CARL: Compiler Assigned Reference Leasing

CHEN DING, DONG CHEN, FANGZHOU LIU, BENJAMIN REBER, and WESLEY SMITH, University of Rochester

Data movement is a common performance bottleneck, and its chief remedy is caching. Traditional cache management is transparent to the workload: data that should be kept in cache are determined by the recency information only, while the program information, i.e., future data reuses, is not communicated to the cache. This has changed in a new cache design named *Lease Cache*. The program control is passed to the lease cache by a compiler technique called Compiler Assigned Reference Lease (CARL). This technique collects the reuse interval distribution for each reference and uses it to compute and assign the lease value to each reference.

In this article, we prove that CARL is optimal under certain statistical assumptions. Based on this optimality, we prove miss curve convexity, which is useful for optimizing shared cache, and sub-partitioning monotonicity, which simplifies lease compilation. We evaluate the potential using scientific kernels from PolyBench and show that compiler insertions of up to 34 leases in program code achieve similar or better cache utilization (in variable size cache) than the optimal fixed-size caching policy, which has been unattainable with automatic caching but now within the potential of cache programming for all tested programs and most cache sizes.

CCS Concepts: • **Theory of computation** → *Design and analysis of algorithms*; *Mathematical optimization*; *Discrete optimization*;

Additional Key Words and Phrases: Cache management, reuse interval distribution, cache replacement policy, lease cache, miss ratio curve, optimality

ACM Reference format:
Chen Ding, Dong Chen, Fangzhou Liu, Benjamin Reber, and Wesley Smith. 2022. CARL: Compiler Assigned Reference Leasing. *ACM Trans. Arch. Code Optim.* 19, 1, Article 15 (March 2022), 28 pages.
https://doi.org/10.1145/3498730

Extension of Conference article. We extend Prechtl et al. [54] by providing a theoretical proof of CARL optimality in a variable-sized cache (Section 4), as well as providing an evaluation of the potential of CARL with a variable cache size using the PolyBench and SPEC CPU 2017 benchmark suites (Section 5).
This work was done when the Dong Chen was a graduate student at University of Rochester.
This work was supported in part by the National Science Foundation (Contract No. CCF-2114319, CNS-1909099, and CCF-1717877) and IBM Center for Advanced Studies. 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.
Authors' address: C. Ding, D. Chen, F. Liu, B. Reber, and W. Smith, Department of Computer Science, University of Rochester, 2513 Wegmans Hall, P.O. Box 270226, Rochester, NY 14627; emails: cding@cs.rochester.edu, jameschennerd@gmail.com, fliu14@cs.rochester.edu, breber@cs.rochester.edu, wsmith6@cs.rochester.edu.
Updated author affilition: Dong Chen, College of Computer, National University of Defense Technology, 109 Deya Road, Changsha, Hunan, 410073, China.

1544-3566/2022/03-ART15 $15.00
https://doi.org/10.1145/3498730

1 INTRODUCTION

On modern systems, from mobile devices to supercomputers, data movement has become one of the most significant costs in terms of both time and energy. The primary mitigation is caching. Modern caches are managed transparently to software: the workload itself cannot determine when and which cacheline to evict. This transparency eases programmability and ensures program portability, but as a result a program has no direct control of caching and cannot use program information to directly improve cache management.

Recent machines supports either direct memory control or automatic caching. On GPUs, the shared memory is program controlled. On Knights Landing, the near memory (MCDRAM) can operate either in the *Cache Mode*: the MCDRAM works as the LLC cache of the DRAM shared by all processors; and the *Flat Mode*: the MCDRAM acts as an extra NUMA node. Applications could control which node to put its data via `hbw_malloc()/malloc()`, or move data between these two with the kernel support, e.g., `numactl`. The new non-volatile memory from Intel operates in two similar modes. So far, the choice is mutually exclusive: one should select either of these two modes during system boot.

An alternative is programmable cache, or software-managed cache [36], which we study in this article. Programmable cache is the cache in which applications can determine which block will be evicted to make room for new data. Recently, Li et al. [44] proposed a new cache design called a *Lease Cache*. In a lease cache, each cacheline is assigned a non-negative integer *lease*, which represents the amount of logical time that it should reside in cache. If a cacheline is accessed before its lease is up, then its lease is refreshed. Otherwise, the cacheline is evicted at the end of its lease. This concept is growing in relevance, e.g., Twitter's *Time-To-Live Cache* [71].

The lease cache has a variable size. Variable cache allocation is interesting for two reasons. First, this is already the *status quo*. When the cache is shared, neither single application uses the whole cache, nor does it maintain a constant cache occupancy. Second, it permits greater synergy. The working set of an application is not a constant size and may benefit from a temporary increase in cache usage. The cache space may be dynamically traded among applications based on demand. In fact, variable-size allocation has been the standard solution used by operating systems to manage memory sharing [20, 24]. The lease cache enables program control, and program informed leases may improve not just performance but also performance predictability in large shared caches.

This article presents a compiler algorithm, its theoretical optimality, and its practical potential:

— ***Compiler Assigned Reference Lease (CARL)***. CARL is a compiler algorithm to program the lease cache (Section 3). It uses a metric called ***profit per unit cost*** *(*PPUC*)* and greedily chooses the lease with the largest PPUC.

— *Theoretical properties*. We present the conditions and formal proof that CARL's greedy heuristic provides optimal program control (Section 4.1) and that this optimality ensures miss curve convexity (Section 4.2) and sub-partitioning monotonicity (Section 4.3).

— *CARL potential*. Through a simulation study using the loop-based PolyBench programs as well as SPEC 2017, we compare cache *utilization* to existing cache management techniques and show that the new solution is not just far superior to LRU performance, but also similar to or better than the optimal (and offline therefore impractical) OPT (Section 5). We also show a complete compiler solution in the appendix.

Any theory is limited by its assumptions. All our theoretical results assume unrestricted dynamic occupancy in the lease cache. While this cannot be always valid in an actual system, the theory formalizes key properties and clarifies their theoretical complexity. In addition, the theory is about program optimization *of* caching, not program optimization *for* caching (e.g., tiling, fusion). Nor does it address directly the general problem of I/O complexity [28, 37] or data layout optimization [42, 53]. Furthermore, the evaluation shows only the potential but not what is realizable on an actual system. Implementation problems are being addressed in a hardware prototype with a fix-size, single-level, and unshared cache [54]. Here, we show the potential of compiler optimization across all cache sizes.

2 BACKGROUND

2.1 Lease Cache

In a lease cache, each cacheline is managed by a *lease*, which determines how long it is allowed to stay in cache. The lease is measured by the number of accesses rather than the physical time. A lease of 1,000 means that the lease cache keeps the data block until 1,000 accesses later. The lease is renewed if the data block is accessed before the end of the lease; otherwise, the block is evicted from the cache. Conventional cache design is reactive—deciding which to replace when the cache is full, but lease cache is prescriptive. The eviction time of a data block is *prescribed* by a lease at its latest access. The lease is a programming interface for cache allocation. By assigning leases to memory accesses, software may leverage program information to improve cache performance. Optimal leasing ensures that data remains cached, while it is useful and is evicted at the end of its usefulness.

2.2 Reference Leases

A reference is defined as the instruction, which invokes a memory access, i.e., the program counter for the load/store instruction. Given a program, a compiler assigns a lease to each reference. When the program executes, each access is given the lease of its reference. The lease of each reference is derived from two attributes for a reference: its ***Reuse Interval Distribution (D)*** and its ***Access Ratio (AR)***. Next, we first define one basic term **Reuse Interval (RI)**, then define these two per-reference attributions.

Definition 2.1 (Reuse Interval (RI)). An RI is defined as the change in logical time between a data block's use and its reuse. Suppose, we have a trace *abccba*, the reuse interval of the datum *a* is $RI = 5$.

Definition 2.2 (Reuse Interval Distribution (D)). A (D) is the distribution of RI's among all of its accesses. Using the same trace *abccba* given in previous definition and assuming there is only one reference, the RI distribution of this reference would contain three different reuses 1, 3, and 5 caused by the access to datum *c*, *b*, and *a*, respectively, each accounting for 1/3.

Definition 2.3 (Access Ratio). The Access Ratio of a reference is the portion of all accesses in the trace, which are invoked by that reference.

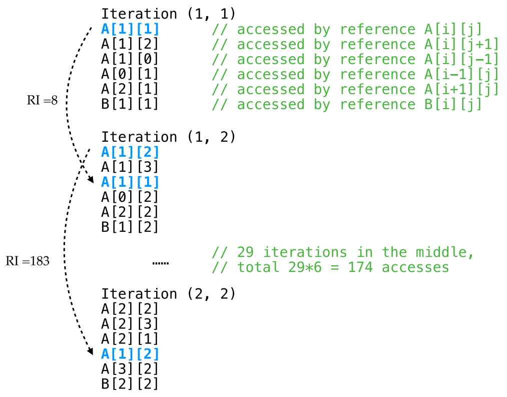
Figure 1(a) shows the loop nest of a 5-point stencil kernel, which appears commonly in scientific computing and image processing. It contains six references, from `A[i][j-1]` to `B[i][j]`. This program is used as a running example throughout this section and the next section. Figure 1(a) also shows the reuse interval distributions for 4 (out of 6) references in the stencil kernel. The remaining two references, `B[i][j]`, `A[i-1][j]`, have no reuses during the execution, hence omitted. For illustration purposes, the RIs in Figure 1(a) are computed at element (rather than data block) granularity. In Figure 1(b), we use `A[i][j]` as an example demonstrating where these RIs come from. RIs of other references can be inferred in a similar way. In the rest of this article, we use the notion $D[i]$ to represent the fraction of RIs whose value is i, e.g., for reference `A[i][j]`, 96.77% of the data form reuses with RI= 8, written as $D[8] = 96.77\%$.

Consider a lease assignment strategy for the stencil loop. Almost all data accesses from the `A[i][j]` and `A[i][j+1]` references will be reused almost immediately in the next iteration of the

```
// Assign lᵢ to refᵢ, i in 1 … 6
for (i = 1; i < 31; i++) {
   for (j = 1; j < 31; j++) {
      // ref₆ = ref₁ + ref₂ + ref₃ + ref₄ + ref₅
      B[i][j] = A[i][j] + A[i][j+1] + A[i][j-1]
               + A[i-1][j] + A[i+1][j];
   }
}
```

A[i][j]		A[i][j+1]		A[i][j-1]		A[i+1][j]	
ri	*D[ri]*	*ri*	*D[ri]*	*ri*	*D[ri]*	*ri*	*D[ri]*
8	96.77%	5	100%	175	100%	171	96.67%
183	3.23%	-	-	-	-	176	3.33%

(a) 5-point stencil code snippet with the RI distribution D of each references, from `A[i][j]` to `A[i+1][j]`.



```
Iteration (1, 1)
A[1][1]      // accessed by reference A[i][j]
A[1][2]      // accessed by reference A[i][j+1]
A[1][0]      // accessed by reference A[i][j-1]
A[0][1]      // accessed by reference A[i-1][j]
A[2][1]      // accessed by reference A[i+1][j]
B[1][1]      // accessed by reference B[i][j]

Iteration (1, 2)
A[1][2]
A[1][3]
A[1][1]
A[0][2]
A[2][2]
B[1][2]

      ......     // 29 iterations in the middle,
                 // total 29*6 = 174 accesses

Iteration (2, 2)
A[2][2]
A[2][3]
A[2][1]
A[1][2]
A[3][2]
B[2][2]
```

(b) Demonstration of all RIs for reference `A[i][j]`

Fig. 1. The RI distribution example using the 5-point stencil program.

inner loop. Hence, a short lease for these references will grant many cache hits while using a low amount of cache space. If the cache size is large enough, it may also be worthwhile to assign a long lease to the `A[i][j-1]` and `A[i+1][j]` references, whose reuse is in the next iteration of the outer loop. In any case, it is sensible to assign leases which grant many cache hits with a low space cost before assigning leases which grant fewer cache hits or take up more space.

3 THE CARL ALGORITHM

We describe the CARL algorithm, which assigns leases to each reference.

3.1 Profit Per Unit Cost

CARL uses a metric for references called PPUC. Profit is the hit ratio. All accesses whose reuse interval is less than or equal to the chosen lease are cache hits. If D is the RI distribution of a reference and l is the chosen lease, then its corresponding contribution to the program's hit ratio would be:

$$\mathcal{H}(D,l) = \sum_{i=0}^{l} D[i] \cdot \text{AccessRatio}. \tag{1}$$

The cost of a reference is its average cache use. We use the terms cache use, occupancy, allocation, and consumption interchangeably in the rest of this article. If data from a reference takes up three cachelines on average throughout execution, the cost is 3. The cost of a reference is proportional to the average resident time among its accesses. The resident time of an access depends on its RI and lease. If its reuse interval ri is less than the lease l, then the cacheline's lease will be refreshed by a new access after ri units of time. Otherwise, an access is resident in cache until its lease expires. Thus, the average cache use of a reference with lease l can be computed as the sum of these two values over the reference's RI distribution, shown in Equation (2):

$$C(D,l) = \left(\sum_{i=0}^{l-1} i \cdot D[i] + \sum_{i=l}^{RI_{max}} l \cdot D[i] \right) \cdot \text{AccessRatio}. \tag{2}$$

ALGORITHM 1: CARL main loop. Functions C (cost) and ΔPPUC are defined in Equations (2) and (3).

```
Input  : The number of references $R$ and reuse interval distributions $D_{1...R}[1...RI_{max}]$
Input  : Target cache size $S$
Output : Reference leases $L[1...R]$
Function Main():
    $L[1...R] \leftarrow 0$;
    $cacheUse \leftarrow 0$;
    while $cacheUse < S$ do
        $(ref, l_{new}) \leftarrow \underset{r \in 1...R,\ l \in L[r]...RI_{max}}{\text{argmax}} \{\triangle PPUC(D_r, L[r], l)\}$;
        $l_{old} \leftarrow L[ref]$;
        if $l_{new} = l_{old}$ then
            /* No more lease to assign */
            break;
        else
            $L[ref] = l_{new}$;
            $cacheUse += C(D_{ref}, l_{new}) - C(D_{ref}, l_{old})$;
        end if
    end while
End
```

In Figure 2(a), we compute the Profit and Cost of all four references in the given 5-point stencil example. Then, the value of PPUC, as the name implies, is the ratio of cache hits and the corresponding cache occupancy. CARL is a greedy algorithm, which iteratively increases leases. It may assign a lease for a reference and then increase that lease later on, so we are interested in the change in PPUC from one lease to another. This is denoted as $\Delta PPUC$. Let D_A be the RI distribution for reference A. The $\Delta PPUC$ when increasing its lease from l to l' is given by

$$\Delta PPUC(A, l, l') = \frac{\mathcal{H}(D_A, l') - \mathcal{H}(D_A, l)}{C(D_A, l') - C(D_A, l)}. \tag{3}$$

3.2 CARL Lease Assignment

Algorithm 1 shows the main loop of the CARL algorithm. CARL takes a target cache size and set of reference RI distributions as input. It initializes a lease of 0 for each reference, and keeps track of the cache use given its set of leases, which is initially 0. Then, in a loop, CARL updates the leases in a greedy fashion: in each step, among all references, it selects the lease that is the most profitable, i.e., the lease with the maximum ΔPPUC. Then the cache occupancy is updated according to the change in cost of the lease. CARL continues this process until (1) the cache use has reached some target value, i.e., the target cache size, or (2) all references have been assigned the maximum possible lease.

The CARL lease assignment process for the 5-point stencil example is demonstrated in Figure 2(b). At the very beginning, all references are initialized to lease 0. In the first step, CARL decides to assign lease 5 to reference `A[i][j+1]`, since it has the highest $PPUC = 16.7\%/0.83$. In the next step, CARL considers whether to assign a longer lease for `A[i][j+1]` or a lease for another reference and decides to assign lease 8 to reference `A[i][j]` since its $PPUC = 16.7\%/1.33$.

3.3 Dual Leases

When CARL assigns a new lease, it updates its cache use according to the cost of that lease. If the running cache size is less than the target cache size, there is no issue and lease assignment

`A[i][j]`			`A[i][j+1]`			`A[i][j-1]`			`A[i+1][j]`		
Lease	*Hit Ratio*	*Cache Use*	*Lease*	*Hit Ratio*	*Cache Use*	*Lease*	*Hit Ratio*	*Cache Use*	*Lease*	*Hit Ratio*	*Cache Use*
8	16.13%	1.33	5	16.67%	0.83	175	16.67%	29.17	171	16.11%	28.5
183	16.67%	2.28	-	-	-	-	-	-	176	16.67%	28.53

(a) The hit ratio and the cache use for 4 references in the 5-point stencil program. Since there are 6 references and each appears once, the access ratio of each reference is 1/6. It shows the benefit (Hit Ratio), and the cost (Cache Occupancy) of each lease candidate. Taking `A[i][j]` for example, if we assign it with lease 8, though 96.77% of the accesses from will `A[i][j]` will be a cache hit, the overall hit ratio will be 96.77%/6 = 16.13%. Similarly, the cache use will grow to 8/6=1.33.

Step	Lease assigned for each reference				Average Cache Use	ΔPPUC			
	`A[i][j]`	`A[i][j+1]`	`A[i][j-1]`	`A[i+1][j]`		`A[i][j]`	`A[i][j+1]`	`A[i][j-1]`	`A[i+1][j]`
0	0	0	0	0	0	16.13% / 1.33	$\boxed{16.67\% / 0.83}$	16.67% / 29.17	16.11% / 28.5
1	0	5	0	0	0.83	$\boxed{16.13\% / 1.33}$	0	16.67% / 29.17	16.11% / 28.5
2	8	5	0	0	2.16	...	...	...	...
...	...	...	...	...	...	...	...	...	...

(b) Greedy lease assignment process in CARL for the stencil example. In each step, CARL chooses the lease with the highest ΔPPUC (Those enclosed by the blue box).

Fig. 2. CARL demonstration using the 5-point stencil program. This greedy algorithm relies on (1) the per-reference RI distributions (Figure 1), and (2) the hit ratio and the cache use of each lease candidate (Figure 2(a)) computed from the per-reference RI distributions using Equations (1) and (2).

continues. However, there may be a dilemma where, if the lease is assigned, the cache is over allocated, but if it is not assigned, the cache is under allocated. The solution is to assign a long lease for a portion of accesses and a short lease for the rest. The division point is selected such that the target cache size is exactly matched.

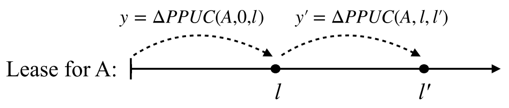
CARL allocates one reference at a time. The effect of the allocation is an aggregate of all its accesses. This is equivalent to allocating the lease in many steps, one for each access. Using dual leases, CARL may effectively stop assigning leases to accesses partway through in the event that the target cache size is exceeded. The upper bound on cache use for a single access is one cache block over the entire program execution, i.e., with a lease equal to the trace length and no reuse. Therefore, CARL may allocate cache within one block of any target cache size.

4 A THEORY OF OPTIMAL CACHE PROGRAMMING

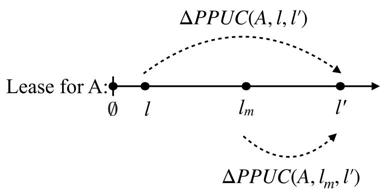
This section proves CARL optimality, miss curve convexity, and sub-partitioning monotonicity.

4.1 Optimality

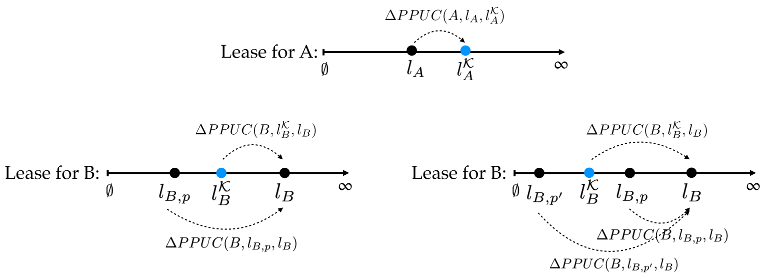
Optimality means maximal cache utilization, i.e., most hits for the given cache size. CARL greedily assigns leases by decreasing PPUC. In the simple case, when the PPUC of leases are independent from each other, the greedy algorithm is trivially optimal. However, in general the PPUC of leases



(a) Let l, l' be two leases selected by CARL and CARL select first l then l'. PPUC monotonicity states $y \geq y'$



(b) Let l, l' be two leases selected by CARL and CARL select first l then l'. The Midpoint Lemma states $\Delta PPUC(A, l_m, l') > \Delta PPUC(A, l, l')$


$\Delta PPUC(A,l_A,l_A^{\mathcal{K}})$
Lease for A:
$\emptyset$
l_A
$l_A^{\mathcal{K}}$
∞
$\Delta PPUC(B,l_B^{\mathcal{K}},l_B)$
Lease for B:
$\emptyset$
$l_{B,p}$
$l_B^{\mathcal{K}}$
l_B
∞
$\Delta PPUC(B,l_{B,p},l_B)$
$\Delta PPUC(B,l_B^{\mathcal{K}},l_B)$
Lease for B:
$\emptyset$
$l_{B,p'}$
$l_B^{\mathcal{K}}$
$l_{B,p}$
l_B
∞
$\Delta PPUC(B,l_{B,p},l_B)$
$\Delta PPUC(B,l_{B,p'},l_B)$

(c) Black dots represent leases chosen by CARL, and blue dots are those leases chosen by policy $\mathcal{K}$. For references A, B, CARL assigns leases l_A, l_B, an alternative policy $\mathcal{K}$ assigns $l_A^{\mathcal{K}}, l_B^{\mathcal{K}}$, and CARL leases are the best as $\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$. Through its intermediate steps, CARL assigns $l_{B,p'}$ and $l_{B,p}$ before l_B. Figure on the left shows the condition when $l_B^{\mathcal{K}} > l_{B,p}$, and figure on the right demonstrate the case when $l_B^{\mathcal{K}} < l_{B,p}$.

Fig. 3. Symbols used in (a) PPUC monotonicity, (b) the midpoint lemma, and (c) CARL optimality.

are not independent. Consider each lease as a cache allocation. It may happen that a sub-optimal allocation will enable a more profitable allocation later on, hence a need to backtrack.

To show that CARL is optimal, We first give two properties of CARL, (1) PPUC Monotonicity, and (2) the Midpoint Lemma, to prove (3) CARL Optimality. Figure 3 shows a diagram for each proof to illustrate the symbols used in the proof.

CARL may not exactly "match" a target cache size, but with dual leases (Section 2), it can allocate a cache size within a single block of the target (assuming the target size is no greater than the data size). The optimality is for only the cache sizes that are produced by CARL.

By choosing the maximal $\Delta PPUC$, CARL attempts to maximize the profit gain at each step. A problem, however, is that CARL recomputes $\Delta PPUC$s after each assignment, so it may be possible that a greater $\Delta PPUC$ appears later. The following theorem rules out such a possibility.

Theorem 4.1 (PPUC Monotonicity). *Let y be the maximal $\Delta PPUC$ at a CARL step, and y' the maximal $\Delta PPUC$ at the next step, then $y \geq y'$.*

Proof. We first consider the case that y, y' are for two different references A, B. After the current step, the $\Delta PPUC$s stay unchanged for B (only $\Delta PPUC$s for A may change). We must have $y \geq y'$; otherwise, the current step would have selected y' instead of y.

We next consider the case when in both steps CARL selects a lease for the same reference A, i.e., first the largest $\Delta PPUC$ y at lease l and then $\Delta PPUC$ y' at l'. We will use the definition of $\Delta PPUC$

given in Equation (3). We assume that the lease assigned for A before l is l_{old}. Naturally $l_{old} < l < l'$, $y = \Delta PPUC(A, l_{old}, l)$, and $y' = \Delta PPUC(A, l, l')$. We now prove $y \geq y'$ by contradiction.

Assume the opposite, $y' > y$. Let $y'_{old} = \Delta PPUC(A, l_{old}, l')$, i.e., the PPUC gain at the lease l' before the update. Because CARL chooses lease l over l' in the current step, we have $y \geq y'_{old}$. Combining the two inequalities yields $y' > y \geq y'_{old}$. By substituting the formula for computing $\Delta PPUC$, we have:

$$\frac{\mathcal{H}(A,l') - \mathcal{H}(A,l)}{C(A,l') - C(A,l)} > \frac{\mathcal{H}(A,l)}{C(A,l)} \geq \frac{\mathcal{H}(A,l')}{C(A,l')}, \tag{4}$$

where $\mathcal{H}(A,l)$ and $C(A,l)$ represent hit ratio (Equation (1)) and average lease (Equation (2)) for reference A, respectively, when assigned lease l. We now show that the first inequality contradicts with the second inequality. Rewriting the first inequality, we have

$$\frac{\mathcal{H}(A,l)}{C(A,l)} \frac{\left(\frac{\mathcal{H}(A,l')}{\mathcal{H}(A,l)} - 1\right)}{\left(\frac{C(A,l')}{C(A,l)} - 1\right)} > \frac{\mathcal{H}(A,l)}{C(A,l)}.$$

Since $\frac{\mathcal{H}(A,l)}{C(A,l)} > 0$, and $C(A,l') > C(A,l)$, by re-arranging the terms, we have $\frac{\mathcal{H}(A,l)}{C(A,l)} < \frac{\mathcal{H}(A,l')}{C(A,l')}$, which is the opposite of the second inequality. The two inequalities contradict; therefore, the assumption is wrong, and we must have $y \geq y'$.

A careful reader may notice that the derivation assumes $l_{old} = 0$, which means l and l' are the first two leases assigned for reference A. In the general case, a previous lease l_{old} has been assigned before l. This necessitates two changes to the PPUC calculation: subtracting $\mathcal{H}(A, l_{old})$ from the numerator and $C(A, l_{old})$ from the denominator. Since this change happens on every fraction in Equation (4), the inequality still holds. □

Next, we show a lemma about the PPUC calculation. For a reference A, assume that CARL chooses first l and then l' as leases. Let l_m be a midpoint, i.e., $l < l_m < l'$. The midpoint lemma says that the incremental PPUC from the midpoint to l' is greater than that from l to l'. A pictorial view can be seen in Figure 3(b).

LEMMA 4.2 (MIDPOINT LEMMA). *Let l, l' be the two successive assignments that CARL selects for reference A, and l_m be a mid-point, i.e., $l < l_m < l'$. We have*

$$\Delta PPUC(A, l_m, l') > \Delta PPUC(A, l, l').$$

PROOF. We define $\mathcal{H}_m$ as the additional cache hits, and C_m the extra cost, when increasing the lease from l to l_m. When increasing the lease from l_m to l', let the two changes be $\Delta\mathcal{H}, \Delta C$, respectively. Because CARL selects l' instead of l_m, we know that $\Delta PPUC(A, l, l') > \Delta PPUC(A, l, l_m)$, that is, $\frac{\mathcal{H}_m + \Delta\mathcal{H}}{C_m + \Delta C} > \frac{\mathcal{H}_m}{C_m}$.

We split the proof into two steps: First, we prove that $\frac{\Delta\mathcal{H}}{\Delta C} > \frac{\mathcal{H}_m}{C_m}$ and then we prove the desired inequality.

Following the same idea from the proof of Theorem 4.1, we lift $\mathcal{H}_m$ and C_m from both the denominator and numerator on both sides.

$$\frac{\mathcal{H}_m + \Delta\mathcal{H}}{C_m + \Delta C} > \frac{\mathcal{H}_m}{C_m} \Rightarrow \frac{\mathcal{H}_m}{C_m} \cdot \frac{1 + \frac{\Delta\mathcal{H}}{\mathcal{H}_m}}{1 + \frac{\Delta C}{C_m}} > \frac{\mathcal{H}_m}{C_m} \Rightarrow \frac{\Delta\mathcal{H}}{\Delta C} > \frac{\mathcal{H}_m}{C_m}.$$

Given this conclusion, we can now derive that $\frac{\Delta\mathcal{H}}{\Delta C} > \frac{\mathcal{H}_m+\Delta\mathcal{H}}{C_m+\Delta C}$

$$\frac{\Delta\mathcal{H}}{\Delta C} > \frac{\mathcal{H}_m}{C_m} \Rightarrow 1 + \frac{C_m}{\Delta C} > 1 + \frac{\mathcal{H}_m}{\Delta\mathcal{H}}$$
$$\Rightarrow \frac{C_m + \Delta C}{\Delta C} > \frac{\mathcal{H}_m + \Delta\mathcal{H}}{\Delta\mathcal{H}} \Rightarrow \frac{\Delta\mathcal{H}}{\Delta C} > \frac{\mathcal{H}_m + \Delta\mathcal{H}}{C_m + \Delta C}. \qquad \square$$

The lemma shows that choosing a midpoint is unwise, because we can always increase PPUC by choosing CARL's next choice (l' in the lemma). Next, we show that no alternative lease assignment algorithm can outperform CARL, known as the CARL Optimiality. The proof compares CARL lease l with another lease assignment $l^{\mathcal{K}}$. Since they differ, inevitably there are at least two references A, B such that CARL gives A a longer lease and B a shorter lease. The proof uses PPUC monotonicity and the midpoint lemma to show that no other lease assignment can outperform CARL.

Theorem 4.3 (CARL Optimality). *No algorithm which determines the leases for a set of references using only their reuse interval distribution can have a lower miss ratio than CARL.*

Proof. Since the lease is determined solely from the reuse interval distribution, any algorithm must assign the same lease to all accesses of a reference (assuming the algorithm is deterministic). Let m be the number of references. Let l_G be the leases that CARL assigns to a reference G. Let $l_G^{\mathcal{K}}$ be the lease to G assigned by an alternative policy $\mathcal{K}$. Now, we would like to prove that, with the same average cache size these two policies will occupy, alternative policy $\mathcal{K}$ could not outperform CARL.

First, we consider the simplest case where the two policies differ in only two lease assignments, i.e., for two references A and B, CARL chooses l_A, l_B, while $\mathcal{K}$ chooses $l_A^{\mathcal{K}}$ and $l_B^{\mathcal{K}}$ (as visualized in Figure 3(c)). Since these two policies will occupy the same amount of cache, we must have $l_A < l_A^{\mathcal{K}}$ and $l_B > l_B^{\mathcal{K}}$. We show $\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$: For the same cache occupancy, increasing the lease from $l_B^{\mathcal{K}}$ to l_B provides at least as many cache hits as it does by increasing the lease from l_A to $l_A^{\mathcal{K}}$. In other words, switching the policy from $\mathcal{K}$ to CARL has the same cache occupancy without losing any cache performance.

For reference B, let $l_{B,p}$ be the lease selected by CARL before l_B. From PPUC monotonicity (Theorem 4.1), PPUC gained by CARL's new lease selection is no smaller than those of the remaining lease candidates. Since CARL favors l_B over $l_A^{\mathcal{K}}$, we have $\Delta PPUC(B, l_{B,p}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$, i.e., every lease that CARL chooses is at least as good as any lease that CARL does not choose. We prove the inequality, $\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$, in three cases:

- If $l_B^{\mathcal{K}} = l_{B,p}$, then $\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) = \Delta PPUC(B, l_{B,p}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$, and the conclusion holds.
- If $l_B^{\mathcal{K}} > l_{B,p}$, then applying the mid-point lemma (Lemma 4.2), we have:

$$\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) > \Delta PPUC(B, l_{B,p}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}}).$$

- If $l_B^{\mathcal{K}} < l_{B,p}$, let $l_{B,p'}$ the CARL's closest choice to $l_B^{\mathcal{K}}$, which is not greater, as shown on the right-hand side of the Figure 3(c). From the monotonicity theorem and trivial algebra,[1] we know that $\Delta PPUC(B, l_{B,p'}, l_B) \geq \Delta PPUC(B, l_{B,p}, l_B)$. From the mid-point lemma, we have

[1] $\frac{a_1}{c_1} \geq \frac{a_2}{c_2} \rightarrow \frac{(a_1+a_2)}{(c_1+c_2)} \geq \frac{a_2}{c_2}$ for all $a_n, c_n > 0$. Profit and cost must of course both be greater than 0 for a selected lease. The proof can be done by lifting a_2, c_2, like what we did in mid-point lemma proof.

$\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) > \Delta PPUC(B, l_{B,p'}, l_B)$. Combining these two together, we have

$$\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) > \Delta PPUC(B, l_{B,p'}, l_B) \geq \Delta PPUC(B, l_{B,p}, l_B)$$
$$\Rightarrow \Delta PPUC(B, l_B^{\mathcal{K}}, l_B) > \Delta PPUC(A, l_A, l_A^{\mathcal{K}}).$$

In the general case, the lease differs in any number of references. Let $\mathcal{B}$ be all references B where CARL assigns a longer lease than $\mathcal{K}$ does, and $\mathcal{A}$ the opposite. Both policies have the same cache occupancy, which means:

$$\sum_{\substack{B \in \mathcal{B} \\ (l_B > l_B^{\mathcal{K}})}} \left(C(B, l_B) - C(B, l_B^{\mathcal{K}}) \right) = \sum_{\substack{A \in \mathcal{A} \\ (l_A^{\mathcal{K}} > l_A)}} \left(C(A, l_A^{\mathcal{K}}) - C(A, l_A) \right).$$

We show that the total gain of the hit ratio from the cache occupancy on the left is the same as or higher than the total gain from the cache occupancy on the right:

$$\sum_{B \in \mathcal{B}} \Delta PPUC(B, l_B^{\mathcal{K}}, l_B)(C(B, l_B) - C(B, l_B^{\mathcal{K}})) \geq \sum_{A \in \mathcal{A}} \Delta PPUC(A, l_A, l_A^{\mathcal{K}})(C(A, l_A^{\mathcal{K}}) - C(A, l_A)).$$

The left sum includes all hits when CARL assigns a longer lease than $\mathcal{K}$ does, and the right sum includes all hits when $\mathcal{K}$ assigns a longer lease than CARL does. For any reference B on the left and A on the right, using the same proof from the two-reference case, we have $\Delta PPUC(B, l_B^{\mathcal{K}}, l_B) \geq \Delta PPUC(A, l_A, l_A^{\mathcal{K}})$. Therefore, the lowest PPUC for any B is at least as large as the highest PPUC for any A. Since both sides of the equation have the same number of units of cache use, the sum on the left is no lower than that on the right.

Therefore, for the same cache occupancy, CARL scores at least the same number of cache hits as any other policy, that is, CARL is optimal. □

The above proof compares CARL leases with a set of alternative leases. Informally, we may transform the alternative leases into the CARL leases by going through the CARL algorithm step by step. At each step, we may add to or subtract from the alternative lease to match the CARL lease. Since the CARL algorithm maximizes the incremental PPUC at each step, this transformation would never degrade the performance of the alternative lease. The process either maintains or improves the performance. The alteration stops at the end when the leases are identical to those of CARL. Therefore, the performance of the alternative leases (at the start of this process) must be either the same as or worse than CARL.

A reader may question how limiting is the theorem's assumption, i.e., that leases are assigned based on the reuse interval distribution. May better leases be found based on other information? To consider, we note that reuse information is all that is needed for optimal caching. The more precise the information, the better the result of optimization. Previous optimal solutions, e.g., OPT, have complete information—the forward reuse is known for every access. CARL is the optimal solution based on aggregate information, i.e., distributions of RIs rather than individual RIs. The restriction comes not from the CARL algorithm or the theory, but from the amount of information that is available. For the same reuse information, other kinds of schemes cannot do better than CARL. For any scheme to be better than CARL, it can only do so because it has more precise reuse information.

4.2 Miss Curve Convexity and Optimal Cache Sharing

For any program, the miss ratio curve of CARL is a convex function of the cache size. This convexity means that miss ratio decreases at the same or a lower rate as the cache size increases.

Theorem 4.4 (Convexity). *For cache sizes c_1, c_2, and $0 \le t \le 1$, we have:*

$$mr(t \cdot c_2 + (1-t) \cdot c_1) \le t \cdot mr(c_2) + (1-t) \cdot mr(c_1),$$

where mr(c) is the miss ratio at cache size c

Proof. The convexity is equivalent to proving $\frac{mr(c_1+t(c_2-c_1))}{t(c_2-c_1)} \le \frac{mr(c_2)-mr(c_1)}{c_2-c_1}$ for $t \neq 0$. By taking t to the smallest value, we have

$$mr(c_2) \ge mr(c_1) + \Delta mr(c_1)(c_2 - c_1).$$

Without the loss of generality, let's assume $c_1 \le c_2$. Let $c_{x_0} = c_1, c_{x_1}, c_{x_2}, \ldots, c_{x_m}, c_{x_{m+1}} = c_2$ be the steps of cache sizes of CARL, and $\Delta\text{PPUC}_0 \ldots \Delta\text{PPUC}_{m+1}$ be their $\Delta PPUC$s. Naturally $\Delta mr(c_{x_i}) = -\Delta\text{PPUC}_i$.

From PPUC Monotonicity, we have $\Delta\text{PPUC}_i \ge \Delta\text{PPUC}_{i+1}$. Considering $\Delta PPUC$s as slopes, the conclusion holds because the miss ratio $mr(c_2)$ computed using the series of slopes (ΔPPUC_i) is no smaller than the miss ratio computed using the steepest slope (ΔPPUC_0), or formally:

$$\begin{aligned} mr(c_2) &= mr(c_1) - \sum_{i=0}^{m}(c_{x_{i+1}} - c_{x_i})\Delta\text{PPUC}_i \\ &\ge mr(c_1) - \sum_{i=0}^{m}(c_{x_{i+1}} - c_{x_i})\Delta\text{PPUC}_0 \\ &\ge mr(c_1) + \Delta mr(c_1)(c_2 - c_1). \end{aligned}$$ □

In the proof, the calculation of $mr(c_2)$ also shows monotonicity. Since $\Delta\text{PPUC}_i \ge 0$, the miss ratio curve is monotonically non-increasing.

We have proved that the CARL miss curve convexity follows trivially from PPUC monotonicity. For fixed-sized caching, OPT was formulated as a stack algorithm and proved optimal in 1970 [49]. The miss ratio curve of a stack algorithm is monotone but not necessarily convex. LRU is a well known example whose miss ratio curve is not convex. The OPT convexity was not proved until nearly half century later, as a corollary in 2015 (with assumptions) [7] and formally and unconditionally in 2016 [51], as a main result of a 20-page article.

Convexity is useful in cache partitioning. In a cache shared by a set of competing processes, the optimal fixed allocation is one "for which the miss-rate derivative ... is equal for all processes", under the condition that their miss ratio curves be convex [58]. Practical caching techniques such as LRU do not satisfy this condition. Two solutions have been developed. The first is to "induce" convexity. Motivated in part by the optimal-partitioning problem being NP-complete, Talus was developed to remove "performance cliffs" and make the miss ratio curve convex [7]. SLIDE uses scaled down simulation to achieve convex miss ratio curves for a broad class of policies including ARC, 2Q, and LIRS [63]. Second, with the aid of the **higher-order theory of locality** (**HOTL**), optimal partitioning can be solved approximately for the LRU cache using dynamic programming (with a time cost linear to the number of programs but quadratic to the size of the cache) [14, 72].

In addition to cache partitioning, there are other problems of cache sharing. Suh et al. [60] developed optimization of time sharing using an analytical cache model. It assumed that the miss ratio curve is convex. Using HOTL, Brock et al. [14] and Ye et al. [72] computed optimal elastic cache partition sharing. Hu et al. [38] computed optimal program symbiosis and fair scheduling in shared cache. Its model, **average eviction time** (**AET**), is mathematically equivalent to HOTL, as shown by [73] in the relational theory of locality. For optimization, both techniques used dynamic programming.

Because of the convexity guaranteed by CARL, these cache-sharing problems are simpler to solve in the lease cache than in the LRU cache, although we do not solve these problems in this article.

4.3 Sub-partitioning Monotonicity

Using reference leases raises a practical question. For two references in a program, a compiler may use CARL to assign them each a separate lease, or the compiler may treat them as a single reference and assign a single lease for both. Sub-partitioning monotonicity means that assigning two leases is always the better option. Sub-partitioning cannot do worse.

Consider all partitions of a set of accesses. The finer-than relation forms a lattice. The top of the lattice is the set of all accesses, and the bottom the partition with each access in its singleton set. CARL guarantees *Sub-partitioning Monotonicity*, which means that for any cache size, the miss ratio of a finer partition is never greater than the miss ratio of the original partition. The following theorem proves sub-partitioning monotonicity for two partitions with the direct finer-than relation in the partition lattice.

Theorem 4.5 (Sub-partitioning Monotonicity). *Consider any program and any CARL lease assignment. Let r be a set of accesses in the original partition, and r_1, r_2 its two sub-sets in the finer partition, i.e., $r_1 \cup r_2 = r, r_1 \cap r_2 = \emptyset$. For any cache size c, let the CARL lease be l for r and l_1, l_2 for r_1, r_2, respectively. Let $mr_l(c)$ be the original miss ratio of r accesses, and $mr_{l_1,l_2}(c)$ the new combined miss ratio of r_1, r_2 accesses. Then we have*

$$mr_l(c) \geq mr_{l_1,l_2}(c),$$

for all subsets r_1, r_2, and cache size $c \geq 0$.

Proof. Let the miss ratio be $mr_{l,l}(c)$ if we assign the same lease l for r_1, r_2. Naturally, we have $mr_l(c) = mr_{l,l}(c)$. Applying the CARL optimality theorem (Theorem 4.3), we have $mr_{l,l}(c) \geq mr_{l_1,l_2}(c)$. Combining the two results, we have $mr_l(c) \geq mr_{l_1,l_2}(c)$. □

The proof shows the monotonicity between two partitions with a direct finer-than relation in the partition lattice. If we apply the theorem on every relation, we have the monotonicity between any two partitions that have a transitive finer-than relation. For example, a set of accesses may be divided into any number of sub-sets.[2]

CARL sub-partitioning monotonicity follows trivially from its optimality. We may draw an analogy with Jensen's inequality, which is the property between the average values of mathematical functions. While Jensen's inequality is based on convexity, the CARL monotonicicy follows from its optimality.

The implication of the result is not entirely trivial. When a finer partition divides a set of accesses into two subsets, it may weaken the precision in one of the sets but cannot worsen the cache performance. Any blunting in one subset must be countered by sharpening in the other. We define *precision* by how precise we can predict the next reuse at the time of an access. An RI distribution makes a probablistic prediction. The greater the probability with which we can predict a reuse, the more precise is the prediction.

In sub-partitioning, a set may be divided into any number of sub-sets. One or more of the sub-sets may have a lower precision and make their prediction less certain. The monotonicity theorem, however, ensures that the overall effect is never negative.

[2]The theorem can be proved without assuming optimality, but the proof is non-trivial (over a page long), which we do not include here.

Table 1. The Number of References and the Compilation Time Measured in Milliseconds ($10^{-3}s$) for the PolyBench Benchmarks

Tests I	#References	Time	Tests II	#References	Time
2 mm	11	.212	mvt	8	.885
3 mm	15	.152	seidel_2d	10	.219
adi	34	30.8	syrk	6	.203
atax	10	.257	trisolv	9	.602
bicg	10	.136	cholesky	13	27.4
deriche	20	23.1	covariance	16	16.3
doitgen	7	.191	correlation	34	25.5
durbin	9	9.29	floyd_warshall	7	.253
fdtd_2d	16	12.0	gramschmidt	15	$2 \cdot 10^4$
gemm	6	.081	lu	11	149
gemver	17	3.93	ludcmp	18	176
gesummv	13	.169	nussinov	18	6.50
heat_3d	22	.754	symm	10	243
jacobi_1d	8	.087	syr2d	8	177
jacob_2d	12	.192	trmm	6	35.8
Average #references		13	Average compilation time		32.0

Sub-partitioning is the refinement of program information. CARL provides the guarantee that having more information can never degrade cache performance. Consider Belady's anomaly, which happens when a program incurs more misses in a larger cache than in a smaller cache [9]. While Belady's anomaly is a problem based on the available space for cache management, another anomaly may happen based on the available information for cache management. With CARL, neither anomaly can happen.

5 EVALUATION

This section evaluates the potential of our reference lease compiler using the benchmark suite PolyBench and SPEC CPU 2017.

5.1 Experimental Setup

5.1.1 Candidate Caching Techniques. Practical caching techniques use fixed-size replacement policies. We use the baseline `LRU` and the ideal policy `OPT`. `LRU` always replaces the least recently used block, and `OPT` looks at future accesses and replaces the block whose next reuse happens the furthest in the future. We measure the miss ratio for all cache sizes. Since CARL is a variable size policy, we show the miss ratio curves produced by CARL in terms of average cache consumption `RL-AVG` and the maximal consumption `RL-MAX`.

While modern techniques perform better than `LRU`, they are still far from `OPT`. With one of the most successful research prototypes called Hawkeye, Jain and Lin [40] reported achieving on average half of the improvement of `OPT` over `LRU`. Two later variants, Glider [57] and Parrot [46] further beat Hawkeye using machine learning models, i.e., **Support Vector Machine** (**SVM**) and Self-Attention, but they still lower than `OPT`. While the best automatic solutions are still unable to, we next see whether a compiler solution can reach the performance of the off-line ideal solution.

5.1.2 Measurements. All measurements are performed with a 64B cache line size. The LRU performance is measured by profiling the reuse distance. Most efficient methods are Zhong et al. [74]

in time complexity and Wires et al. [66] in space complexity. We use the approximation algorithm in [74]. We use 99% accuracy (measured reuse distance is guaranteed to be between 99% and 100% of actual). As for OPT, the measurement uses the setup from [33] based on the code from [59].

RL-AVG. Instead of a given target cache use, we run the CARL assignment loop until all references are assigned leases equal to their greatest RIs. We use the algorithm to compute the miss ratio and average cache use of each assignment. In addition, we connect two consecutive data points with a line, which shows the effect of dual leases (Section 2).

Table 1 lists the number of references for PolyBench and the compilation overhead of RL, measured by running CARL on a machine with Intel Xeon Silver 4114 processors with 20 cores and 16 GB memory. On average, the CARL algorithm adds 0.03s overhead (collecting RI distributions for each reference) on PolyBench with sampled RI distributions.

RL-MAX. We have developed an algorithm to measure the maximal cache use and implemented it in a simulator. Given reference leases and the access trace, the simulator maintains the actual cache consumption at each access by tracking the two ways it may change. (1) The cache occupancy increases by 1 at a cache miss. For all data blocks, a hash table stores the last access time, and a second hash table stores the lease at the time of the last access. (2) The cache consumption decreases when a lease expires. To track expiration, an expiration counter is allocated and maintained for each future time t when there is at least one block that expires at t. When the execution reaches time t, the counter value is deducted from the cache size, and its memory is freed. In addition, at a cache hit, the lease is renewed, which means decrementing the previous expiration counter in addition to incrementing the current expiration counter.

Static Analysis vs Profiling. CARL requires the per reference RI distributions. The results in Figure 4 are measured using the accurate RI distributions obtained by profiling. CARL can be applied without profiling using SPS [19]. The results are shown in Appendix in Figure 8. In both profiling and SPS, CARL computes leases for all cache sizes by compiling a program only once. Chen et al. [19, Table 2] showed that the overhead of SPS to collect RI distributions increases linearly with the data size [19].

5.1.3 Benchmarks. We use PolyBench/C 4.2.1, which contains 30 numerical kernels extracted from linear algebra, image processing, physics simulation, dynamic programming, and statistics applications [47]. This benchmark suite has been extensively used. For example, Olivry et al. [52] use it to evaluate their static analysis that derives the I/O complexity lower bounds for affine programs; Abella-González et al. [1] developed their Python version to evaluate the profitability of using Python for regular numerical codes.

The data sizes are assigned according to the depth of nested loops in the program. We choose loop bound 1,024 for one and two nested loops, 256 for three nested loops, and 64 for four nested loops. We chose a relatively small size due to the time-consuming OPT simulation. Each element in an array is 8 bytes (A cache block contains 8 elements). In Section 5.4, we show results for fragments of SPEC 2017 program traces.

5.2 CARL vs. OPT

For each test program, Figure 4 shows three miss ratio curves for RL-AVG,[3] LRU, and OPT. Cold misses are included so the curves do not drop to 0. In addition, it shows the maximal cache consumption RL-MAX, which we discuss in the next section.

[3]RL-AVG shown in the figure does not violate the miss curve convexity (Section 4.2). Because we draw cache sizes in the logarithmic scale, what seems visually concave is actually convex if plotted with a linear-scale x-axis.

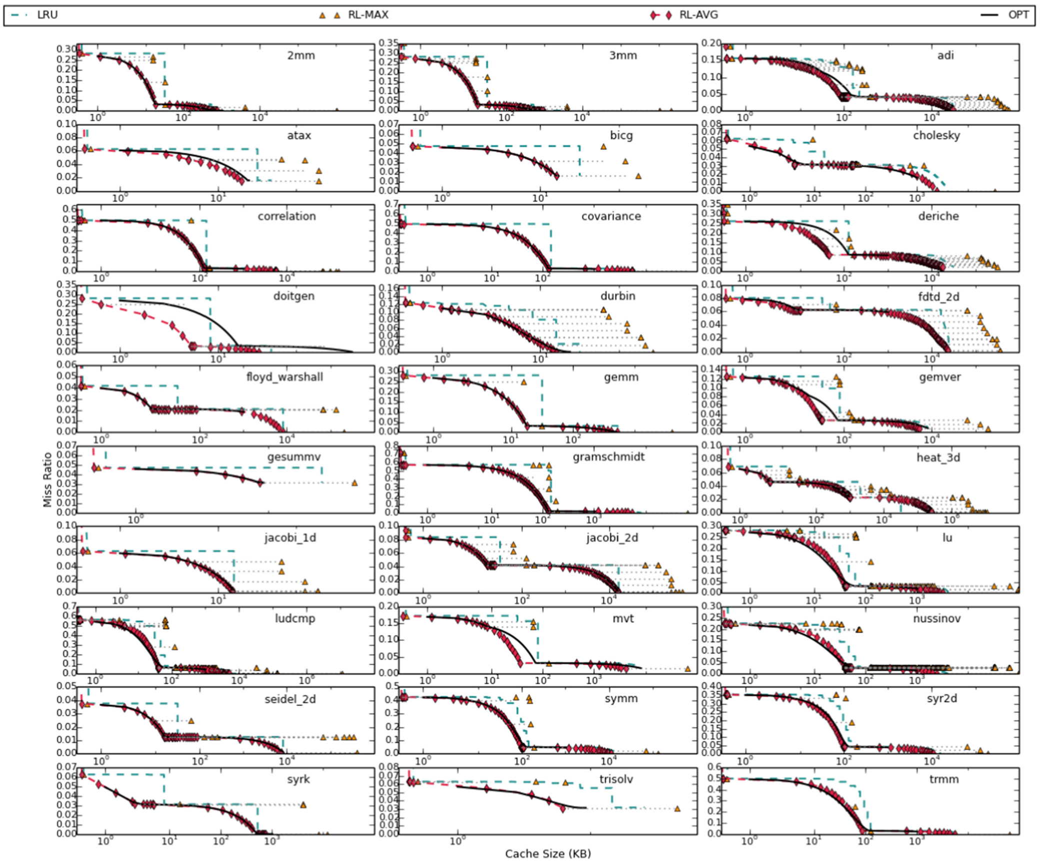


Fig. 4. Miss ratios of LRU, RL-AVG, and OPT on PolyBench benchmarks are shown as curves. The maximal cache sizes of CARL, RL-MAX, are shown as points. The *x*-axis are different cache sizes, unit in KB and the *y*-axis is their corrsponding cache miss ratio. Lower is better.

RL-AVG performs visibly better than OPT in 8 of the 30 tests: adi, atax, deriche, doitgen, gemver, mvt, syr2d, and trisolv, and slightly worse than OPT in 2 tests: luand trmm. In nussinov, RL-AVG is slightly worse in small cache sizes and better in large cache sizes. In the remaining 19 cases, the two are indistinguishable. It is fair to say that when measured by the average cache consumption or the cache utilization, CARL matches or exceeds OPT in performance.

While OPT is impractical, a recent technique called Hawkeye uses novel hardware support to imitate Belady (OPT for a single cache size) [40]. Hawkeye achieves half of the improvement of OPT over LRU. The study tested four other techniques which achieved a fraction of that of Hawkeye. Machine learning models are also adopted to improve the cache performance, e.g., Glider [57] and Parrot [46]. On average, Glider increases the hit ratio by 12.8% over LRU, while OPT increases it by 18.6%. Parrot, an offline solution, increases the hit ratio by 16.6% over LRU. In this evaluation, We do not compare with them directly. The target applications also differ. We use compiler benchmarks, while Hawkeye, Glider, and Parrot used SPEC CPU benchmarks. However, we note that the best achievable performance of these techniques is below OPT, often significantly.

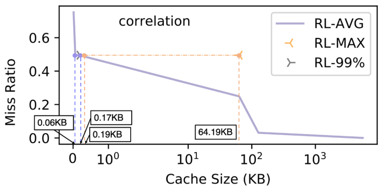


Fig. 5. The maximal cache consumption is close to the average for 99.85% of accesses, shown by the small gap between RL-99% and RL-AVG. By reducing the lease to 0 for 0.4% of accesses, the maximal consumption is reduced from 64.19 KB to 0.17 KB.

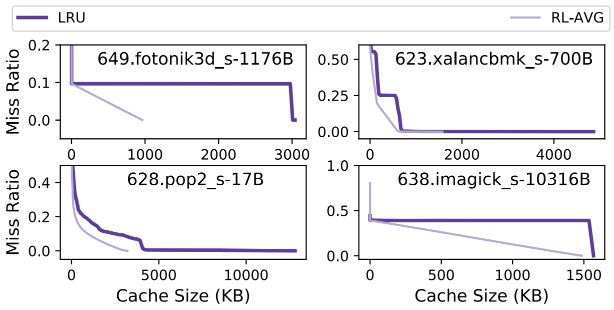


Fig. 6. Miss ratio curves for CARL and LRU on 10M-instruction fragments.

CARL outperforming OPT. A careful reader may notice that, in some cache sizes, CARL could exceeds OPT in cache utilization. This is because CARL is a variable-size caching policy while OPT is a fixed size cache policy.

Optimal caching is traditionally solved as a run-time problem and requires the memory access trace. The evaluation results show two new ways to achieve optimal caching. First, the caching problem may be solved at the program level using program code instead of a trace. At trace level, OPT ranks the data by the dynamic order of their accesses, which is costly to measure and use. At program level, CARL solves the same problem using RI distributions and reference leases, which seem comparable to OPT dynamic ranking.

Second, in these tests, compiler control may be sufficient to manage the cache optimally. Trace-level OPT has to control data eviction at each access. In loop-based code, the program structure contains information on data usage and provides a means of control. For reference leasing, accesses from the same reference are more likely to not only have the same reuse pattern but also benefit from the same control. As a result, CARL needs only a small number of leases, on average 13 as shown in Table 1. It achieves comparable performance to OPT using on average just 13 numbers.

5.3 Average vs. Maximal Cache Consumption

The cache consumption varies in CARL. The variation is shown in Figure 4 by the maximal consumption, RL-MAX. At high miss ratios, RL-MAX is similar to RL-AVG. For low miss ratios, the maximal can be much greater than the average. In these cases, some references are given a long lease, causing the cache use to jump temporarily.

The gap between the average and maximal cache consumption may be large. This is the case in correlation, which we show in more detail in Figure 5.

In correlation, at the miss ratio around 50%, the average cache use is 0.19 KB, but the maximum is 64.19 KB. There is a large gap. The correlation code is shown in List 1. The long lease is 3075 to ref_3, which captures the spatial reuse for ref_3 across outer-loop iterations. It temporarily increases the maximal cache size by 1,024 cache blocks but only for this particular loop nest.

In gramschmidt, starting at a miss ratio of around 57%, the average and maximal cache uses are 0.09 KB and 0.19 KB, respectively. For the next three consecutive lease assignments, the average

```
for (col = 0; col < 1024; col++) {
  ... // ref1 store
  for (row = 0; row < 1024; row++) {
    //  ref4 = ref2 + ref3
    mean[col] = mean[col] + data[row * 1024 + col];
  }
  ... // two more references
}
```

Listing 1. Code snippet of `correlation` which calculates the mean of columns of a matrix.

use increases to 0.13 KB, 0.19 KB, and 0.23 KB, while the maximum increases to 64 KB, 85 KB, and 128 KB. The miss ratio is lower by 0.03%, 0.06%, and 0.08%. The compiler may assign 0 lease to the 0.3% of memory accesses and reduce the maximal cache use from 128 KB to 0.23 KB.

Prechtl et al. [54] studied a more realistic setup. Their CLAM system has implemented a split single-level data cache of 8 KB CycloneV-GT FPGA. In fewer than half of the tests, CARL exceeds the cache capacity. Data are evicted randomly when the leased space exceeds 8 KB. To limit the over allocation, they developed **Phase-based Reference Leases** (**PRL**), which divides a program into phases and runs CARL within each phase. They found that PRL is robust and consistently performing the best over CARL and two other policies including Static **Re-reference Interval Prediction** (**RRIP**) [41].

For fixed-size cache, PRL would prevent the cache overflow problems such as the one in *correlation* (the short loop nest is a phase, and CARL would allocate within the cache capacity). We leave as future study the effect of PRL on maximal cache use but note that at each phase, PRL is identical to CARL, so the theorems hold for each phase of PRL.

5.4 Applicability and Performance on SPEC CPU 2017

Doerfert et al. [25] examined the applicability of the Polly tool for polyhedral compilation in the SPEC 2,006 benchmarks. Of 1,862 code regions (single-entry, single-exit with at least one loop) surveyed across nine benchmarks in SPEC, Polly could compile 275 regions (14.8%). In contrast, SPS [19] can handle regions that Polly could not due to non-affine expression (1,230 regions), non-affine loop bounds (840), non-canonical induction variables (384 loops that do not start at 0 and are not incremented by 1), overflow issues from unsigned comparisons (199), presence of function calls (532), and complex CFG (253, due to, e.g., switch). SPS cannot deal with regions that contain aliasing (1,093), but clearly many more loops are compile-time enumerable than those subject to the restrictions of the polyhedral model.

We use DPC3 SPEC CPU 2017 traces[4] to test CARL on full applications. Each trace contains 10 million instructions. For 20 SPEC benchmarks, the average number of static references is 1,801 per 10M-instruction trace, the average number of different RIs per reference is 215. For all cache sizes measured for each benchmark, CARL reduces miss ratio by 0%–1% for 68%, 1%–5% for 13%, 5%–10% for 8%, 10%–20% for 6%, 20%–50% for 3%, and 50% or more for 1.4% of cache sizes (absolute reduction).

Figure 6 shows the comparison for the subset of traces where CARL makes the largest improvements over `RL-AVG` and `LRU`. Though for most of the benchmarks CARL reduces less than 5% of the misses, the results confirm two benefits. First, CARL is at least as good as LRU and does not degrade performance. Second, in a significant number of cases, CARL makes large improvements

[4]https://dpc3.compas.cs.stonybrook.edu.

for complex programs with a large number of references and more complex RI distributions. Note that these results assume perfect program information and are not based on compiler analysis.

6 RELATED WORK

Lease Cache. The lease cache is recently introduced by Li et al. [44]. They developed **Optimal Steadystate Lease** (**OSL**), which analyzed a trace and assigned a lease for each page based on the RI histogram for its accesses. OSL is a trace-level solution and needs to examine all pages and their accesses to assign a lease for each page. Neither the information nor the lease control is feasible at program level. The theoretical results of CARL, PPUC monotonicity, midpoint lemma, and convexity apply also to OSL. It is a simple corollary that OSL is optimal, which Li et al. [44] conjectured but did not prove.

Li et al. [44] showed that OSL performs similarly or better than OPT for a storage workload. However, the space overhead is significant: the traces contain between 162 thousand and 230 million pages, and OSL used an RI histogram per page [44, Table 1]. Among 30 benchmarks in PolyBench, CARL uses 6–34 RI distributions (Table 1), while OSL uses 256–4,194,048, which is many orders of magnitude less. Besides, the number of OSL leases grows with the data size, but the number of CARL leases does not. With significantly lower space requirement, CARL still obtains a similar comparison with OPT (Figure 4) as OSL did.

In a workshop article, Chen et al. [18] proposed **Compiler Lease of Accelerator Memory** (**CLAM**), which was the first study of reference leases. Through simulation, they measured the effect of reference leasing in element and cache-block granularity and compared it with that of LRU. The results show that CARL is more effective than LRU in evicting cache blocks with poor utilization. In other words, compiler knowledge allows the cache to better retain data blocks with good spatial reuse.

Prechtl et al. [54] developed a hardware prototype of the lease cache on FPGA and made CLAM a practical design for using leases in a fix-size, single-level, and not shared hardware cache. They tested and compared techniques including CARL and PRL (discussed in Section 5.3).

Uniform Lease Cache. The simplest possible lease assignment policy is ***uniform lease* (UL)** *policy*, where each access receives the same lease. We call the cache managed by the UL policy the *UL cache*. The UL assignment can be considered as the special case of CARL in which all accesses are in the same set. Chen et al. [17] concluded that the performance of UL cache and LRU cache are similar in most cases. As such, lease caching can be said to be performance safe, as LRU performance can be achieved with minimal program information.

Reference Locality Modeling and Optimization. It is common for loop-nest optimization to classify references by the temporal and spatial reuse [2, 3, 50]. Reuse distance, i.e., LRU stack distance, allows such locality be quantified. Marin and Mellor-Crummey [48] analyzed the reuse distance histogram for each reference and used it for performance modeling across program inputs and machine types. Other techniques also used this notion of locality. Fang et al. [29] used it to find critical memory loads. Beyls and D'Hollander [11] used it to compile a program with cache hints and to build a program tuning tool SLO to identify the cause of poor locality and give suggestions for restructuring the code.

CARL is based on reference locality but differs in two aspects. First, the reference locality is defined by reuse intervals instead of reuse distances. Second, it is used to directly control the cache. Reuse intervals have been used to model cache performance in working-set theory from its inception [23] and a number of later techniques [27, 39, 56, 68]. Reuse intervals are efficient to measure and as a result frequently used in profiling and run-time analysis. Shen et al. [56] developed statistical conversion between the reuse interval and reuse distance, which is then used by the locality tuning tool SLO [12] and recently a near real-time analysis tool based on hardware data watchpoints [64]. The relation between the reuse interval and reuse distance is formalized in the relational theory of locality [69, 73].

Compile-time Modeling of Cache Performance. Compiler techniques has long been used to improve cache performance by improving data reuses [2, 3, 5, 15, 31, 67]. To reduce the run-time overhead, a cache stores data in block granularity, which complicates analysis but can be modeled by a compiler [16, 30, 62, 70]. Two recent papers [6, 35] have given symbolic and accurate cache models. There are precise models of cache behavior.

The lease cache permits direct cache control. Such control simplifies cache modeling. The theory in this article shows two such benefits. The first is synergy between analysis and control. The CARL algorithm for cache allocation is at the same time the model for cache performance, as it selects an allocation based on its effect on performance (measured by PPUC). The second is reduction in complexity. For analysis, the most complex problem is to model the order of data access. In CARL, the RIs are used for their distribution without any ordering information, yet the cache utilization matches and exceeds OPT, which requires knowing the full trace. More powerful static techniques may improve the cache utilization even beyond CARL. Furthermore, an open problem is whether the variation of cache size can be analyzed or more importantly, be bounded. More powerful compiler analysis may address this problem.

Collaborative cache. Hardware–software collaborative caching was first used by Wang et al. [65] and Beyls and D'Hollander [10, 11]. Gu et al. [32] proved that cache hints can obtain the performance of optimal cache. Furthermore, the collaborative caching policy, called the LRU-MRU cache, is a stack algorithm and observes the inclusion property [33]. They gave the algorithm to compute the LRU-MRU stack distance. Gu and Ding [34] then studied priority hints beyond single-bit hints and showed non-uniform inclusion property of this general policy and their one-pass evaluation. Brock et al. [13] developed a practical solution for collaborative caching on loop-based code called **Program Assisted Cache Management** (**Pacman**).

CARL differs in two ways. First, CARL targets variable-size cache. Second, cache hints are suggestions, while leases mean direct control using program information. The baseline solution also differs: it is to give no hint in the collaborative cache and the uniform lease in the lease cache.

Register and Scratchpad Memory Allocation. Registers may be viewed as a compiler managed cache. In the bibliographical notes after the chapter on register allocation, Cooper and Torczon [21, Chap. 13] pointed out that the Belady algorithm published in 1966 [8] was independently invented by Sheldon Best working on the first Fortran compiler [4], and "Best's algorithm has been rediscovered and reused in many contexts over the years" [22, 45]. **Scratchpad memory** (**SPM**) allocation is solved by extending the techniques of allocating scalars to registers to allocating arrays in SPM.

Udayakumaran et al. [61] showed that dynamic allocation using their compile-time heuristics could significantly outperform optimal static allocation. Taking a different approach, Li et al. [43] used the static solution of graph coloring but enabled dynamic allocation through live-range splitting.

Past compiler solutions did not try to claim OPT optimality. Udayakumaran et al. [61] showed that the performance of their compiler solution called the **Data-Program Relationship Graph** (**DPRG**) was comparable to a direct-map cache. Li et al. [43] improved DPRG by as high as 13% (in speedup) but did not compare the results with OPT. In this study, we show a large gap between LRU and OPT and that CARL can potentially close this gap in almost all our tests.

Cache System Design. The literature of cache system design is long and still rapidly growing. Duong et al. developed the ***Protecting Distance-based Policy (PDP)*** [26]. PDP "prevents replacing a cache line until a certain number of accesses to its cache set." It is the same as the uniform lease, i.e., the baseline solution discussed previously. PDP is a run-time solution, while UL is static. It is a widely recognized problem that LRU does not perform well for the streaming pattern, e.g., when an array larger than the cache size is repeatedly traversed. Many techniques have been developed, including page coloring, cache replacement techniques such as SRRIP [41], cache bypassing, and cache hints. Two recent techniques are Talus [7] and SLIDE [63]. Talus partitions the access stream to have the effect of dividing the working set, and SLIDE, with scaled-down simulation, achieves this effect for stack or non-stack cache policies. CARL has a similar effect, but it uses program rather than cache control.

Many past designs target fixed-size cache with the goal of closing the gap between LRU and OPT. Hawkeye, by actually simulating OPT in real-time, has moved close to OPT, achieving half of the improvement of OPT over LRU [40]. Recent designs called Glider [57] and Parrot [46] improve over Hawkeye with the help of machine-learning techniques, as also in [55]. Glider is direct improvement over Hawkeye and uses SVM. Parrot is an offline solution and uses the newest sequence learning technique called Self Attention. On average, Glider increases the hit ratio by 12.8% over LRU, while OPT increases it by 18.6%. Parrot, an offline solution, increases the hit ratio by 16.6% over LRU. In this evaluation, We do not compare with them directly. The target applications also differ. We use compiler benchmarks, while Hawkeye, Glider, and Parrot used SPEC CPU benchmarks.

Cache systems are automatic, while CARL is a programming solution. All automatic solutions require empirically determined thresholds, while CARL uses compiler analysis and provides theoretical guarantees. Optimal caching is traditionally solved as a run-time problem and requires the memory access trace. CARL shows three new findings. First, the caching problem may be solved at the program level using program code instead of a trace. Second, in these tests, compiler control may be sufficient to manage the cache optimally. Finally, using variable size caches, we may exceed OPT performance.

7 SUMMARY

As memory hierarchies become larger and more complex, fully automatic control, i.e., caching, can be too restrictive since it does not utilize program knowledge. The lease cache hardware and reference leasing compiler provide programmable cache and its program control.

This article has presented a theory and a potential study and shown that the cache can be programmed automatically and optimally. We prove the PPUC monotonicity, the midpoint lemma, and CARL optimality. From this optimality, we prove the miss curve convexity and sub-partitioning monotonicity, which is useful for optimizing a shared cache and for simplifying the compiler design, in either hardware or software cache. When evaluated on loop-based kernel code, we show that just 6–34 reference leases can achieve the same or better cache utilization than OPT, which is not yet attainable with automatic solutions (including recent techniques utilizing large-scale machine learning) but now within the potential of cache programming for all tested programs and most cache sizes.

APPENDICES

A DIRECT PROOF OF SUB-PARTITIONING MONOTONICITY

Theorem 4.1 is proved based on the optimality theorem. Here, we show a direct proof without assuming optimality. First, we prove a lemma that establishes that a lease to a group of references must lie between leases to the individual references that make up the group. Let two references be ref_1, ref_2,[5] their RI distributions D_{ref_1}, D_{ref_2} and access ratios AR_1, AR_2. Let ref_{12} represent the combined effect of the two. We have the combined access ratio and RI distribution:

$$AR_{12} = AR_1 + AR_2, \tag{5}$$

$$D_{ref_{12}}[i] = \frac{D_{ref_1}[i] * AR_1 + D_{ref_2}[i] * AR_2}{AR_{12}} \qquad i = 1 \ldots N. \tag{6}$$

Assume a program exists with just these two references. We obtain individual leases by running CARL using D_{ref_1}, D_{ref_2}. At each step t, the individual leases are l_1^t, l_2^t. At the first step, we have $l_1^0 = l_2^0 = 0$. We obtain joint leases by running CARL using $D_{ref_{12}}$. At each step, the group lease is l_{12}^t.

Lemma A.1. *For two references ref_1, ref_2 with any RI distributions, the lease assigned to the reference group composed of ref_1, ref_2, known as l_{12}, always lies within the range of the lease assigned when considering the references separately. $l_1 \leq l_{12} \leq l_2$.*

Proof. When assigning leases to each individual reference, compared with that for reference groups, the only difference is the granularity of the access we consider. The cache space for the reference access groups should be equal to the sum of the cache space costs for each reference, written as $Cost(D_{ref_{12}}, l_{12}) = Cost(D_{ref_1}, l_1) + Cost(D_{ref_2}, l_2)$. We use Equation (2) to compute the space cost with respect to the reuse-interval distribution D and the assigned lease l.

$$\begin{aligned} LEFT &= \left(\sum_{i=0}^{l_{12}} i * D_{ref_{12}}[i] + \sum_{i=l_{12}+1}^{N} l_{12} * D_{ref_{12}}[i]\right) * AR_{12} \\ &= \left(\sum_{i=0}^{l_{12}} i * D_{ref_1}[i] + \sum_{i=l_{12}+1}^{N} l_{12} * D_{ref_1}[i]\right) * AR_1 + \left(\sum_{i=0}^{l_{12}} i * D_{ref_2}[i] + \sum_{i=l_{12}+1}^{N} l_{12} * D_{ref_2}[i]\right) * AR_2 \end{aligned} \tag{7}$$

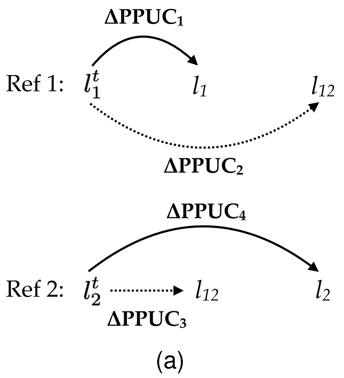
$$RIGHT = \left(\sum_{i=0}^{l_1} i * D_{ref_1}[i] + \sum_{i=l_1+1}^{N} l_1 * D_{ref_1}[i]\right) * AR_1 + \left(\sum_{i=0}^{l_2} i * D_{ref_2}[i] + \sum_{i=l_2+1}^{N} l_2 * D_{ref_2}[i]\right) * AR_2. \tag{8}$$

As $LEFT - RIGHT = 0$, we have:

$$\begin{aligned} &\left(\sum_{i=0}^{l_1} i * D_{ref_1}[i] - \sum_{i=0}^{l_{12}} i * D_{ref_1}[i] + \sum_{i=l_1+1}^{N} l_1 * D_{ref_1}[i] - \sum_{i=l_{12}+1}^{N} l_{12} * D_{ref_1}[i]\right) * AR_1 \\ &\quad + \left(\sum_{i=0}^{l_2} i * D_{ref_2}[i] - \sum_{i=0}^{l_{12}} i * D_{ref_2}[i] + \sum_{i=l_2+1}^{N} l_2 * D_{ref_2}[i] - \sum_{i=l_{12}+1}^{N} l_{12} * D_{ref_2}[i]\right) * AR_2 = 0. \end{aligned} \tag{9}$$

Equation (9) means that the cache space difference ($\Delta Cost$) when switching the lease from l_1 to l_{12} is equal to that when switching the lease from l_{12} to l_2. If we assume that $l_1 > l_{12}$, then Equation (9)

[5]We always mark the reference with smaller lease as r_1. Then we have $l_1 \leq l_2$.



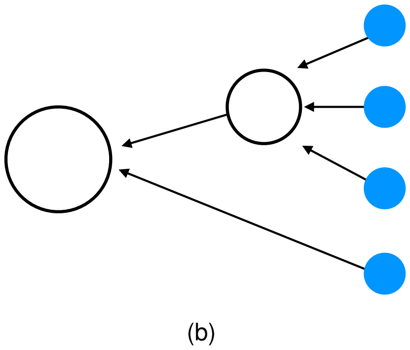


Fig. 7. (a) At time t, l_1^t, and l_2^t are the temporary leases for ref_1, ref_2. When treating ref_1, ref_2 as a group, l_{12} will be assigned to both of them, while when considering individually, l_1 and l_2 will be their corresponding leases. All arrows with ΔPPUCs indicate the performance gain brought by lease switching. (b) The DAG representation of merging references to reference groups. Every vertex represents either a single reference (blue nodes) or a reference group (white nodes). The edge points to the reference group it belongs to.

can be rewritten as

$$\left(\sum_{i=l_{12}+1}^{l_1}(i-l_{12})*D_{ref_1}[i]+\sum_{i=l_1+1}^{N}(l_1-l_{12})*D_{ref_1}[i]\right)*AR_1$$
$$+\left(\sum_{i=l_{12}+1}^{l_2}(i-l_{12})*D_{ref_2}[i]+\sum_{i=l_2+1}^{N}(l_2-l_{12})*D_{ref_2}[i]\right)*AR_2=0.$$

However, each term in this equation is greater than 0. So, the assumption $l_1 > l_{12}$ does not hold and we have $l_{12} \geq l_1$. Similarly, if we assume that $l_{12} > l_2$, all terms in this equation are smaller than 0. So, l_{12} always lies within the range (l_1, l_2) with respective to any RI distributions. □

Next we will prove that the per-reference lease assignment can achieve higher hit ratio than reference-group lease assignment. We call it as the *group monotonicity*.

Theorem A.2 (Sub-partition Monotonicity). *For two references ref_1, ref_2 with any RI distributions and access ratios, let l_1, l_2 be the leases selected individually and l_{12} the lease selected for them as group. Then l_{12} cannot give a lower miss ratio for the same or less cache allocation than l_1, l_2, that is, for all $l_{12}, l_1, andl_2$ such that $Cost(D_{ref_1}, l_1) + Cost(D_{ref_2}, l_2) = Cost(D_{ref_{12}}, l_{12})$, we must have*

$$\mathcal{H}(D_{ref_1}, l_1) + \mathcal{H}(D_{ref_2}, l_2) \geq \mathcal{H}(D_{ref_{12}}, l_{12}),$$

where $\mathcal{H}$ represents the hit ratio.

Proof. As indicated in Figure 7(a), at time t, leases l_1^t and l_2^t were assigned to references ref_1, ref_2, respectively. Equation (3) indicates that the change of PPUC could be derived from the ratio between the increment of hit ratio and that of space, written as $\Delta\text{PPUC} = \frac{\Delta\mathcal{H}}{\Delta Cost}$. When assigning l_{12} to both references, the hit ratio will become $\mathcal{H}_{init} + (\Delta\text{PPUC}_2 + \Delta\text{PPUC}_3) * \Delta Cost_{grp}$ and that will become $\mathcal{H}_{init} + (\Delta\text{PPUC}_1 + \Delta\text{PPUC}_4) * \Delta Cost_{ref}$ when assigning l_1, l_2 separately to ref_1, ref_2, where $\mathcal{H}_{init}$ is the hit ratio achieved at time t, $\Delta Cost_{grp}$ and $\Delta Cost_{ref}$ represents the increment of space when assigning leases to reference groups and single references, respectively. In Lemma A.1, we have already proved that $\Delta Cost_{grp} = \Delta Cost_{ref}$. In CARL, each time a old lease was replaced if and only if the increment PPUC for the new lease is greater than that for the old lease ($\Delta\text{PPUC}_{new} \geq \Delta\text{PPUC}_{old}$).

Hence, we have $\Delta PPUC_4 \geq \Delta PPUC_3$ and $\Delta PPUC_1 \geq \Delta PPUC_2$. For the same cache size, this inequality of PPUC implies the inequality of hit ratios, i.e., $\mathcal{H}(D_{ref_1}, l_1) + \mathcal{H}(D_{ref_2}, l_2) \geq \mathcal{H}(D_{ref_{12}}, l_{12})$. □

In the following corollary, we extend it to all reference groups, since it shows that grouping references cannot improve cache performance, and per reference leases give the best solution.

Corollary A.3. *For any reference groups with any RI distributions, the lease assigned based on smaller reference groups will always outperform the lease assigned based on larger reference groups.*

Proof. Figure 7(b) shows one example of representing the reference grouping into DAG format. Theorem A.2 could be applied on every edge. Given that, every destination node will incur higher miss ratio than its source nodes. □

B CARL RESULTS BASED ON SPS SAMPLED RI DISTRIBUTIONS

Figure 8 shows the RL-AVG results with SPS sampled RI distributions (5% sampling rate). We can see that for 24 of 30 benchmarks, SPS-5% overlaps with RL-AVG. For cholesky, lu and ludcmp, there is a small gap between SPS-5% and RL-AVG for some of the cache sizes because the reuse intervals are scattered for some of the references in the benchmarks. For nussinov, floyd_warshall, which contain branches, the RI distribution collected is not that reliable as current SPS implementation ignores branches. One may notice that this figure does not show the SPS-5% miss ratio curves for symm and syrk. This is because their miss ratios are higher than the upper bound. For example, in syrk, the miss ratio for a 128 KB cache is 0.76, much higher than bound 0.07

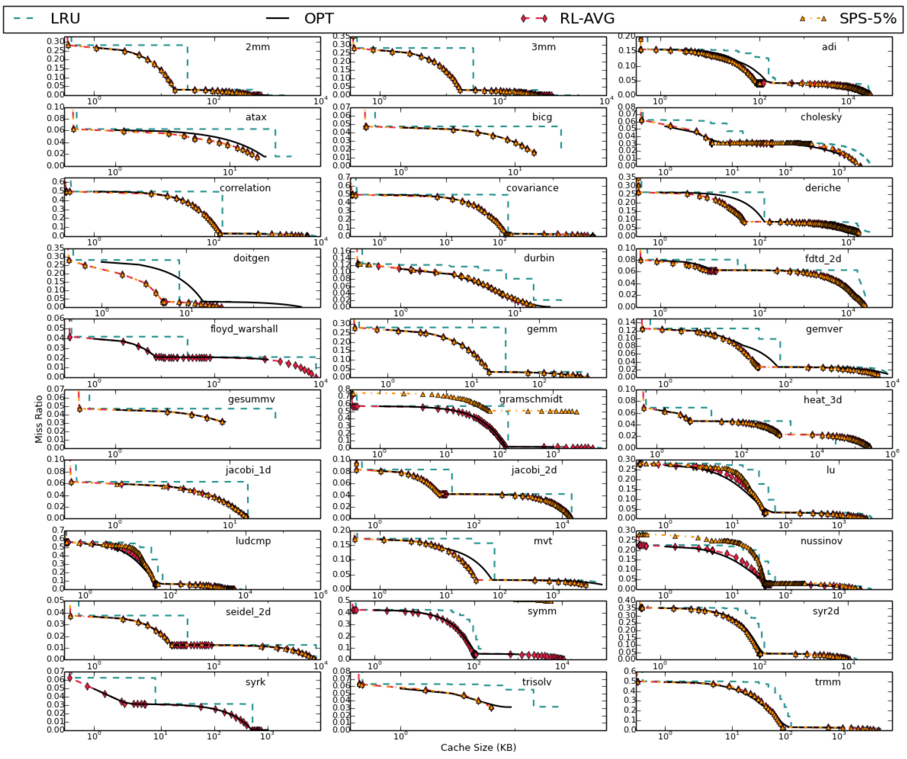


Fig. 8. Miss ratio curves for LRU, RL-AVG, OPT and RL-AVG with sampled RI distributions (SPS-5%).

Table 2. The Number of References and the Total Time (SPS in 5% Sampling + CARL Lease Assignment) Measured in Milliseconds ($10^{-3}s$)

Tests I	#references	Time	Tests II	#references	Time
2 mm	11	38	mvt	8	214
3 mm	15	30	seidel_2d	10	24
adi	34	398	syrk	6	21
atax	10	340	trisolv	9	73
bicg	10	218	cholesky	13	914
deriche	20	288	covariance	16	5, 935
doitgen	7	9	correlation	34	4, 063
durbin	9	22	floyd_warshall	7	17, 513
fdtd_2d	16	242	gramschmidt	15	60, 132
gemm	6	23	lu	11	1, 881
gemver	17	189	ludcmp	18	1, 573
gesummv	13	221	nussinov	18	123
heat_3d	22	5189	symm	10	4, 585
jacobi_1d	8	0.3	syr2d	8	5, 894
jacob_2d	12	258	trmm	6	25, 236
Average #references		13	Average compilation time		4,522

C CARL OVERHEAD

Table 3 shows the overhead of RI Distribution collection process collected by SPS under 5% sampling rate with CARL lease assignment process, unit in milliseconds. On average, among all 30 benchmarks, CARL adds just 4.36% of overhead of the entire process of compiler cache allocation (SPS + CARL).

Table 3. The Number of References and the Total Time (Profiling + CARL Lease Assignment) Measured in Seconds

Tests I	CARL	Toal	Tests II	CARL	Total
2 mm	.09	74.7	mvt	.03	4.60
3 mm	.01	120.2	seidel_2d	.08	67.1
adi	33.0	163.8	syrk	.05	18.4
atax	.01	4.11	trisolv	.01	.53
bicg	.01	4.43	cholesky	.92	254.
deriche	35.3	59.5	covariance	6.96	1452.
doitgen	.002	35.4	correlation	20.8	1464.7
durbin	.09	0.84	floyd_warshall	.001	2398.6
fdtd_2d	23.7	125.7	gramschmidt	.04	3622.1
gemm	.001	31.2	lu	88.7	947.8
gemver	3.86	12.2	ludcmp	90.9	638.7
gesummv	.02	3.49	nussinov	901.7	1301.1
heat_3d	.06	2771.8	symm	11.2	2500.8
jacobi_1d	.000	.01	syr2d	6.25	2596.24
jacob_2d	.10	78.2	trmm	2.97	1134.1
Average CARL time		40.9	Average total time		728.6

ACKNOWLEDGMENTS

Pengcheng Li and Colin Pronovost contributed to the first formulation of CARL optimality and its proof. Mingyang Jiao collected the OPT results shown in Figure 4. Hannah Simons helped with the illustration. We also thank many people who have given feedback and suggestions including Donovan Snyder, Noah Bertram, Katherine Seeman, Joshua Radin, Aaron Gindi, the rest of the University of Rochester systems research group, and the anonymous reviewers of this journal.

REFERENCES

[1] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: Benchmarking python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea). Association for Computing Machinery, New York, NY, 59–70. DOI: https://doi.org/10.1145/3446804.3446842

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[3] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers.

[4] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference.* 188–198.

[5] Utpal Banerjee. 1997. *Dependence Analysis.* Kluwer. I–XVII, 1–214 pages.

[6] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *PACMPL* 2, POPL (2018), 32:1–32:26. DOI: https://doi.org/10.1145/3158120

[7] 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. DOI: https://doi.org/10.1109/HPCA.2015.7056022

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

[9] Laszlo A. Belady, Robert A. Nelson, and Gerald S. Shedler. 1969. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM* 12, 6 (1969), 349–353.

[10] 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.

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

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

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

[14] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal cache partition-sharing. In *Proceedings of the International Conference on Parallel Processing.*

[15] Calin Cascaval and David A. Padua. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing.* 150–159.

[16] S. Chatterjee, E. Parker, P. J. Hanlon, and A. R. Lebeck. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* Snowbird, UT.

[17] 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* (Virtual, Canada). Association for Computing Machinery, New York, NY, 15–27. DOI: https://doi.org/10.1145/3459898.3463908

[18] Dong Chen, Chen Ding, and Dorin Patru. 2019. CLAM: Compiler leasing of accelerator memory. In *Languages and Compilers for Parallel Computing, LCPC 2019, Atlanta, GA, October 22-24, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11998)*, Santosh Pande and Vivek Sarkar (Eds.), Springer, 89–97.

[19] 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. DOI: https://doi.org/10.1145/3192366.3192402

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

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

[22] Jack W. Davidson and Christopher W. Fraser. 1984. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems* 6, 4 (1984), 505–526. DOI: https://doi.org/10.1145/1780.1783

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

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

[25] Johannes Doerfert, Clemens Hammacher, Kevin Streit, and Sebastian Hack. 2013. Spolly: Speculative optimizations in the polyhedral model. In *Proc. 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT'13)*, Armin Größlinger and Louis-Noël Pouchet.

[26] 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. DOI: https://doi.org/10.1109/MICRO.2012.43

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

[28] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. 2015. On characterizing the data access complexity of programs. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 567–580. DOI: https://doi.org/10.1145/2676726.2677010

[29] Changpeng Fang, Steve Carr, Soner Önder, and Zhenlin Wang. 2005. Instruction based memory distance analysis and its application. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques.* 27–37.

[30] S. Ghosh, M. Martonosi, and S. Malik. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems* 21, 4 (1999), 703–746.

[31] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. POLLY - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010. DOI: https://doi.org/10.1142/S0129626412500107

[32] 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.

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

[34] Xiaoming Gu and Chen Ding. 2012. A generalized theory of collaborative caching. In *Proceedings of the International Symposium on Memory Management.* 109–120.

[35] Tobias Gysi, Tobias Grosser, Laurin Brandner, and Torsten Hoefler. 2019. A fast analytical model of fully associative caches. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 816–829. DOI: https://doi.org/10.1145/3314221.3314606

[36] E. G. Hallnor and S. K. Reinhardt. 2000. A fully associative software-managed cache design. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201).* 107–116. DOI: https://doi.org/10.1145/339647.339660

[37] J. Hong and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the ACM Conference on Theory of Computing.* Milwaukee, WI.

[38] 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. DOI: https://doi.org/10.1109/TPDS.2016.2611572

[39] 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. DOI: https://doi.org/10.1145/3185751

[40] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the International Symposium on Computer Architecture.* 78–89. DOI: https://doi.org/10.1109/ISCA.2016.17

[41] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture.* ACM, 60–71. DOI: https://doi.org/10.1145/1815961.1815971

[42] Rahman Lavaee. 2016. The hardness of data packing. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 232–242. DOI: https://doi.org/10.1145/2837614.2837669

[43] Lian Li, Hui Feng, and Jingling Xue. 2009. Compiler-directed scratchpad memory management via graph coloring. *TACO* 6, 3 (2009), 9:1–9:17. DOI: https://doi.org/10.1145/1582710.1582711

[44] 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. DOI: https://doi.org/10.1145/3297858.3304067

[45] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. 1999. Evaluation of algorithms for local register allocation. In *Proceedings of the International Conference on Compiler Construction*. 137–152. DOI: https://doi.org/10.1007/978-3-540-49051-7_10

[46] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning, (ICML'20), 13-18 July 2020, Virtual Event, Vol. 119*. PMLR, 6237–6247. http://proceedings.mlr.press/v119/liu20f.html. An Imitation Learning Approach for Cache Replacement. arXiv:2006.16239. Retrieved from https://arxiv.org/abs/2006.16239.

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

[48] G. Marin and J. Mellor-Crummey. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 2–13.

[49] 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.

[50] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems* 18, 4 (1996), 424–453.

[51] Pierre Michaud. 2016. Some mathematical facts about optimal cache replacement. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 50:1–50:19. DOI: https://doi.org/10.1145/3017992

[52] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, 808–822. DOI: https://doi.org/10.1145/3385412.3385989

[53] E. Petrank and D. Rawitz. 2002. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[54] 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*.

[55] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a cost-effective cache replacement policy using machine learning. In *Proceedings of the 2021 IEEE International Symposium on High Performance Computer Architecture*. IEEE, 297–308.

[56] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–61.

[57] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, 413–425. DOI: https://doi.org/10.1145/3352460.3358319

[58] Harold S. Stone, John Turek, and Joel L. Wolf. 1992. Optimal partitioning of cache memory. *IEEE Transactions on Computers* 41, 9 (1992), 1054–1068. DOI: https://doi.org/10.1109/12.165388

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

[60] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–12.

[61] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computer Systems* 5, 2 (2006), 472–511. DOI: https://doi.org/10.1145/1151074.1151085

[62] Xavier Vera, Nerina Bermudo, Josep Llosa, and Antonio González. 2004. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Transactions on Programming Languages and Systems* 26, 2 (2004), 263–300. DOI: https://doi.org/10.1145/973097.973099

[63] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *Proceedings of USENIX Annual Technical Conference*. 487–498. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger.

[64] Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight reuse-distance measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 440–453. DOI: https://doi.org/10.1109/HPCA.2019.00056

[65] 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.

[66] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. 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.

[67] M. J. Wolfe. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA.

[68] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time modeling of program working set in shared cache. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 350–360.

[69] 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.

[70] Jingling Xue and Xavier Vera. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput.* 53, 5 (2004), 547–566. DOI: https://doi.org/10.1109/TC.2004.1275296

[71] 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*. USENIX Association, 191–208. Retrieved from https://www.usenix.org/conference/osdi20/presentation/yang.

[72] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. 2017. Rochester elastic cache utility (RECU): Unequal cache sharing is good economics. *International Journal of Parallel Programming* 45, 1 (2017), 30–44. DOI: https://doi.org/10.1007/s10766-015-0384-3

[73] 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. DOI: https://doi.org/10.1145/3341109

[74] 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.

Received August 2021; revised November 2021; accepted November 2021