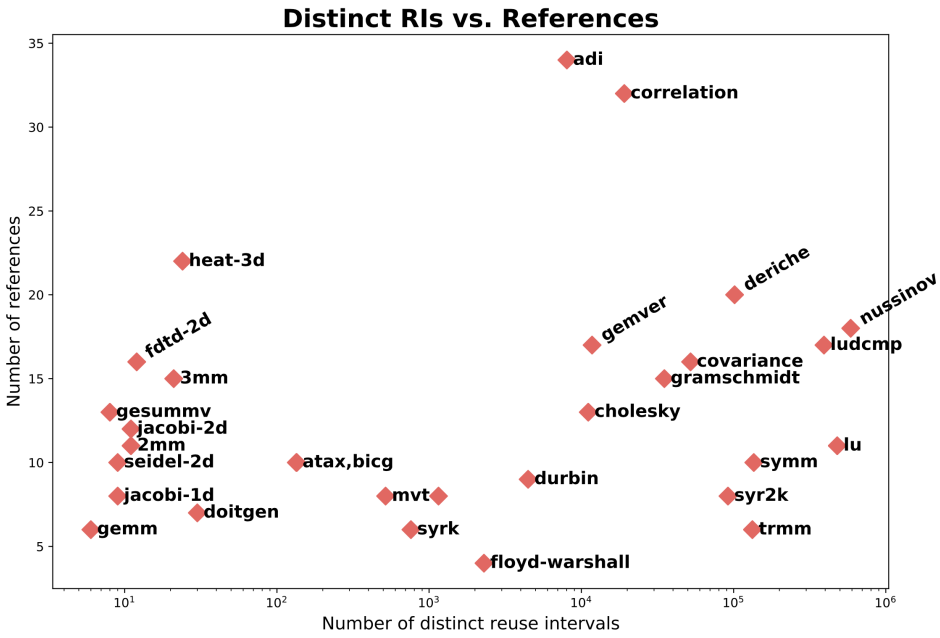
5.3 Performance

Performance modeling is much faster for the UL cache than for the LRU cache. By understanding how they perform the same or differently, we may substitute the expensive LRU cache models [23] with the efficient UL cache models (Section 3.1). In our evaluation, on average, fully precise RI collection for 30 benchmarks is around 12x faster than RD collection with approximation using bins whose range of RI values (of a single bin) is at most 4096.

While hardware caches are set associative, we consider only full associativity here for two reasons. First, the miss



(a) Data size vs. trace length.



(b) The number of references vs. distinct RI values.

Figure 4. Program characteristics of all 30 PolyBench benchmarks.

ratio is defined for all non-negative integer cache sizes, so it provides the fullest comparison. In the shared cache, the actual cache occupancy of a program may be any non-negative integer. Second, the effect of associativity depends on not just the number of ways but also the mapping of data into cache sets. Still, neither argument obviates the importance of set-associativity in practice, which should be studied but is not done here.

5.4 Cases of Equivalence

Figure 5 shows the miss ratios of these two caches. The cache size is shown in (base 10) logarithmic scale on the x-axis.

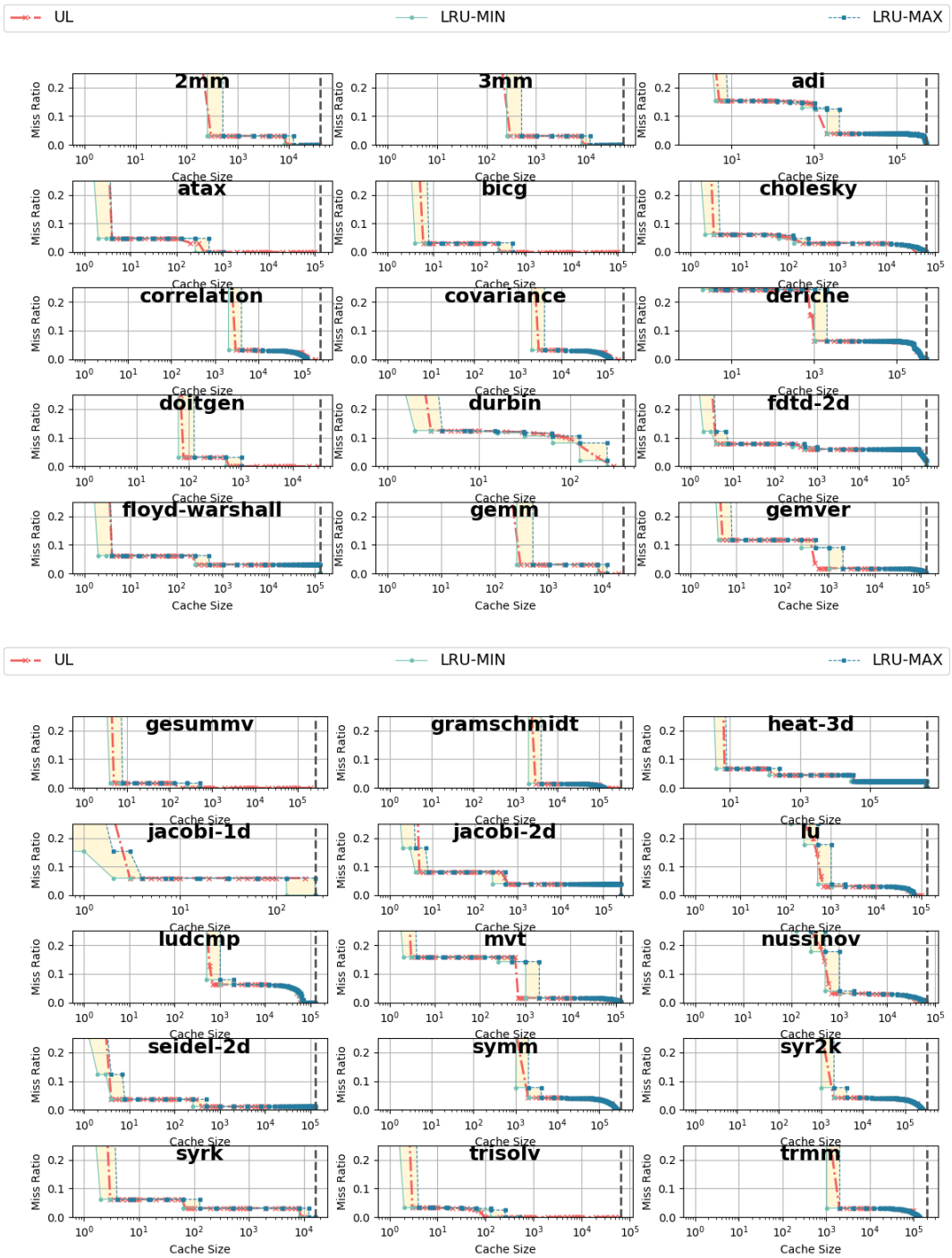


Figure 5. Miss ratio curves of the UL cache vs. the fully-associative LRU cache for 30 benchmarks in PolyBenchC-4.2.1 across all cache sizes. The x-axis is unit in cache block numbers. The LRU miss ratio has a bounded imprecision (it is between the upper and lower bounds shaded in yellow).

Within each power of 10 are nine cache sizes evenly dividing the interval (note that these points are *not evenly spaced* visually, because of the logarithmic-scale plotting). For efficiency, the LRU miss ratio is measured using approximate reuse distance analysis by Zhong et al. [39]. It has a bounded

Table 1. Uniform Lease Assignments for a cache with 256 cache blocks (16KB). The middle column shows the length of the lease assigned to each reference and the right column is the corresponding cache miss ratio.

Benchmark	Uniform Lease	Miss Ratio
2mm	905	0.280
3mm	904	0.280
adi	1651	0.151
atax	5945	0.031
bicg	7919	0.031
cholesky	6054	0.031
correlation	514	0.495
covariance	513	0.496
deriche	1043	0.243
doitgen	6054	0.031
durbin	6538	0.002
fdtd-2d	3246	0.077
floyd-warshall	6080	0.041
gemm	905	0.280
gemver	2157	0.117
gesummv	8137	0.000
gramschmidt	509	0.500
heat-3d	4211	0.044
jacobi-1d	4065	0.000
jacobi-2d	3092	0.081
lu	943	0.247
ludcmp	474	0.493
mvt	1598	0.158
nussinov	943	0.247
seidel-2d	6949	0.036
symm	719	0.349
syr2k	718	0.349
syrk	7126	0.031
trisolv	4093	0.000
trmm	512	0.492

imprecision, and the actual miss ratio is in the area between the two bounds.[5] Furthermore, the miss ratios are for capacity misses. The cold-start misses are the same for the UL and the LRU caches for every cache size and are not included in the plot. The graphs in Figure 5 show the miss ratio up to 25% and omit the results for smaller cache sizes.

The UL and LRU caches have the same miss ratios across all cache sizes for the majority of the benchmarks. Due to the approximation, we consider it is equivalent if the UL cache miss ratio does not deviate outside the bounds of the LRU miss ratio at the same cache size. The two caches are equivalent if their miss ratios are equivalent for all cache sizes.

[5]The bounds are powers of two for cache sizes up to 8192 cache blocks (512KB). For larger caches, the imprecision is at most 4096 blocks (256KB).

The reason for the equivalence is that the data access follows the cyclic pattern described in Section 3.2, but in large scales with more complex structures, e.g. nesting. An example is `2mm`. The program has two matrix multiplications. Each is a triple nested loop with three cyclic patterns. About 47% reuses happen within each innermost loop iteration, 25% between two consecutive iterations of the innermost loop, and 31% between two consecutive iterations of the outermost loop. Next, we focus on the minority programs where the two caches are not always equivalent.

5.5 UL Outperforming LRU

There are 4 out of 30 programs where the UL cache outperforms the LRU cache: `adi`, `deriche`, `gemvar`, `mvt`. In all cases, the miss ratio is equivalent except for one region of difference. The difference has the same pattern we may call a *moving cliff*. A cliff is when the miss ratio drops from a higher plateau to a lower plain. A moving cliff is that UL and LRU have the same cliff, but the UL cliff happens at a smaller cache size than the LRU cliff.

The reason is phased behavior. Upon inspection of the code, we see a specific form of phase that causes the moving cliff. We first describe it in generic terms and then based on individual programs. Consider a program traversing a $n \times n$ matrix in two loop nests: the first in rows and the second in columns. The miss ratio "cliff" happens when the spatial reuse is fully realized in the column traversal.

There are two effects causing the UL cliff to happen before the LRU cliff. First, for the same lease $l = n$, row traversal loop has a smaller UL cache size than a column traversal loop. The UL cache stores one matrix row for the row traversal loop and eight matrix columns for the column traversal loop. The two UL cache sizes are $n/8$ and n cache blocks respectively, considering each 64-byte cache block stores 8 double-precision matrix elements.

Second, the overall UL cache size is the average of the two UL cache sizes, whereas the LRU cache size is the maximum of the two. The average cache size is smaller than the maximum for the two loops. The RD for LRU cache is $n/8$ for row traversal loop, and n for column traversal loop. Since LRU is fixed size, the overall size is n.

We see the moving cliff pattern in 4 programs where UL outperforms LRU, `adi`, `deriche`, `gemver`, and `mvt`. They all have the specific phased pattern described above in the code. `deriche` is the clearest example of a UL cliff happening significantly before the corresponding LRU cliff. Its plot shows that the UL miss ratio drops from 0.243 to 0.153 when the cache is 50KB (800 cache blocks) and it finally reach 0.064 at the 1000-block cache, while the same LRU drop does not happen until the cache size reaches 128KB (2048 cache blocks). An examination of the program source code clarifies this; `deriche` is composed of six depth-2 loop nests, all traversing a square matrix. Four of these loops traverse the matrix in row-major order (reflecting memory order), while two use

column-major order. The drop in the LRU cache miss ratio at cache size 128KB reflects the point at which the cache is now large enough to capture these cross-column reuses (*RI* = *RD* = 2048). The reason UL drops earlier is due to some of the other four nests: when a loop nest contains multiple references to the same data, the UL cache can achieve the same number of hits as an LRU cache while maintaining a much smaller cache. LRU, when increasing cache size, increases cache size for the whole execution; as such, UL does not waste resources in trying to capture long reuse in the same way as LRU, leading to an earlier drop in miss rate.

The same happens to `mvt`, `gemver` and `adi`. The first two benchmarks do matrix-vector multiplications twice, one using the original matrix and the other using its transposed form. `adi` divides its task into two stencil computations. One does column traversal, the other traverses in row. As described before, the difference in cache size requirement between row and column traversal makes the UL cache achieves the same miss ratio prior to the LRU cache.

5.6 LRU Outperforming UL

We do not observe any significant difference when the LRU cache outperforms the UL cache. It happens in a few programs at very small cache sizes. For example, at cache size two in `seidel-2d`, the LRU miss ratio is 32%, lower than the UL miss ratio of 47%. The program performs 9-point stencil computation. In the innermost loop, it has 10 accesses, which has a sawtooth pattern in part of the sequence. These differences are negligible for caching, since they happen at too small cache sizes to matter. Moreover, with an optimizing compiler, such nearby reuses are captured by register allocation rather than caching [5, 8].

6 Related Work

LRU Cache Locality Modeling. The HOTL technique has been used extensively to predict the LRU cache performance. The past results of Wang et al. [30, Figure 5] include 16 SPEC 2006 program traces and compare the HOTL prediction with LRU for all sizes between 256KB and 8MB. Wires et al. [32, Figure 4] makes the same comparison for 14 disk I/O traces on storage caches for cache sizes up to 1TB. The comparison is between the higher-order theory of locality (HOTL) and the reuse distance. Both HOTL and UL use the RI distribution. Yuan et al. [37] show that the formulas based on the footprint are mathematically equivalent for long execution traces. The previous work used HOTL as a model of LRU. In this paper, we show the comparison as one between two methods of caching, i.e., between UL and LRU.

Wang et al. [30] and Wires et al. [32] compare the two policies for tens of test traces and on many cache sizes. The tests are traces from large-size programs or long-running file servers. While the data enumerate the differences, the papers do not give a precise explanation because of the scale and complexity of the traces. In this paper, we use small-size programs and are able to analyze, verify, and explain by examining program code. The prior work shows aggregate differences at the trace level, here we show differences at the source level.

Drudi [12] recognizes that because it uses the average working-set size, HOTL will have trouble if a trace has distinct phases with drastically different working-set sizes. Drudi shows an example trace where HOTL under-predicts the miss ratio by 50% but notes that the example is "highly artificial and cannot be interpreted as a realistic model for system behaviour." In this paper, this difference is viewed as how UL and LRU caches behave differently for phased behavior in both an artificial example (Section 3.2.2) and the occurrences in PolyBench.

Past studies show that HOTL may over predict, e.g. the result by Wires et al. [32] reproduced in Figure 1, but do not explain how over prediction may happen and why it seems rare. This paper shows that these are cases when the variable-size UL cache performs worse than LRU. It happens for the saw-tooth access pattern (Section 3.2.2). The evaluation on PolyBench shows that it happens infrequently and only for small cache sizes (Section 5.6).

Lease Cache. The UL cache behaves identically to the ideal working-set cache (see Section 3.3). In the working-set cache, the data accessed in the last time window are cached [9, 10]. A reader may see Denning [11] for a recent survey of the working-set theory. Like the LRU cache, the working-set cache is based on recency. The theoretical connections between these two have been documented, most recently by Yuan et al. [37]. The comparison between UL and LRU in this paper is also the comparison between ideal working-set policy and LRU. We have given a precise explanation of when and how the ideal working-set policy and LRU are the same or different in non-trivial programs.

Lease cache provides greater control of the cache than what is allowed in conventional caches. An intermediate solution is collaborative caching, where software provides cache hints to influence the replacement decisions in hardware. It has been developed using "evict-me" bits [31] and cache placement hints [2, 3]. The lease cache differs in two ways. First, it targets variable-size cache. Second, leases mean direct control, while cache hints are just suggestions.

In the best case with the complete data access trace, Gu et al. [16] proved that collaborative caching can obtain the performance of optimal cache. Li et al. [21] compared this optimal performance with the optimal lease cache. In the worst case, collaborative caching has no program information, gives no cache hint and behaves as a conventional cache. In the lease cache, the corresponding baseline is the uniform lease. Therefore, the results of this paper can be used as a

comparison of the baseline performance of the collaborative LRU cache with the baseline lease cache (and the ideal working-set policy).

Finally, we note that the lease cache can be efficiently designed and has been recently implemented on an FPGA for both the uniform lease policy [25] and an optimal lease policy [26].

Time-To-Live (TTL) Caches [35] play a significant role in Twitter's caching ecosystem. These TTL caches are not identical to lease caches: they are fixed sized and maintain an eviction strategy alongside the leasing mechanism, in contrast to variable-sized lease caches with no eviction mechanism in the conventional sense. Twitter's TTL caches also do not reset leases on cache hits. Traditionally, TTL caches have existed only in high-level domains, but lease caching is bringing the concept behind TTL to hardware caching. The differences between traditional TTL caches and the concepts behind lease caches are also mentioned in [27].

7 Summary

In this paper, we have studied and evaluated the UL cache. Like LRU and the working-set policy, UL is based on the recency of data access. Unlike them, however, UL is prescriptive and variable sized. We present the first comparison between UL and LRU for PolyBench. It is also the first time that such comparison is done with a complete explanation of program-level causes.

The equivalence between UL and LRU is important in both theory and practice. In theory, we understand how the variable-size cache performs the same, better than, or worse than the fixed-size cache, i.e., the three basic patterns: cyclic, phased, and sawtooth, and the patterns in real code. In practice, we compute the miss ratio of LRU efficiently by using the UL cache miss ratio. Finally, variable-size caching is more efficient for shared cache since an application may be given additional space temporarily. In future work, we plan to quantify this benefit of prescriptive caching.

Acknowledgments

Yu Zhang participated in the early stage of this study. The maximal expected RI lemma was first suggested to us by Yechen Li, then a student at Peking University. We would also thank the anonymous reviewers of NPC 2020 and ISMM 2021 and our shepherd, Huimin Cui, for feedback and suggestions that improved the presentation. The research is supported in part by the National Science Foundation (Contract No. CNS-1909099, CCF-1717877). 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] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 64–75. https://doi.org/10.1109/HPCA.2015.7056022

[2] K. Beyls and E.H. D'Hollander. 2002. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*. Paderborn, Germany.

[3] Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *Journal of Systems Architecture* 51, 4 (2005), 223–250. https://doi.org/10.1016/j.sysarc.2004.09.004

[4] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *International Conference on Parallel Processing*. https://doi.org/10.1109/ICPP.2015.84

[5] Steve Carr and Ken Kennedy. 1994. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1768–1810. https://doi.org/10.1145/197320.197366

[6] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality analysis through static parallel sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 557–570. https://doi.org/10.1145/3192366.3192402

[7] Dong Chen, Chencheng Ye, and Chen Ding. 2016. Write Locality and Optimization for Persistent Memory. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 77–87. https://doi.org/10.1145/2989081.2989119

[8] Keith Cooper and Linda Torczon. 2010. *Engineering a Compiler, 2nd Edition*. Morgan Kaufmann.

[9] Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333. https://doi.org/10.1145/363095.363141

[10] Peter J. Denning. 1980. Working sets past and present. *IEEE Transactions on Software Engineering* SE-6, 1 (Jan. 1980). https://doi.org/10.1109/TSE.1980.230464

[11] Peter J. Denning. 2021. Working Set Analytics. *ACM Computing Survey* 53, 6 (2021), 113:1–113:36. https://doi.org/10.1145/3399709

[12] Zachary Drudi. 2014. *A Streaming Algorithms Approach to Approximating Hit Rate Curves*. Technical Report MS Thesis. The University Of British Columbia.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 15, 16 pages. https://doi.org/10.1145/2901318.2901344

[14] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 389–400. https://doi.org/10.1109/MICRO.2012.43

[15] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 104–117. https://doi.org/10.1109/HPCA.2018.00019

[16] Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding. 2008. P-OPT: Program-Directed Optimal Cache Management. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*. 217–231.

[17] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 626–639. https://doi.org/10.14778/3436905.3436921

[18] Xiameng Hu, Xiaolin Wang, Yechen Li, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2017. Optimal Symbiosis and Fair Scheduling in Shared Cache. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2017), 1134–1148. https://doi.org/10.1109/TPDS.2016.2611572

[19] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Transactions on Storage* 14, 2 (2018), 12:1–12:34. https://doi.org/10.1145/3185751

[20] Intel. 2018. System Memory at a Fraction of the DRAM Cost. https://www.intel.com/content/dam/www/public/us/en/documents/brief/intel-ssd-software-defined-memory-with-vm.pdf

[21] 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 *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 243–256. https://doi.org/10.1145/3297858.3304067

[22] Louis-Noel Pouchet and Tomofumi Yuki. 2018. PolyBench/C 4.2. http://https://sourceforge.net/projects/polybench/files/.

[23] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM System Journal* 9, 2 (1970), 78–117. https://doi.org/10.1147/sj.92.0078

[24] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write Off-loading: Practical Power Management for Enterprise Storage. *Trans. Storage* 4, 3, Article 10 (Nov. 2008), 23 pages. https://doi.org/10.1145/1416944.1416949

[25] Ian Prechtl, Chen Ding, and Dorin Patru. 2020. Design and Evaluation of a Fixed-size Programmable Working-set Cache on FPGAs. preprint online at https://dx.doi.org/10.13140/RG.2.2.24423.60320.

[26] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler Lease of Cache Memory. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. https://doi.org/10.1145/3422575.3422800

[27] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. arXiv:2003.03423 [cs.DC]

[28] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.* 41, 9 (1992), 1054–1068. https://doi.org/10.1109/12.165388

[29] Carl A. Waldspurger, Nohhyun Park, Alexander T. Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 95–110. https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger

[30] X. Wang, Y. Li, Y. Luo, X. Hu, J. Brock, C. Ding, and Z. Wang. 2015. Optimal Program Symbiosis in Shared Cache. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*. https://doi.org/10.1109/CCGrid.2015.153

[31] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. 2002. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. Charlottesville, Virginia. https://doi.org/10.1109/PACT.2002.1106018

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

[33] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 343–356. https://doi.org/10.1145/2451116.2451153

[34] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[35] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[36] Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache Exclusivity and Sharing: Theory and Optimization. *ACM Transactions on Architecture and Code Optimization* 14, 4, 34:1–34:26. https://doi.org/10.1145/3134437

[37] Liang Yuan, Chen Ding, Wesley Smith, Peter J. Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Transactions on Architecture and Code Optimization* 16, 3 (2019), 33:1–33:26. https://doi.org/10.1145/3341109

[38] Liang Yuan, Wesley Smith, Sicong Fan, Zixu Chen, Chen Ding, and Yunquan Zhang. 2018. Footmark: a New Formulation for Working Set Statistics. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*. 61–69. https://doi.org/10.1007/978-3-030-34627-0_5 Springer Lecture Notes 11882.

[39] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Transactions on Programming Languages and Systems* 31, 6 (Aug. 2009), 1–39. https://doi.org/10.1145/1552309.1552310