Data Access Complexity: Monotonicity and Proportionality

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

Yifan Zhu
University of Rochester
Rochester, NY, USA
yifanzhu@rochester.edu

ABSTRACT

Locality is an abstraction of cache performance, while locality complexity is its asymptotic behavior as program inputs and cache sizes vary. This paper presents two properties of locality complexity. First, Monotonicity is a necessary condition when a larger input size must lead to a reduction in locality. Second, Proportionality states that with cache optimization, its benefit scale at least linearly with its size. The properties are formally proved: Monotonicity holds for fully associative LRU caches and working-set caches, while Proportionality applies to all programs and inputs.

1 INTRODUCTION

In computing, locality refers to the phenomenon where a program can access its data in local memory most of the time, even if the program's data is too large to fit in local memory [13]. Locality is essential for performance due to the memory wall [36]. The bandwidth constraint is a fundamental bottleneck that limits the speed of data transfer between local and remote memory.

Time complexity measures the "processor work." In comparison, locality measures the "memory work," which is the cost of transferring data from remote memory to local memory. We call it *data access complexity*. Although program operations count directly into the processor work, memory operations may or may not incur memory work. It depends on the effect of caching on local memory. Higher locality is synonymous with lower data access complexity. The locality depends on the algorithm, the size of the problem, and the size and management of the cache.

When measuring time complexity, the unit is an operation. In data access complexity, we consider a simple machine model with a processor and the main memory on two computer chips. The processor may be multicore or a GPU. To exploit locality, the processor chip contains a cache, which is local memory. In locality or data access complexity, the unit is a data transfer. Time complexity serves as an upper bound on data access complexity because a data transfer can only be triggered by a memory operation (in the case of demand caching.[1]). This upper bound is weak because not every memory operation incurs a data transfer.

Monotonicity. A fundamental feature of time and space complexity is monotonicity. A large problem requires more time and space than a small problem. Typically, higher time and space complexity also means worse locality. We say that *monotonicity* holds if locality decreases as the size of the problem increases. However, locality is not always monotonic. For example, the locality of the n-body simulation depends on the distribution of objects in a simulated system, and there may be better locality in a system with a larger number of objects (with greater time and space complexity) than in a small system. In this paper, we formalize the conditions of locality monotonicity.

Proportionality. The best-case locality is straightforward. If a single data item is accessed and cached, only one data transfer is needed from remote to local memory. The worst locality occurs when there is no caching, resulting in every memory access requiring a remote data fetch. While poor cache management can lead to the worst locality, we are considering the scenario under the best possible caching. We demonstrate that the memory access complexity can always be reduced in proportion to the cache size, and this performance is always achievable.

These complexity results have the following practical implications:

- We have formal conditions to check whether an algorithm and its implementation are monotone in its locality.
- We can always improve cache implementation so that the benefit is at least linearly proportional to the cache size.

Locality is an abstraction that characterizes performance beyond what happens on a single machine. Locality complexity further raises the level of abstraction to characterize *all inputs* of a program. This paper shows two properties of complexity for *all programs*. Monotonicity divides all programs into two types: monotone and non-monotone. Proportionality holds for all programs. To show abstract definitions, we use naive matrix multiplication as an example.

Locality in this paper means cache misses without prefetching. Latency tolerance can often effectively reduce the time cost of cache misses but not the energy cost. Hence, the data access complexity in this paper refers to the energy cost of moving data, not the time cost. The results of the paper are derived based on data reuses, which can be element- or block-granularity. We use element granularity in examples to make them simpler to understand. The findings, Monotonicity, Identity Equation, and Proportionality hold for any fixed data granularity.

When prefetching is used, some misses are converted into cache hits. However, prefetching can also load too much data, resulting in unnecessary memory transfers. The miss count for demand caching is the amount of memory transfer that occurs without prefetching, compared to the least amount of memory transfer when prefetching is applied.

[1] Demand caching transfers one data block at a time when the data is needed. It is an extension of the concept of demand paging in virtual memory management [11]. Prefetching (or prepaging) reduces the exposed latency but does not reduce the total amount of data transfer.

2 LOCALITY COMPLEXITY

2.1 Background

2.1.1 Terminology. By Peter J. Denning, "the principle of locality is the tendency fro programs to cluster references to subsets if address space for extended periods" [12, 14]. Complexity is an asymptotic measure of the resources that a program or algorithm requires to run. Locality complexity is the asymptotic cost of accessing the memory hierarchy by a program or algorithm.

In the data access trace, a data reuse refers to two consecutive accesses to the same data item. There are two measures of data reuse: the **reuse interval (RI)** and the **reuse distance (RD)**. RI is the length of the reuse interval in time, and RD is the number of distinct items accessed in a reuse interval, including the item being accessed. If logical time is used, RI is the difference between the time of first access and the time of reuse. For example, a sequence "abcca" has two data reuses. The RI and RD of the reuse of "a" is 4 and 3 respectively.

RI is also known as the inter-reference interval [15], inter-reference recency [24], re-reference interval [22], and reuse distance [5]. We use the term RI as it is used in recent papers [13, 38]. RD is a short name for the LRU stack distance [26] and has been in use since the early 2000s [6, 16].

2.1.2 Symbolic vs Numerical Locality. Although concepts like the miss ratio are traditionally numerical and relate to execution, locality complexity is symbolic and relates to a program or algorithm. We first distinguish between symbolic program locality and numerical trace locality.

Established in the 1960s and 1970s, the **principle of locality** forms the foundation of modern memory system design and optimization [13]. It is a property of computing in that a computing task uses a subset of data at each phase of computation, and the data subset changes at phase transitions. The locality, as defined by Denning, is the property of a memory access trace.

The **program locality** is the version of the problem in which a program is given. For locality analysis, a program means the set of all its executions, one for each input. A similar distinction exists between locality optimization and program locality optimization.

For locality, a program is considered entirely by its data or memory behavior. The data behavior of a program execution is a sequence of data references. When implemented in actual computer memory, the memory behavior is a memory address trace or, synonymously, a memory access trace. Although the theorems in this paper hold for both the data locality, i.e., element granularity, and cache locality, i.e., block granularity. we use element granularity in the examples to make them simpler to understand.

2.1.3 The Working-set Cache. In the working-set cache, each data item stays in the cache for time x every time it is accessed. The working set is the basis for managing virtual memory [13, 15]. Denning and Schwartz [15] provided a pair of equations to calculate the miss ratio and the average working-set size (WSS). The input is a distribution of RIs $P(ri)$, where $P(ri = x)$ is the portion of RIs whose value is equal to x.

$$m(x) = P(ri > x)$$
$$c(x+1) = c(x) + m(x)$$

where $P(ri)$ is the RI distribution, $m(x)$ is the miss ratio, and $c(x)$ is the average WSS. The two formulas iterate the time parameter x in increasing order. At each step, the miss ratio is the portion of RI values greater than x, and the cache size increases by the miss ratio.

In this paper, we call the c value the size of the working-set cache. The cache miss ratio is

$$mr(c) = m(c(x))$$

To differentiate, we call $m(x)$ the time-window miss ratio and $mr(c)$ the cache miss ratio. In the working-set cache, the cache size may be fractional.

2.2 LRU Cache Locality Monotonicity

For two problem sizes $\mathcal{P}, \mathcal{P}'$, we say $\mathcal{P}'$ is larger than $\mathcal{P}$ if the following two conditions hold. The first condition is called *sequence embedding*. This means that the memory access trace of $\mathcal{P}$ can be embedded in the trace of $\mathcal{P}'$. The memory access trace be a_i, $i = 1, \ldots, n$ for $\mathcal{P}$ and b_j, $j = 1, \ldots, n'$ for $\mathcal{P}'$, there exists a strictly increasing index sequence j_i such that $a_i = b_{j_i}$ for all $i = 1, \ldots, n$, where equality means that a_i, b_{j_i} access the same data item.

The second condition is that the embedding must not have intercepts between data reuses. Once embedded, each data reuse in $\mathcal{P}$ involves two accesses in $\mathcal{P}'$. An *intercept* occurs when the same data item is accessed between these accesses in $\mathcal{P}'$. If no intercept occurs between data reuses of $\mathcal{P}$, we say that the embedding is *intercept-free*.

The reuse interval and reuse distance are measures between consecutive accesses to the same data. An intercept between two data accesses will affect the reuse interval or reuse distance. To quote Reviewer 1, "while sequence embedding maintains access order, intercept-free embedding ensures that reuse is not reduced."

Theorem 2.1. *Let $mc(\_, c)$ be the number of cache misses in a fully associative LRU cache of size c. For all integers $c \geq 0$, $mc(\mathcal{P}', c) \geq mc(\mathcal{P}, c)$, if $\mathcal{P}$ is an intercept-free embedding of $\mathcal{P}'$.*

Proof. For any cache size $c \geq 0$, we prove that every miss in $\mathcal{P}$ is a miss in $\mathcal{P}'$. For each cache miss at a_i in $\mathcal{P}$, let its reuse distance be d. Since a_i is a miss, we have $d > c$. Let b_j be the embedding of a_i in $\mathcal{P}'$, and let its reuse distance be d'. Since the embedding is intercept-free, we have $d' \geq d > c$. Hence, b_j must be a miss. □

For every cache miss in $\mathcal{P}$, the corresponding access in a larger $\mathcal{P}'$ must be a miss in the cache of the same size. Monotonicity holds for LRU caching. The proof is order-sensitive and uses information about the execution sequence.

Consider naive matrix multiplication for square matrices. Let the problem size be n for naive matrix multiplication for square matrices of size $n \times n$. To prove monotonicity, we need to show that for any $n > 0$, the two conditions hold when we increase the problem size from n to $n + 1$.

- Sequence embedding: The memory access trace of $(n+1) \times (n+1)$ matrix multiplication includes all memory accesses of $n \times n$ matrix multiplication.
- Intercept freedom: There is no intercept between reuses of embedded $n \times n$ matrix accesses.

The second condition does *not* hold given the embedding in the first condition. Let the three matrices be $C = A \times B$. When computing element $C[i, j]$, it traverses the row i of A and the column j of B. At problem size n+1 when $1 \le i, j \le n$, the access to the first n elements of row i and column j are "embedded" accesses of problem size n. The non-embedded accesses are not intercepts because they access different data. At element $C[i, n+1]$, however, the non-embedded accesses traverse the row i of A, which creates intercepts.

The second requirement is effectively unworkable when considering spatial reuse (due to non-unit size cache blocks). On all computers, a cache block stores multiple data elements. A spatial reuse happens when different elements in the same block are accessed. However, all hope is not lost. We next show a less stringent condition of monotonicity.

2.3 Working-set Locality Monotonicity

Let $\mathcal{P}, \mathcal{P}'$ be two problem sizes of a program. There exists a monotonic injection function from the set of RIs of $\mathcal{P}$ to the RIs of $\mathcal{P}'$, where monotonic injection means that the value of the source RI in $\mathcal{P}$ must be no greater than that of the sink RI in $\mathcal{P}'$. In other words, for any RI in $\mathcal{P}$, there is a corresponding RI in $\mathcal{P}'$ that is the same or higher in value. We call this condition *Monotonic RI Injection*.

In the working-set cache, when a data item is loaded into the cache, it remains in the cache for a period before being evicted. Since the cache size may vary, we measure *cache consumption* by the time-space product, which is the total duration of all cache stays of all data. The unit of cache consumption is one unit of data times one unit of time.

When accessing a data item, we define *tenancy* as the time it takes for a data item from being accessed (in the cache) to either the next reuse or eviction from the cache, whichever occurs first.

Theorem 2.2. *Let $\mathcal{S}$ be the set of accesses in $\mathcal{P}$ that are cache hits. Let $c(\mathcal{P}, \mathcal{S})$ be the cache consumption for these hits in $\mathcal{P}$. Assume that $\mathcal{P}$ has a monotonic injection in $\mathcal{P}'$, and the corresponding accesses are also cache hits. Let $c(\mathcal{P}', \mathcal{S})$ be the cache consumption for these hits in $\mathcal{P}'$. We have $c(\mathcal{P}, \mathcal{S}) \le c(\mathcal{P}', \mathcal{S})$.*

Proof. Let $R_{\mathcal{S}}$ be the set of backward RIs for the accesses in (S). Each access is a cache hit, so the tenancy of each source access is equal to its RI value. The cache consumption $c(\mathcal{P}, \mathcal{S}) = \sum_{r \in R_{\mathcal{S}}} r$.

Since $\mathcal{P}$ has a monotonic injection in $\mathcal{P}'$, let $R'_{\mathcal{S}}$ be the RIs for the corresponding accesses of (S). Since these corresponding accesses are cache hits, the cache consumption $c(\mathcal{P}', \mathcal{S}) = \sum_{r \in R'_{\mathcal{S}}} r$, which is not less than $\sum_{r \in R_{\mathcal{S}}} r = c(\mathcal{P}, \mathcal{S})$. □

The monotonicity in the working-set cache means that the total cache consumption for the same set of cache hits can only increase when computing on a larger problem.

Corollary 2.3. *The locality of naive matrix multiplication for square matrices is monotonic for both element and block granularity caching.*

Proof. Let the problem size be n for naive matrix multiplication for square matrices of size $n \times n$. We show monotonic RI injection when we increase the problem size from n to $n+1$.

First, we consider the element granularity. According to Smith et al. [33], the reuse interval for matrix multiplication can be symbolically represented in the left two columns of Table 1. We see that for each RI in problem size n, the same RI exists in program size $n+1$.

Next, we consider the block granularity. The first and third columns of Table 2 show symbolic RI values and the count of each RI value. All RI value counts in the third column are monotone in n. Hence, there is a monotonic injection of RI. □

In each innermost loop iteration of the GEMM kernel, matrix multiplication computes `C[i,j]+=A[i,k]*B[k,j]`, which can be viewed as having either 3 or 4 accesses. We use 3-access matrix multiplication in element granularity, following Smith et al. [33] but 4-access in block granularity.

Table 1: RI and RD for 3-access matrix multiplication (Element Granularity)

ri	$\#ri$	rd	$\#rd$
3	$n^3 - n^2$	3	$n^3 - n^2$
$3n$	$n^3 - n^2$	$2n+1$	$n^3 - n^2$
$3n^2$	$n^3 - n^2$	$n^2 + 2n$	$n^3 - n^2$
∞	$3n^2$	∞	$3n^2$

Table 2: RI for 4-access matrix multiplication (block size b)

ri	$P(ri)$	$\#ri$
1	$\frac{1}{4}$	n^3
3	$\frac{1}{4} - \frac{1}{4bn}$	$n^3 - \frac{n^2}{b}$
4	$\frac{1}{4} - \frac{1}{4b}$	$\frac{b-1}{b} n^3$
$4n - 4b + 4$	$\frac{1}{4b} - \frac{1}{4bn}$	$\frac{b-1}{b} n^2$
$4n$	$\frac{1}{4} - \frac{1}{4b}$	$\frac{b-1}{b} n^3$
$4n^2 - 4nb + 4n$	$\frac{1}{4b} - \frac{1}{4bn}$	$\frac{b-1}{b} n^2$
∞	$\frac{3}{4bn}$	$\frac{3n^2}{b}$

When $n \ge 1$, all the RI counts listed in the two tables are monotone functions. By Theorem 2.2, matrix multiplication has Working-set Locality Monotonicity. Not proved here, but the LRU cache locality is also monotone. For element granularity, the third and fourth columns of Table 1 show RD values and counts. All the RD counts are monotone functions. The monotonicity conditions, Monotonic RI Injection defined in this section, and Sequence Embedding and Intercept Freedom from the last section are sufficient conditions but not necessary.

Locality Monotonicity is about the number of cache misses, not the miss ratio. In fact, in the same program, monotonicity can hold for miss count but not miss ratio. Consider the cold-start misses [23]. For element-granularity matrix multiplication, the cold-start miss ratio is $\frac{1}{n}$, the ratio of data size (infinite RIs in Table 1) to the number of memory accesses, which decreases as n increases.

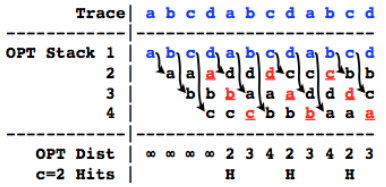

Figure 1: The OPT algorithm and stack for cyclic accesses with $n = 4$, reproduced from Brock et al. [8]

2.4 Bounds on the Least Cache Performance

In locality complexity, the best case or the lowest complexity is trivial: a single data item is accessed, and a single cell is needed to cache all data reuses. This section shows (1) the access pattern that has the highest complexity and (2) a reachable bound on the least cache performance (despite the complexity).

The locality complexity depends on cache management. First, we prove the complexity of random access for any caching policy.

Theorem 2.4 (Random Access Miss Ratio). *When accessing n data objects randomly, the miss ratio by any cache replacement algorithm that satisfies the inclusion property [26] is*

$$mr(c) = \begin{cases} \frac{n-c}{n} & 0 \le c \le n \\ 0 & c > n \end{cases}$$

Proof. If a cache replacement algorithm satisfies the inclusion property, it can be simulated using a stack algorithm. At each access, the stack stores all data items in linear order. Since access is random, each data item on the stack has an equal probability of being accessed. Hence, the stack distance is uniformly distributed between 1 and n, and the miss ratio is $\frac{n-c}{n}, 0 \le c \le n$. The stack distance is at most n, $mr(c) = 0$ for $c > n$. □

Mattson et al. [26] gave the definition of the inclusion property and stack algorithms and showed that the property holds for the cache replacement policies LRU, MRU, LFU, Random, and OPT. Theorem 2.4 shows that they all have the same miss ratio function, except OPT, which requires future knowledge and cannot be used when data access is random and nondeterministic. In addition, collaborative caching LRU-MRU and its generalized form were shown to also be stack algorithms [20, 21].

Mattson et al. [26] defined the OPT stack algorithm which computes the miss count for any data reference trace for all cache sizes. Brock et al. [8] gave an example that is reproduced in Figure 1. The example is a sequence of cyclic accesses of four data items.

For a given cache size, Belady [4] gave the optimal replacement method called MIN, which replaces the block that is accessed furthest in the future. In other words, MIN evicts the least imminently used data item. OPT simulates MIN for all cache sizes using a stack data structure. The OPT stack is a priority list for what blocks should be in the cache at any given time: a cache of size (1, 2, 3, ...) will contain the first (1, 2, 3, ...) blocks in the stack, and thus access to a data block is a miss when its stack position (OPT distance) is greater than the size of the cache [26].

Brock et al. [8] used Figure 1 to demonstrate the application of two rules for managing the OPT cache stack: (1) Upward Movement: A block can only move up when accessed, and then it moves to the top; (2) Downward Movement: Vacancies are filled by whichever a data item above it will not be used for the longest time in the future. A new item is treated as having vacated a new spot at the bottom. Moving an item downward to fill a vacancy represents its eviction from all caches smaller than the position of the vacancy (and the item with the most remote future use is evicted).

Brock et al. [8, Appendix A] proved that the OPT distance for cyclic accesses of n data items repeats between 2 and n.

Theorem 2.5 (Optimal Cyclic Access Locality). *The optimal locality of cyclic accesses is asymptotically the same as the locality of random accesses.*

Proof. The distribution of the OPT stack distances of cyclic accesses is as follows:

$$P(d) = \begin{cases} \frac{d}{n} & 2 \le d \le n \\ 0 & \text{otherwise} \end{cases}$$

The OPT stack distance is uniformly distributed between 2 and n with a periodity of $n - 1$ [8]. The miss ratio function of optimal caching is therefore:

$$mr(c) = \begin{cases} \frac{n-c}{n-1} & 0 \le c \le n \\ 0 & c > n \end{cases}$$

Since $n - 1$ is asymptotically equivalent to n, the miss ratio function is asymptotically equivalent to that of random access. □

We prove that for any program accessing n objects, with optimized caching, the worst-case locality is bounded and proportional. Specifically, the miss ratio is at worst no higher than the OPT miss ratio for cyclic accesses at the same cache and data size.

Theorem 2.6 (Worst Case Proportionality). *The cache locality can be optimized so that the worst-case miss ratio for any workload accessing n data times is*

$$mr(c) \le \begin{cases} \frac{n-c}{n-1} & 0 \le c \le n \\ 0 & c > n \end{cases}$$

Proof. Given a static trace of k accesses involving n distinct items, one strategy is to sort the items by their access frequency and then cache the top c most frequently accessed items. Since these c items are the most frequently occurring in the trace, each of them must appear at least $\frac{kc}{n}$ times. This observation ensures that caching these items can capture a significant portion of the overall accesses.

Hence, the following inequality holds:

$$mr(c) \le \frac{n-c}{n} \le \frac{n-c}{n-1}$$

□

Combining the three theorems in the section, we have the corollary that the worst-case bound is reachable.

Corollary 2.7. *The worst-case miss ratio function is reachable.*

Under optimal caching, cyclic traversals have the worst locality or highest data access complexity, as do random accesses. To our knowledge, this is the first time these two access patterns have been proved to be equivalent in their locality.

The proportionality is guaranteed given optimized caching. However, it does not require program optimization, just better cache management.

3 RELATED WORK

Complexity Analysis. Symbolic analysis of locality started in 1970s, including stationary stochastic processes [15] and independent reference models (IRM) [10, 25] and later Zipfian access patterns in Web workloads [2]. These are theoretical workloads.

For program locality, precise symbolic analysis was pioneered by cache miss equations [18] and reuse distance equations [7]. Both formulated the problem using integer equations and solved them using Presburger arithmetics implemented using the Omega calculator [30]. Recent improvements have reduced the analysis cost, but still required solving integer equations [3]. The solution has to be approximated to handle large programs [28, 37]. While these studies targeted overall cache performance, Reineke [32] developed symbolic worst-case analysis for real-time systems.

The working-set cache in this paper is the same as the uniform-lease cache, prototyped and evaluated on the FPGA hardware [29]. The lease cache is designed to enable software-hardware collaborative caching through lease programming. The ideal working-set cache has a varying size. Reber et al. [31] referred to it as the virtual cache size and studied the effect on a hardware cache with a fixed size for scientific kernels. Chen et al. [9] compared the uniform-lease cache and the LRU cache in theory and simulation.

Proportionality. Gu and Ding [19] proved that the LRU cache miss ratio of random access. The Random Access Miss Ratio Theorem shows that the same miss ratio for all stack algorithms. Since LRU is one of the stack algorithms, the new theorem generalizes the previous result. The Worst-Case Proportionality further generalizes and shows the same function as the upper bound miss ratio for all programs under optimized caching. A caching algorithm is random replacement, which does not have the same locality as random access. Random replacement miss ratio can be modeled by a binomial distribution [34, 35].

Optimal cache replacement in a single cache was given by Belady [4] in 1966. The technique is called MIN or Belady. Four years later, Mattson et al. [26] gave the OPT algorithm, which simulates optimal caching in all cache sizes. OPT is a stack algorithm, so its miss ratio function is monotone. It took over four decades before Michaud [27] proved that the miss ratio function is convex. The Worst-case Proportionality Theorem in this paper shows the upper bound of the OPT miss ratio function for all programs. Monotone and convex miss ratio functions exist above the bound derived in this paper, so the (previous results of) convexity and monotonicity do not imply Proportionality.

Cache Performance Analysis. While locality measures such as the miss ratio are traditionally numerical and a property of an execution, locality complexity is symbolic and a property of a program or an algorithm. Trace or online analysis, e.g., HPCToolkit [1], can analyze binary code and non-determinism in parallel executions. An online tool measures not just the memory cost but also other factors, e.g., power [17], and the actual running time. For monotonicity, complexity analysis can prove it for all problem sizes while testing cannot. Finally, proportionality can be used to infer optimization opportunities based on the observed performance.

4 SUMMARY

This paper has presented two properties of locality complexity, derived and proved for all programs. We have formalized two sufficient conditions of locality monotonicity and proved it for naive square matrix multiplication. In addition, we prove Proportionality, which states that the benefit of having more cache be at least linear. Monotonicity divides all programs into two types: monotone and non-monotone. Proportionality holds for all programs. The monotonicity is proved for fully associative LRU caches and working-set caches, and proportionality for all caching policies. These results hold for both element and block granularity caching.

ACKNOWLEDGMENTS

The authors wish to thank Woody Wu and Yekai Yan for the discussion on monotonicity and its proof and the reviewers of MEMSYS 2025 for the helpful comments and suggestions on the presentation of the paper.

REFERENCES

[1] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. 2010. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

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

[3] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, and P. Sadayappan. 2018. Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 32:1–32:26.

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

[5] Erik Berg, Håkan Zeffer, and Erik Hagersten. 2006. A statistical multiprocessor cache model. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 89–99.

[6] K. Beyls. 2004. *Software methods to improve data locality and cache behavior*. Ph. D. Dissertation. Ghent University.

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

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

[9] Dong Chen, Chen Ding, Fangzhou Liu, Benjamin Reber, Wesley Smith, and Pengcheng Li. 2021. Uniform lease vs. LRU cache: analysis and evaluation. In *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, Zhenlin Wang and Tobias Wrigstad (Eds.). ACM, 15–27.

[10] Asit Dan and Donald F. Towsley. 1990. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 143–152.

[11] Peter J. Denning. 1970. Virtual Memory. *Comput. Surveys* 2, 3 (1970), 153–189. https://doi.org/10.1145/356571.356573

[12] Peter J. Denning. 2005. The locality principle. *Commun. ACM* 48, 7 (July 2005), 19–24. https://doi.org/10.1145/1070838.1070856

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

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

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

[16] Chen Ding and Ken Kennedy. 1999. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[17] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. 2010. PowerPack: Energy Profiling and Analysis of High-Performance

Systems and Applications. *IEEE Transactions on Parallel and Distributed Systems* 21, 5 (2010), 658–671.

[18] 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).

[19] Xiaoming Gu and Chen Ding. 2008. *Reuse distance distribution in random access.* Technical Report URCS #930. University of Rochester, Rochester, NY. *a short version appeared in the 2008 MSPC workshop.*.

[20] 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.

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

[22] Allan Hartstein, Vijayalakshmi Srinivasan, Thomas R. Puzak, and Philip G. Emma. 2008. On the Nature of Cache Miss Behavior: Is It $\sqrt{2}$? *J. Instr. Level Parallelism* 10 (2008).

[23] M. D. Hill. 1987. *Aspects of cache memory and instruction buffer performance.* Ph. D. Dissertation. University of California, Berkeley.

[24] Song Jiang and Xiaodong Zhang. 2005. Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems. *Perform. Eval.* 60, 1-4 (2005), 5–29.

[25] W. F. King. 1971. Analysis of demand paging algorithms. In *Proceedings of IFIP Congress*. 485–490.

[26] 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.

[27] Pierre Michaud. 2016. Some Mathematical Facts About Optimal Cache Replacement. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 50:1–50:19. https://doi.org/10.1145/3017992

[28] Arjun Pitchanathan, Kunwar Grover, and Tobias Grosser. 2024. Falcon: A Scalable Analytical Cache Model. *Proc. ACM Program. Lang.* 8, PLDI, Article 222 (jun 2024), 25 pages. https://doi.org/10.1145/3656452

[29] Ian Prechtl, Ben Reber, Chen Ding, Dorin Patru, and Dong Chen. 2020. CLAM: Compiler Lease of Cache Memory. In *MEMSYS 2020: The International Symposium on Memory Systems, Washington, DC, USA, September, 2020*. ACM, 281–296.

[30] William W. Pugh and David Wonnacott. 1998. Constraint-Based Array Dependence Analysis. *ACM Transactions on Programming Languages and Systems* 20, 3 (1998), 635–678.

[31] Benjamin Reber, Matthew Gould, Alexander H. Kneipp, Fangzhou Liu, Ian Prechtl, Chen Ding, Linlin Chen, and Dorin Patru. 2023. Cache Programming for Scientific Loops Using Leases. *ACM Transactions on Architecture and Code Optimization* 20, 3, Article 39 (jul 2023), 25 pages.

[32] Jan Reineke. 2018. The Semantic Foundations and a Landscape of Cache-Persistence Analyses. *Leibniz Trans. Embed. Syst.* 5, 1 (2018), 03:1–03:52.

[33] Wesley Smith, Aidan Goldfarb, and Chen Ding. 2022. Beyond Time Complexity: Data Movement Complexity Analysis for Matrix Multiplication. arXiv:2203.02536 [cs.DS]

[34] Shaotong Sun, Yifan Zhu, Xingzhi Ye, and Chen Ding. 2024. Measuring Data Access Latency in Large CPU Caches. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*. 1–11.

[35] Richard West, Puneet Zaroo, Carl A. Waldspurger, and Xiao Zhang. 2010. Online cache modeling for commodity multicore processors. *Operating Systems Review* 44, 4 (2010), 19–29.

[36] William A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995), 20–24.

[37] 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. https://doi.org/10.1109/TC.2004.1275296

[38] 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.