A Relational Theory of Locality

LIANG YUAN, SKL of Computer Architecture, Institute of Computing Technology, CAS
CHEN DING, University of Rochester
WESLEY SMITH, University of Edinburgh
PETER DENNING, Naval Postgraduate School
YUNQUAN ZHANG, SKL of Computer Architecture, Institute of Computing Technology, CAS

In many areas of program and system analysis and optimization, locality is a common concept and has been defined and measured in many ways. This article aims to formally establish relations between these previously disparate types of locality. It categorizes locality definitions in three groups and shows whether and how they can be interconverted. For the footprint, a recent metric, it gives a new measurement algorithm that is asymptotically more time/space efficient than previous approaches. Using the conversion relations, the new algorithm derives with the same efficiency different locality metrics developed and used in program analysis, memory management, and cache design.

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

Additional Key Words and Phrases: Locality, memory hierarchy, working set

ACM Reference format:
Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Trans. Archit. Code Optim.* 16, 3, Article 33 (August 2019), 26 pages.
https://doi.org/10.1145/3341109

1 INTRODUCTION

Locality is a fundamental property of computation and a central principle in software, hardware, and algorithmic design. As defined by Denning, it is the "tendency for programs to cluster references to subsets of address space for extended periods" (Denning and Martell 2015, p. 143). Locality has been exploited to design high-performance caches and memory managers in operating systems. These systems have relied on a set of locality measures that enable memory systems to adapt to demand. In modern computer systems, "the increasing gap between processor and

The manuscript is new and not a revision of a previous conference paper.

This work is supported by the National Key R&D Program of China (Grant No. 2017YFB0202001), the National Science Foundation (Contract No. CNS-1909099, CCF-1717877, CCF-1629376, CNS-1319617), an IBM CAS Faculty Fellowship, the National Science Foundation of China (Contract No. 61328201, 61432018, 61602443), the Science Foundation of Beijing (L182053), and Guangdong Province Key Laboratory of Popular High Performance Computers 2017B030314073.

Authors' addresses: C. Ding and W. Smith, Computer Science Department, 3401 Wegmans Hall, University of Rochester, Rochester, NY 14627-0226; emails: {cding, wsmith6}@ur.rochester.edu; L. Yuan and Y. Zhang, School of Computer Architecture, Institue of Computing Technology, No. 6 Kexueyuan South Road Zhongguancun, Haidian District Beijing, China, 100190; emails: {yuanliang, zyq}@ict.ac.cn; P. Denning, Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943; email: pjd@nps.edu.

1544-3566/2019/08-ART33
https://doi.org/10.1145/3341109

Table 1. Example Locality Analysis Uses, Targets, and Metrics

Uses of locality analysis	Targets of analysis	Metrics
program analysis and optimization	data accesses	access frequency, reuse distances
virtual memory management, cache sharing	execution phases	working sets, footprints
cache design, performance modeling	cache systems	miss ratio curves, average eviction times

memory speeds has rendered the organization, architecture, and design of memory subsystems an increasingly important part of computer-systems design" (Jacob et al. 2010). Memory is also increasingly diverse, with different materials, configurations, and interconnects providing different tradeoffs among capacity, speed, cost, and other factors: "a well-implemented hierarchy allows a memory system to approach simultaneously the performance of the fastest component, the cost per bit of the cheapest component, and the energy consumption of the most energy-efficient component" (Jacob et al. 2010).

There is a large and growing body of literature on locality analysis and optimization. Locality has been defined in many ways; Table 1 shows examples of locality concepts in three groups, each by their target of analysis. For a program, the analysis measures its data accesses by the reuse distance or frequency (hotness). For resource sharing, the analysis measures the dynamic data demand by the working set or footprint. For cache design, the analysis measures the cache performance by the miss ratio curve or the average eviction time.

While most past studies focused on one locality definition, this article creates a unified mathematical framework encompassing widely used, distinct locality definitions. We will formalize and prove their relations. Through this standardization, we provide the groundwork for future study unimpeded by assorted and distinct previously used ideas and techniques. In addition, the framework we will introduce allows retrospective analysis of past locality research through reintroduction of formulae and definitions within the context of a more well defined and precise mathematical language.

We call the new framework the *relational theory of locality (RTL)*, which consists of precise mathematical descriptions of a set of locality definitions and their relations. It includes three categories, and locality is defined in each category with similar objectives and parameters.

- *access locality*: measures of locality for each memory access
- *timescale locality*: functions that measure locality with length of time as a parameter
- *cache locality*: functions that measure locality with cache size as a parameter

Locality definitions also differ by their levels of abstraction. An access trace is most concrete in that it encodes the complete memory behavior. From a trace, different definitions of locality extract and retain different aspects of the data access information. By establishing their relations, the new theory shows that timescale locality captures the most useful information from a trace and enjoys the highest time and space efficiency.

The relational theory is useful both pedagogically and in practice. Locality definitions are intuitively related. For example, data reuses in a program are likely beneficial, since they are likely cache hits if the cache is large enough. However, the intuition is not precise, because it cannot say how large is large enough. The relational theory shows the precise conversion between these and other concepts.

The practical benefits are many and can be divided into two areas. The first is measurement. If two locality definitions are mathematically equivalent, then a measurement technique developed for one metric can be used for both by converting between the two metrics. For example, two newest techniques, AET (Hu et al. 2018), for storage cache, and RDX (Wang et al. 2019), for CPU cache, use sampling to measure the miss ratio curve with an extremely low cost. Using the relational theory, Section 3.3 will show that these techniques, although independently developed with different areas of applications, produce results mathematically related to previous locality definitions and hence to each other.

The second and more important area is optimization. The relational theory shows how optimizing one metric may affect other metrics, e.g., how to minimize the number of misses in cache by transforming data reuses in a program. A theory inevitably makes simplifying assumptions, but given their assumptions, the theoretical relations are both universal, i.e., across programs, and reliable, i.e., across transformations.

Contributions

The article presents a taxonomy of common locality concepts with precise definitions and groups similar definitions into categories. It unifies concepts and equations introduced in preexisting literature and shows a set of new relations as follows.

In the case of access locality, it shows equivalence between definitions based on sequences and non-equivalence between definitions based on histograms. In the case of timescale and cache locality, it shows the first mathematical relation between the working set by Denning et al. (Denning 1968; Denning and Schwartz 1972) and the footprint by Xiang et al. (2011b, 2013) as well as introducing a new way to approach using Denning recursion on finite traces. The relation leads to new, simple proofs of boundedness and concavity. More importantly, it shows the mathematical relation among four previous techniques in computing the miss ratio, which have been used in recent publications without a comparison with each other. Further derivation also gives the first theoretical justification to a formula discovered four decades ago to model shared cache miss ratio (Easton and Fagin 1978). In addition, the article gives a new, short explanation of the previous formula for computing the footprint and an alternative that is asymptotically faster.

Finally, it summarizes with a relation graph that connects all locality definitions in all categories. The rest of the article will provide the pieces missing from previous works but that are necessary to show the complete relations.

The new theory has a limited scope. It defines and measures locality but does not address the problem of locality optimization (Aho et al. 2006; Allen and Kennedy 2001; Coffman Jr. and Denning 1973; Cooper and Torczon 2010; Meyer et al. 2003; Wolfe 1996). It measures the amount of data movement but not the time overhead, which depends on other factors such as latency, prefetching, and burstiness of communication. It neither considers spatial locality, nor does it optimize the data layout (Lavaee 2016; Petrank and Rawitz 2002). It assumes automatic cache management and does not solve the more general problem of I/O complexity (Elango et al. 2015; Hong and Kung 1981). Finally, the article does not cover all locality measures.

2 LOCALITY DEFINITIONS AND RELATIONS

We will first present an overview that divides the locality measurements into six categories and then present them in subsections.

2.1 Overview

A *trace* is a sequence of references to data or memory locations. Each reference is a memory address. We also call a reference a *trace element*, and its target a *data item*. The words “sequence,”

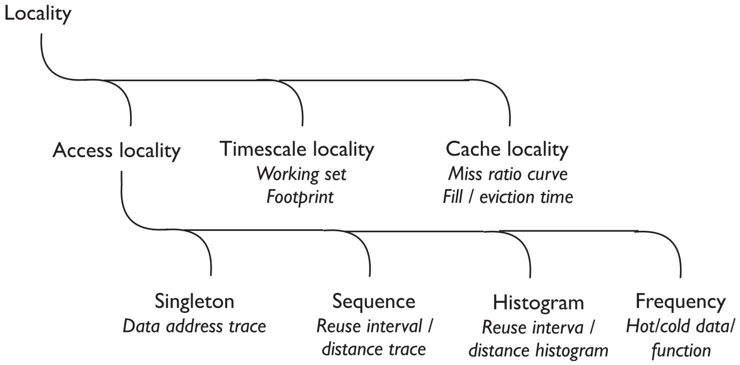

Fig. 1. The categories of locality measurements and new theoretical results (in italics).

"trace," and "execution" are used interchangeably, as are the phrases "memory access" and "memory address." We ignore any issue of granularity. A data item may be a variable, a data block, a page, or an object.

A *locality definition* is an equivalence relation among all execution traces—two executions are equivalent if and only if they have the same locality by the definition. Each locality definition partitions executions into equivalence classes.

Figure 1 shows locality definitions in three top-level and four second-level categories.

The first category is *access locality*, which quantifies locality for each access. It has four types as shown in Figure 1. The simplest is *singleton locality*—the locality is the execution itself.[1] Singleton locality defines the strictest notion of equivalence. Two executions have the same singleton locality if and only if they are identical. By comparing equivalence classes, we can precisely compare different definitions of locality.

We use the terms locality *definition* and *measurement* interchangeably. Singleton locality requires no measurement. Other locality definitions require a way to measure, and a way to measure locality also defines the locality. It is often more convenient to define locality based on another locality definition, not on an execution trace, e.g., the miss ratio defined from the reuse distance. We call it the locality *conversion*. The following sections show the relation between locality definitions by the conversion between each other. With the complete relation between locality definitions, conversion becomes synonymous to measurement, because to measure is to convert from singleton. A relational theory is also a measurement theory.

The following sections describe all locality categories. The sequence locality has the most information but also incurs the highest cost. The histogram locality allows for a compact representation. This benefit is exploited by the timescale locality to model the cache locality. These locality categories are progressively higher level, more abstract, and more efficient to use in practice. As running examples, we will use a number of trace examples composed with just three data elements a, b, c, including those repeating them once in the same order (i.e., $abc\ abc$), in opposite order (i.e., $abc\ cba$), or repeating indefinitely (i.e., $abc\ abc \dots$).

2.2 Sequence Locality

We describe sequence and histogram locality and leave the frequency locality to Section 2.6. We define the following:

[1] The name "singleton" is an adaptation of Lu and Scott, who defined determinism as an equivalence relationship among concurrent executions (Lu and Scott 2011).

n is the length of a trace.
$N = mt(1 \ldots n)$ is a memory address trace.
m is the number of distinct memory addresses accessed by the trace.
$M = \{e_1 \ldots e_m\}$ is the set of distinct memory addresses.

The locality may be measured by one of the following three sequences:

- *Address independent (AI) sequence.* Given an access trace, the AI sequence is constructed by renaming the memory addresses to $M = \{1 \ldots m\}$ and indexing them in order. The memory address is i if it is ith earliest in the order of first appearance in the trace. An AI sequence standardizes data-to-memory mappings. For example, two traces *abc abc* and *cba cba* have the same AI sequence e_1, e_2, e_3 e_1, e_2, e_3. AI locality is more abstract than singleton locality. If a program is run multiple times with the same input but different memory allocations, e.g., address space layout randomisation (ASLR), then the (singleton) trace changes but the AI sequence does not.
- *Reuse interval (RI) sequence.* For each access, the reuse interval is the increment of logical or physical time since the last access of the same datum. For example, the RI sequence is $\infty\infty\infty$ 333 for *abc abc* and $\infty\infty\infty$ 135 for *abc cba*. The reuse interval is ∞ if it is its first access. For a finite reuse interval, the minimal is 1 and the maximum $n - 1$. The reuse interval has been called the inter-reference interval (iri) in the working set theory (Denning 1968), inter-reference gap in LIRS (Jiang and Zhang 2002), reuse distance in StatCache and StatStack (Eklov et al. 2011), and reuse time in our earlier papers.
- *Reuse distance (RD) sequence.* For each access, the reuse distance is the number of distinct data accessed since the last access to the same datum, including the reused datum. For example, the RI sequence is $\infty\infty\infty$ 333 for *abc abc* and $\infty\infty\infty$ 123 for *abc cba*. The reuse distance is ∞ if it is its first access. For a finite reuse distance, the minimum is 1 (because it includes the reused datum), and the maximum is m. The reuse distance is the same as the LRU stack distance (Mattson et al. 1970), which is often called stack distance for short.

For either RI or RD, the locality may be represented by the entire sequence or be broken down into per-datum sequences:

- *Per datum (PD) sequence of reuse interval (PD·RI) and reuse distance (PD·RD)*, which converts a trace into a set of RI or RD sub-sequences $pd[e] = (f_e, r_2, \ldots, r_{n_e})$ for each datum e, where f_e is the time of e's first access, n_e the number of accesses, and r_i the reuse interval of ith access in PD·RI and the reuse distance of ith access in PD·RD. Note that $r_1 = \infty$ is omitted. For element a in *abc abc*, the PD·RI and PD·RD sequences are the same: $pd[a] = (1, 3)$.

The reuse interval r_i may be either *forward* or *backward*. It is the *forward reuse interval* of the $(i - 1)$th access and the *backward reuse interval* of the ith access. They are equivalent when used for whole-trace analysis but different when used in online analysis. The backward reuse interval shows the history, and the forward interval the future. There is the similar distinction between forward and backward reuse distance. The following discussion assumes the backward reuse.

2.2.1 Equivalence. The five definitions in the preceding section are all equivalent. This section shows this equivalence by mutual conversions between each pair of definitions.

It is trivial to show mutual conversion between RI and AI and between RI and PD·RI. For example, to convert from AI to RI, we traverse the trace and use a hash table to record the last access time of each element. At each access, the reuse interval is the difference between the current time and the last access time. The three definitions are shown as boxes in Figure 2 and the two pairwise

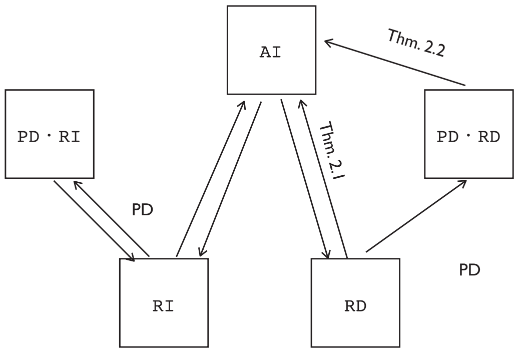

Fig. 2. Sequence locality definitions and their conversions.

conversions by four directed edges. By transitivity, AI and PD·RI are also mutually convertible. Hence, all three definitions are equivalent.

From past work (Mattson et al. 1970) in reuse distance measurement, AI can be converted to RD, which is then trivially converted to PD·RD. Next, we show two theorems that establish the conversion first from RD to AI and then from PD·RD to AI. The two conversion results, as shown by Figure 2, produce a completely cyclic graph including all five locality definitions, proving their equivalence.

Theorem 2.1. *The address-independent sequence AI can be built from the reuse distance sequence RD.*

Proof. The LRU stack contains the access sequence at the top position (Mattson et al. 1970). To obtain the AI sequence, we show that the RD sequence can construct the LRU stack as follows. When the reuse distance is ∞, a new data item i is created and placed on top of the stack (first position). At a finite reuse distance x, the data item at stack position x is moved to the top, and the items in positions $1 \ldots x - 1$ are moved down by one position. □

The construction of an AI trace is more difficult than from per datum (PD) reuse distances, because the order of reuses between data items is lost in the PD conversion.

Theorem 2.2. *The AI trace can be built from per datum reuse distances PD · RD.*

Proof. See appendix. □

Algorithm 1 gives a simple conversion from PD·RD to AI. A more complicated algorithm for converting PD·RD to AI, Algorithm 3, is presented in the appendix along with a proof of correctness; for readability, we give here only the simple algorithm along with a proof of equivalence. We do not give a direct proof of correctness for Algorithm 1. Algorithm 1 follows a simple procedure. Initially, all data elements are sorted by their first access time. At each position i of the AI trace, we put the element at the head of the list. Then we reinsert the element based on its next reuse distance. The algorithm omits the case when an element has no more reuses, in which case it is not re-inserted into the list at Line 7.

Algorithm 1 is based on a linked list, while Algorithm 3 uses vectors. Next, we show that the list-based algorithm is correct, because it generates the same trace as the vector-based algorithm does.

Theorem 2.3. *Algorithm 1 constructs the correct AI trace.*

ALGORITHM 1: PD·RD → AI conversion (list based)

```
1 $ai[1 \ldots n] \leftarrow 0$
2 $list[1 \ldots m] \leftarrow \{e_i\}$ sorted by first-access time
3 for $i = 1$ to $n$ do
4     let $list[1]$ be $e$
5     $ai[i] \leftarrow e$
6     let $d$ be the next reuse distance of $e$
7     remove $list[1]$ and re-insert $e$ as $list[d]$, moving down $list[d \ldots m-1]$ to $list[d+1 \ldots m]$
8 end
```

Proof. At each $ai[i]$, the list in Algorithm 1 is ordered the same as one sorted by *nextpos* in Algorithm 3. The two algorithms both start the same, with the order of the first-access time. The update to *nextpos* in Algorithm 3 has the same effect as re-insertion into the list in Algorithm 1. Note that the order in Algorithm 3 is a partial order by *nextpos* but a total order when consider *lastpos*.

Algorithm 1 selects the first element of the list. In Algorithm 3, the corresponding element is e with smallest $nextpos[e]$ (such that $nextpos[e] \geq i$). Since Algorithm 3 is correct, we must have $nextpos[e] = i$, so choosing the first list element by Algorithm 1 is always correct. □

The complexity of Algorithm 1 is $O(nd)$, where n is the length of the trace, and d the average reuse distance, which is $O(m)$ in the worst case. The complexity can be reduced by using a balanced search tree to store the ordered list. At each iteration, Line 7 removes the left-most child node and re-inserts it as the dth node. To find the insertion point and to maintain the balance, each tree node keeps a record of the sub-tree size. The complexity can be reduced to $O(n \log m)$.

2.3 Histogram Locality

Histogram construction (*HI*) produces two types of histograms:

- The RI histogram $ri(i)$, which counts the number of reuse intervals that equal i, $i = 1, \ldots, n-1, \infty$ and $0 \leq ri(i) \leq n$. The RI histogram $ri(i)$ for *abc abc* is 3 when $i = 3, \infty$ and 0 otherwise. It is a (unordered) summary of its reuse intervals.
- The RD histogram $rd(i)$, which counts the number of reuse distances that equals i, $i = 1, \ldots, m, \infty$ and $0 < rd(i) \leq n$. The RD histogram for *abc abc* is the same as its RI histogram shown before, i.e., $rd(i) = ri(i)$.

We denote the reuse interval and reuse distance histogram as $HI \cdot RI$ and $HI \cdot RD$, where *RI* and *RD* are the reuse interval and reuse distance trace discussed above, and *HI* is the histogram conversion. The HI conversion loses all information about memory address, access time, and order of reuses.

The reuse interval histogram was called the interreference density (Denning and Schwartz 1972). If we normalize the bins of a reuse interval histogram by dividing them with the total number of reuse $n - m$, then the histogram can be viewed as a probability function that represents the interreference distribution. The reuse distance histogram was called the locality signature (Zhong et al. 2009).

Reuse distances have a direct relation with cache performance, and the histogram is a compact summary. In cache analysis, the RD histogram gives the miss ratio of the fully associative cache (Mattson et al. 1970), direct-mapped or set-associative cache (Marin and Mellor-Crummey 2004; Nugteren et al. 2014; Qasem and Kennedy 2005; Smith 1976), and cache with other

Table 2. Space Requirements of Sequence and Histogram Locality

		RI/RD sequence	RI histogram $ri(x)$	RD histogram $rd(v)$
indexing parameter		time $t \in [1 \dots n]$	window length $x \in [1 \dots n-1]$	volume $v \in [1 \dots m]$
space cost	accurate	$O(n)$	$O(n)$	$O(m)$
	compact	n.a.	$O(\log n)$, $O(1)$	$O(\log m)$, $O(1)$

reuse-based replacement policies (Sen and Wood 2013) of all sizes. It is used to separate the locality effect by the program structure (Marin and Mellor-Crummey 2004) and the load/store operation (Fang et al. 2005), model the change of locality as a function of the input (Fang et al. 2005; Marin and Mellor-Crummey 2004; Zhong et al. 2009) and the degree of parallelism (Wu and Yeung 2011), and predict the performance of different cache designs and parameters (Wu et al. 2013; Zhong et al. 2007), making it the most widely used metric of access locality.

2.3.1 Compactness. A sequence is indexed by time. A histogram is an enumerated representation by value (which may be either a reuse interval or a reuse distance). Indexing is synonymous to sorting: a sequence is sorted by time and a histogram by value. Just as the Fourier transform converts a signal from a time function to a frequency distribution, histogram conversion changes the locality representation from a trace to a distribution.

Table 2 compares sequence and histogram locality in its parameter and space consumption. Sequence locality takes linear space, but histogram locality can be approximated and stored in logarithmic or constant space. A histogram enables a compressed representation.

The main benefit is compactness. Once the time information is removed and only values are kept, the values are sorted, and they can be stored by *binning* or bucketing. A basic solution divides the full value range evenly. This solution is constant size and general, but it may waste space when values are sparsely distributed. A specialized solution is a *reference histogram*, which sorts all reuse distances by their values and divides them evenly into 1,000 bins, so each bin stores exactly 0.1% of reuse distances (Zhong et al. 2009). A reference histogram may still waste space, because two adjacent bins may store identical values. Another solution is recursive division, which stops dividing a group when its values are identical (Marin and Mellor-Crummey 2004).

Logarithmic size histograms are commonly used. In the basic solution, the ith bin stores the range $[2^i, 2^{i+1} - 1]$. There are at least two ways to improve precision. The first is to record the average value in each bin and assume a constant or linear distribution by the values in the range (fitted to give the same average) (Fang et al. 2005). The second is a k*-sublog* histogram, which further divides a power-of-two range into 2^k sub-ranges for a pre-determined constant $k > 0$ (Xiang et al. 2011a, 2013). For example, a 8-sublog histogram uses 256 sub-ranges and is accurate from 0 to 511 and then divides each successive power-of-two ranges into 256 bins. The asymptotic space cost is logarithmic rather than linear.

Compactness implies approximation, which means sparsity rather than imprecision. A sublog RD histogram of size $O(\log n)$ can be used to compute $O(\log n)$ miss ratios. The computed miss ratios are accurate. Section 2.5.3 will show a similar result for RI histograms.

2.3.2 Non-equivalence. The two types of histograms are not equivalent. This can be proved by showing memory traces with different reuse interval histograms but the same reuse distance histogram and memory traces with different reuse distance histograms but the same reuse interval histogram.

Consider the following four traces:

$$\begin{array}{rl} t_1\text{:} & e_1, e_2, e_3, e_4, e_3, e_4, e_1, e_2, e_3, e_4, e_3, e_2, e_3, e_2, e_3, e_4, e_3, e_2, e_1 \\ t_2\text{:} & e_1, e_2, e_3, e_4, e_3, e_2, e_1, e_2, e_3, e_4, e_3, e_2, e_3, e_4, e_3, e_4, e_3, e_2, e_1 \\ \hline t_3\text{:} & e_1, e_2, e_3, e_4, e_3, e_4, e_1, e_2, e_3, e_4, e_3, e_2, e_1 \\ t_4\text{:} & e_1, e_2, e_3, e_4, e_3, e_4, e_2, e_1, e_3, e_4, e_3, e_2, e_1 \end{array}$$

It can be shown that t_1, t_2 have the same RI histogram but different RD histograms and t_3, t_4 have the same RD histogram but different RI histograms. A locality property useful in modeling cache sharing is composability, discussed in Section 2.5.5. A consequence of the non-equivalence is that the RI histogram is composable, but the RD histogram is not.

Next we introduce the timescale locality, which is based on the RI histogram, and it is both compact and composable.

2.4 Timescale Definitions of Locality

A timescale is a length of time that may be measured in seconds or years in physical time or number of memory accesses in logical time. A timescale metric is a mathematical function $f(x)$, where x ranges across all timescales, i.e., $x \geq 0$. It shows the growth of the working set size over timescales.

2.4.1 The Denning Working Set Recursion. The original timescale metric of locality is the average working set size (WSS) $s(x)$ formulated by Denning (1968). By adopting a probabilistic approach, Denning and Schwartz (1972) derived the recursive formula for $s(x)$. Initially, the working set is empty $s(0) = 0$, and the miss ratio is 100% $m(0) = 1$. The function $m(x)$ is the *time-window miss ratio*. At the window length x, an access is a miss if its reuse interval t is greater than x, that is, $m(x) = P(t > x)$. The working set size is computed by iteratively adding the time-window miss ratio,

$$s(x) = s(x-1) + m(x-1) = \sum_{i=0}^{x-1} m(i) = \sum_{i=0}^{x-1} P(ri > i). \tag{1}$$

We call Equation (1) the *Denning working set recursion* or Denning recursion for short. It is inductive: The WSS at x is the WSS at $x - 1$ plus the working set increase, which is the time-window miss ratio. For the infinite trace *abc abc* ..., we have $m(x) = 1$ for $0 \leq x \leq 2$ and $m(x) = 0$ for $x \geq 3$. The Denning recursion computes

$$s(x) = \begin{cases} x, & 0 \leq x \leq 3 \\ 3, & x > 3 \end{cases}.$$

2.4.2 Footprint. In an execution, every consecutive sub-sequence of accesses is a time window, formally as (t, x), where t is the end position and x the window length. The number of distinct elements in the window is the *working set size* $\omega(i, x)$ (Denning 1968). For a length x, the footprint $fp(x)$ is the average working set size, computed by the total working set size divided by the number of length-x windows:

$$fp(x) = \frac{1}{n - x + 1} \sum_{t=x}^{n} \omega(t, x). \tag{2}$$

For the infinite long trace *abc abc* ..., the footprint is the same as before, i.e., $fp(x) = s(x)$. It is still the same for *abc abc*. The footprint for *abc cba* is $fp(x) = 0, 1, 1.8, 2.5, \frac{8}{3}, 3$ for $x = 0, 1, 2, 3, 4, 5+$. Comparing the two footprint functions, we see the difference in locality in that the second function grows slower than the first.

2.4.3 Computing the Footprint. Xiaoya Xiang gave the following formula to compute the footprint from reuse intervals and the times of first and last accesses (Xiang et al. 2011b):

$$fp(x) = m - \frac{1}{n-x+1}\left(\sum_{i=x+1}^{n-1}(i-x)ri(i) + \sum_{k=1}^{m}(f_k - x)I(f_k > x) + \sum_{k=1}^{m}(n-x+1-l_k)I(n-x+1 > l_k)\right). \tag{3}$$

The symbols in the Xiang formula are as follows:

- $ri(i)$: the number of accesses whose reuse interval is i.
- f_k: the first access time of the kth datum (counting from 1).
- l_k: the last access time of the kth datum (counting from 1).
- $I(p)$: the predicate function to 1 if p is true; otherwise, 0.

Xiang et al. (2011b) used two pages in their paper to derive the formula based on "differential counting" of how the working set changes over successive windows. Next is a new, shorter explanation. The idea is "absence counting," by starting with assumption of all data in all windows and then counting all absences and subtracting their effects. For people who have filed income tax in the United States, taking deductions is a familiar process.

The first deduction is based on data reuses. If a reuse interval i is greater than x, then there are $i - x$ windows of length x that do not access the reused datum. The working set size should be reduced by $i - x$ to account for this absence. The total absence from all reuses is $\sum_{i=x+1}^{n-1}(i-x)ri(i)$.

The next two deductions follow a similar rationale. If the kth datum is first accessed at time f_k and $f_k > x$, then it is absent in the first $f_k - x$ windows of length x. Similarly, it is last accessed at $l_k < n - x + 1$, and it is absent in the last $n - x + 1 - l_k$ windows of length x. The total adjustment are shown by the last two terms of the Xiang formula.

2.4.4 Relation between the Working Set and Footprint. This section shows an equivalence relation between the Denning recursion and a simplified Xiang formula.

First, we introduce the limit-case Xiang formula. If a trace is infinitely long $n = \infty$, then the footprint is $\lim_{n\to\infty} fp(x)$. The limit-case formula is much simplified, because it uses only the reuse interval and not the first- and last-access times,

$$\lim_{n\to\infty} fp(x) = m - \sum_{i=x+1}^{\infty}(i-x)P(ri = i),$$

where $P(ri = i)$ is the portion of accesses that have reuse interval i. Consider the trace: *abcabc* .... Since every access has the same reuse interval (3), we have $P(ri = 3) = 1$. It is easy to verify that $\lim_{n\to\infty} fp(x) = x$ for $x = 0, 1, 2$, and 3 for $x \geq 3$. The Denning recursion computes the same result. The example suggests an equivalence relation between the two. We prove a general relation for not just an infinite long trace but any length trace.

We can use the limit-case Xiang formula for finite length traces except for one problem: We cannot calculate $i - x$ when $i = \infty$. We next define the *reuse-interval term footprint*, which uses the limit-case formula but treats infinite long reuse intervals separately as follows[2]:

[2] This is mathematically equivalent to treating infinite long reuse intervals as intervals of length n, which was used by Denning and Slutz to count the end corrections for space-time working set (Denning and Slutz 1978) and by Liang et al. for deriving the equivalence between the footmark and the Denning recursion (Yuan et al. 2018).

$$rtfp(x) = m - \sum_{i=x+1}^{n-1} (i-x)P(ri = i) - (n-x)P(ri = \infty). \tag{4}$$

Theorem 2.4 (WS-RTFP Equivalence). *The Denning recursion and the reuse term footprint differ by a constant,*

$$s(x) = rtfp(x) - rtfp(0),$$

for all integer $x \geq 0$.

Proof. The relation is proved by induction. In the base case, $s(0) = rtfp(0) - rtfp(0) = 0$. Assuming $s(x) = rtfp(x) - rtfp(0)$, we prove the inductive case. In the following derivation, $m(x)$ in the Denning recursion is the time-window miss ratio, and m in the reuse term footprint is the data size.

The increment of Denning recursion is

$$s(x+1) - s(x) = m(x) = P(ri > x).$$

The increment of the reuse term footprint is the same:

$$\begin{aligned} rtfp(x+1) - rtfp(x) &= m - \sum_{i=x+2}^{n-1} (i-x-1)P(ri = i) - (n-x-1)P(ri = \infty) \\ &\quad - \left(m - \sum_{i=x+1}^{n-1} (i-x)P(ri = i) - (n-x)P(ri = \infty)\right) \\ &= P(ri > x). \end{aligned}$$

Hence, the inductive hypothesis is correct, as is the theorem. □

The same relation holds if we do not count infinite reuse intervals. We compute the probability as $P'(ri = i) = \frac{ri(i)}{n-m}$, where the function $ri(i)$ is the number of reuse intervals of length i, and m is the number of infinite reuse intervals, i.e., the data size. For Denning recursion, we define $s(x+1) = s(x) + P'(n > ri > x)$, and for the limit-case Xiang formula, we define $rtfp(x) = m - \sum_{i=x+1}^{n-1}(i-x)P'(ri = i)$. A similar derivation shows that these definitions are still equivalent, i.e., $s(x) = rtfp(x) - rtfp(0)$. In addition, for infinitely long traces, we have $rtfp(0) = 0$, so $s(x) = rtfp(x)$.

2.4.5 *Finite Trace Denning Recursion.*

The original timescale definition of locality, the Denning recursion, was derived based on stochastic assumptions—that a trace is infinite and generated by a stationary Markov process, i.e., a limit value exists (Denning and Schwartz 1972). Later studies used the formula on finite-length traces, with adjustments to account for boundary effects (Denning and Slutz 1978; Slutz and Traiger 1974). Here we will present a straightforward way to incorporate these boundary effects into the Denning recursion; this allows the Denning recursion to correctly operate on finite traces.

Define a "run" to be a period during which a data block is in the working set, i.e., a reference to that block always exists in the sliding length-x window. Recently, Li et al. (2019)'s definition of *lease* cache allows for a modern interpretation of the working set as such: Each data block is assigned a lease equal to window length. If this lease expires, then the block is removed from the cache, and the lease is renewed on an access. A run begins with a miss and contains zero or more reuse intervals (see Figure 3).

Here we examine runs and working sets on example trace *abcaabdd* with window length 3. At each element in the trace, a 1 in a row indicates that that element is currently in the working set.

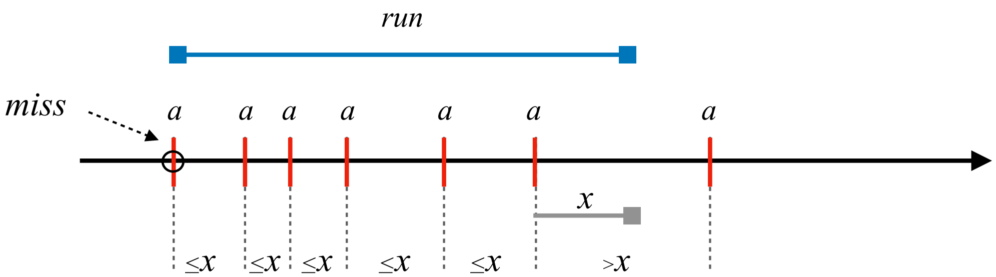

Fig. 3. Consider a sequence of accesses to data block *a*. Beginning with a miss, every consecutive reuse interval less than window length contributes to that run. The run ends with a reuse interval greater than window length. Note that the last window *x* contains the last access to *a* within the run.

Misses, which begin runs, are 1s directly preceded by 0s and are shown in bold in this example. Each column represents the working set at that point in the trace.

	{}	*a*	*b*	*c*	*a*	*a*	*b*	*d*	*d*
a	0	**1**	1	1	1	1	1	1	0
b	0	0	**1**	1	1	0	**1**	1	1
c	0	0	0	**1**	1	1	0	0	0
d	0	0	0	0	0	0	0	**1**	1

Because each run begins with a miss and each miss begins a run, the number of runs is the miss count $mc(x)$. Let $st(x)$ denote the sum of the lengths of all runs. To adapt Equation (1) to finite traces, we look at the effect on $st(x)$ of incrementing window size from x to $x + 1$. Were the trace infinite, each run would increase in length one unit, and we would have the relation $st(x + 1) = st(x) + mc(x)$. Dividing by n here yields Equation (1). For finite traces, the only runs not incremented by 1 by incrementing window size are runs where the distance from the last access of the element to the end of the trace is $\leq x$. Let $e(x)$ denote the number of such elements given window length x; an equivalent notion is the number of distinct elements appearing in the last window of the trace. Then Equation (5) correctly adjusts the Denning recursion for boundary effects of finite traces:

$$st(x + 1) = st(x) + mc(x) - e(x). \tag{5}$$

Dividing both sides of Equation (5) by n results in the modified form of Equation (1).

When the trace length is infinite, the $e(x)$ term vanishes when divided by n and the unmodified Denning recursion is correct. This was shown first by its derivation (Denning and Schwartz 1972).

2.4.6 *From Footprint to Reuse Interval.* Denote total working set size as $\mathcal{W}(x) = (n - x + 1)fp(x)$. Using the Xiang formula, the first- and second-order finite differences of $\mathcal{W}(x)$ are as follows:

$$\begin{aligned} \Delta\mathcal{W}(x+1) &= \mathcal{W}(x+1) - \mathcal{W}(x) = m + \sum_{i=x+1}^{n} ri(i) - \sum_{f_e < x+1} 1 - \sum_{n-x<l_e} 1 \\ \Delta^2\mathcal{W}(x+1) &= \Delta\mathcal{W}(x+1) - \Delta\mathcal{W}(x) \\ &= -ri(x) - \sum_e I(f_e = x) - \sum_e I(l_e = n - x + 1). \end{aligned}$$

Therefore, footprint can be used to derive the reuse interval histogram if the first and last access times are known.

2.5 Cache Definitions of Locality

The practical purpose of a locality theory is cache performance. This section discusses the metrics of cache locality and their relation with timescale locality. It reviews two previous methods for computing the miss ratio and presents two asymptotic improvements, one in time and the other in space. Finally, it discusses cache sharing, associative, multi-level caches, and spatial locality.

2.5.1 Performance of Fully Associative LRU Cache. The following metrics have been defined to model the performance of fully associative LRU caches:

- *miss ratio* $mr(c)$, which is the portion of memory accesses that are cache misses.
- *inter-miss time* $im(c) = \frac{1}{mr(c)}$, which is the average number of accesses between two consecutive misses.

By restricting the problem to fully associative caches, we can define cache performance for all integer cache sizes. There are important reasons to model the general cache size $c \geq 0$, not just powers of two. First, some real-cache solutions requires the miss ratio of all cache sizes and not just the size of the target cache.[3] Second, cache in practice is often shared. The occupancy of a program in shared cache can be any size, not just powers of two. Third, in software cache and some hardware cache, the size may not be powers of two. It is often useful to know how much the miss ratio changes when a program is given 10% more space than it currently has in cache.

2.5.2 Converting from Timescale Locality. We can compute the miss ratio from the timescale locality in two ways.

Denning-HOTL Conversion. The miss ratio can be computed by the finite difference of the footprint. When the cache size is the footprint $c = fp(x)$, the cache stores and only stores the working set of the last length-x window. The next access is a miss if and only if it expands the working set. For example, let the miss ratio be 10%. Then 10% times the working set size is increased by 1, so the average increase is 0.1. We take the footprint increase, i.e., $fp(x+1) - fp(x)$, as the miss ratio. Equation (6) formulates this conversion using the finite difference:

$$mr(c) = \Delta fp(x)\Big|_{fp(x)=c}, \tag{6}$$

where Δ is the finite-difference operator, i.e., $\Delta f(x) = f(x+1) - f(x)$, and c the cache size. Denning (1968) was the first to use this conversion. HOTL applied it to the footprint (Xiang et al. 2013). We call it the Denning-HOTL conversion.

The original working set size is given by the Denning-Schwartz forumla $s(x)$ (see Section 2.4.1). To use it to compute the miss ratio, we substitute $fp(x)$ in Equation (6) with $s(x)$. In either case, the conversion has the same form but uses a different timescale metric. This shows that Denning-HOTL is a general conversion from timescale locality to cache locality.

From the relation between the two timescale locality definitions studied in the previous section, we can analyze the relation between their converted miss ratios. In particular, from the WS-RTFP equivalence (Theorem 2.4), we see that the miss ratios computed using the Denning-Schwartz formula are exactly the same as those from the limit-case Xiang formula, shown by the following equation:

[3] Past work showed that the effect of cache associativity can be estimated using the full reuse distance distribution (Marin and Mellor-Crummey 2004; Nugteren et al. 2014; Smith 1976), which is equivalent to the miss ratio curve of the fully associative LRU cache.

$$\Delta s(x)\Big|_{s(x)=c} = \Delta rtfp(x)\Big|_{rtfp(x)-rtfp(0)+mx/n=c}.$$

Reuse-interval Conversion. The second way to convert is based on reuse interval. When the cache size is the footprint $c = fp(x)$, the cache stores and only stores the working set of the last length-x window. The next access is a miss if and only if its data have not been accessed in the last window, i.e., its backward reuse interval is greater than x,

$$mr(c) = P(ri > x)\Big|_{fp(x)=c}. \tag{7}$$

To use the Denning recursion to compute the miss ratio, we substitute $fp(x)$ in the preceding equation with $s(x)$. Like Denning-HOTL, the reuse-interval conversion is a general conversion from timescale locality to cache locality.

In practice, reuse-interval conversion has two benefits. First, it counts the cold-start misses correctly. These are first accesses whose reuse interval is infinite, since $ri > ft(c)$ for all c. Second, in short traces, e.g., sampled executions, the footprint may not be concave at the timescale close to the trace length, so the miss ratios computed by Denning-HOTL may not be monotone. Reuse-interval conversion, however, guarantees monotone miss ratios.

Monotonicity. There is a gap between the footprint of two consecutive timescales. For example, we have $fp(2) = 1.8$ and $fp(3) = 2.5$ for *abc cba*. The conversion may "miss" a cache size, i.e., computing the miss ratio for a size higher and lower but not the target size. This gap problem is solved by monotonicity, that is, the miss ratios are monotonically non-increasing with the cache size.

By its definition, the Denning recursion is concave; hence, the miss ratio computed by Denning-HOTL conversion is monotone. From the WS-RTFP equivalence (Theorem 2.4), the limit-case Xiang formula differs by a constant term; therefore, the limit-case Xiang formula is also concave.[4] For the reuse-interval conversion, Xiang et al. (2011b) showed that the Xiang formula is monotone, and hence the miss ratio computed from the footprint is monotone. If using the limit-case Xiang formula, then Xiang et al. (2013) proved that the reuse-interval conversion by Equation (7) computes identical results as Denning-HOTL by Equation (6).

Based on monotonicity, we know that the miss ratio of any missing cache size lies between the miss ratios of the adjacent, computed cache sizes. This is similar to the case when we make a physical measurement, the target value falls between two markings on the instrument.

2.5.3 Computing the Miss Ratio Curve Incrementally in Linear Time. The full range of the timescale is from 1 to n. From Equation (7), computing the miss ratio for cache size c is the same problem as finding the timescale x such that $fp(x) = c$. However, usually we only need a relatively small part. One weakness of the Xiang formula is that to compute the footprint for any timescale it requires the whole reuse interval histogram as well as all first access times and last access times. We present a lemma and two new formulae for computing the footprint. One of the two new formulas requires just a partial range of reuse intervals, first access times, and last access times.

The following lemma constructs a connection between the reuse interval histogram and the first and last access times.

Lemma 2.5. $\sum_{i=1}^{n-1} i \times ri(i) = \sum_{e=1}^{m}(l_e - f_e)$

Proof. A reuse has two properties we consider here: which element the reuse accesses and the reuse interval. The left and right sides of this lemma compute the summation of all reuse intervals from these two perspectives.

[4] Xiang et al. (2013) first proved that the limit-case footprint is concave. Here we have given a different proof based on the WS-RTFP equivalence.

To form the left side, all reuses are sorted according to their reuse interval, i.e., the reuse interval histogram. Recall that $ri(i)$ records the number of accesses with reuse interval i. So $i \times ri(i)$ is the summation of all reuse interval of i and thus the left term traverses all accesses.

To interpret the right side, all reuses are sorted according to their accessed elements. For each element e the summation of all its reuse intervals is $l_e - f_e$. Thus the right term also sums all reuse intervals. □

Substituting the lemma into Xiang formula, we obtain the following two new formulas. We call the first additive formula, since it uses the reuse interval histogram as a positive term. We refer to the second formula as the incremental formula. We provide general explanations for deriving them directly,

$$(n-x+1)fp(x) = xm + \sum_{i=1}^{n-1} \min(i,x) \times ri(i) - \sum_{e=1}^{m}(x-f_e)I(x-f_e) - \sum_{e=1}^{m}(l_e-n+x-1)I(l_e-n+x-1). \quad (8)$$

Additive formula. Formula 8 follows the idea that if one window contains an element more than once, then the footprint counts the first appearance and ignores the rest. For an access $mt(t)$ with reuse interval i, there are a total of $\min(i,x)$ windows where $mt(t)$ contributes to $fp(x)$ as a first appearance, i.e., $\omega(t-\min(i,x)+1,x)\ldots\omega(t,x)$. The first accesses and last accesses contribute to the footprint in a similar way.

$$(n-x+1)fp(x) = xn - \sum_{i=1}^{x-1}(x-i) \times ri(i) - \sum_{e=1}^{m}(x-f_e)I(x-f_e) - \sum_{e=1}^{m}(l_e-n+x-1)I(l_e-n+x-1). \quad (9)$$

Incremental formula. Formula 9 first assumes $fp(x) = xn$, i.e every access contributes x. Then it derives the correct value by subtracting the redundant counts. If an access's reuse interval is greater than or equal to x, then this access and its previous access do not exist in a same window. Thus there is no redundancy. If the reuse interval of an access $mt(t)$ is smaller than x, then it incurs $x-i$ redundant counts, since this access and its previous access exist in $x-i$ windows, i.e., $\omega(t-x+1,x)\ldots\omega(t-i,x)$. The first accesses and last accesses contribute to the footprint in a similar way.

If we only need a partial range of footprint $fp(1\ldots x)$, then the incremental formula allows for storing only a part of reuse interval histogram $ri(1\ldots x-1)$, the first access time portion that is smaller than x and the last access time portion that is greater than $n-x+1$.

2.5.4 Using Compact Histograms. Another weakness of the Xiang formula (Equation (3)) is that the entire reuse-time histogram is required when computing the footprint of any timescale x. The total time and space to compute the complete footprint $fp(x)$ for all timescales is $O(n)$. For real-world applications, n is extremely large, so the linear cost is too high. In addition, when modeling cache performance, constant factor analysis, e.g., the effect of using 10% more cache, is usually sufficient.

In this section, we show how to compute the footprint using a compact histogram. An example is the *sublog* histogram. As described in Section 2.3.1, a sublog histogram is a logarithmic histogram where each bin range is divided into a constant number of sub-bins. The size of a sublog histogram is $O(\log n)$. By choosing the number of subrange bins, a sublog histogram strikes a balance between

ALGORITHM 2: Logarithmic time and space Xiang formula

```
$totalSum = totalCount = 0$
for $j = 1$ to $b$ do
    $totalCount = totalCount + \mathit{rifl}(i).cnt$
    $totalSum = totalSum + \mathit{rifl}(i).sum$
end
$partialCount = partialSum = 0$
for $i = 1$ to $b$ do
    $fp(\mathit{rifl}(i).min) = m - \frac{(totalSum - partialSum) - (totalCount - partialCount) * \mathit{rifl}(i).min}{n - \mathit{rifl}(i).min + 1}$
    $partialCount = partialCount + \mathit{rifl}(i).cnt$
    $partialSum = partialSum + \mathit{rifl}(i).sum$
end
```

the cost and the "resolution," i.e., the number of cache sizes for which we can compute the miss ratio.

In profiling, we take one sublog histogram for all reuse intervals and all first- and last-access times. We call the histogram $\mathit{rifl}(1 \ldots b)$, where b is proportional to $O(\log n)$. The time cost of profiling is linear $O(n)$. The space is the size of the histogram $O(\log n)$. Each bin $\mathit{rifl}(i)$ stores information in the following fields:

- $\mathit{rifl}(i).min$ is the lower bound value of the ith range
- $\mathit{rifl}(i).cnt$ is the number of values in the ith range
- $\mathit{rifl}(i).sum$ is the total value of all reuse intervals in the ith range

Algorithm 2 computes the Xiang footprint using the sublog histogram. It has two loops each runs b iterations with constant work per iteration, so the time and space complexity is $O(b)$, which is $O(\log n)$.

The number of footprint values it computes is limited by the number of bins in the histogram. For each bin i, it computes $fp(x)$ at $x = \mathit{rifl}(i).min$ the exact value as computed by the original Xiang formula. The algorithm can be used unchanged for any compact reuse interval histogram.

2.5.5 Composability and Cache Sharing. On modern multicore processors, the cache is shared at one or more levels. Ding et al. (2014) defines *composability*, which means that the locality of a co-run can be computed from the locality of solo runs. The reuse interval is composable between independent programs if they are uniformly interleaved. In a co-run, each reuse interval in a solo run is increased by a constant factor. The reuse distance cannot be converted in the same way. However, the RD sequence is composable indirectly through the equivalence with the RI sequence, shown in Section 2.2.

Cache sharing has been modeled using the concurrent reuse distance (CRD) (Schuff et al. 2010; Wu and Yeung 2013). CRD computes the miss ratio accurately, but it is not composable. Many other techniques are hybrids where the locality is by reuse distance and the interference is by footprint (Chen and Aamodt 2009; Suh et al. 2001; Xiang et al. 2011a), including one of the first models of multicore cache (Chandra et al. 2005). An earlier model is given by Easton and Fagin (1978), which we will discuss in Section 3.3.4.

The effect of cache sharing has also been considered for threads that share data (Luo et al. 2017) and for independent programs with all possible interleavings (Brock et al. 2018). The all-interleaving result shows that for independent programs, the serial execution has the best locality.

2.5.6 Set Associative and Multi-level Caches. Until now, cache locality has been limited to fully associative LRU caches. However, the same locality definitions have been used to model many aspects of caching. As mentioned in Section 2.3, the effect of cache associativity can be modeled using the reuse distance histogram (Marin and Mellor-Crummey 2004; Nugteren et al. 2014; Qasem and Kennedy 2005; Smith 1976), as well as non-LRU policies (Sen and Wood 2013). Luo et al. (2018) showed that footprint enables more general models of cache associativity, as well as modelling the sub-block cache, and these models can be used together. Furthermore, the effect of multi-level caches, i.e., the effect of cache “filtering,” can be modeled using an extension of the footprint called the ‘victim footprint” by Ye et al. (2017) and “average eviction time” (AET) by Hu et al. (2018).

2.5.7 Spatial Locality. Data layout is important for cache performance. Gu et al. (2009) measured spatial locality as the change in temporal locality when increasing the block size. Gupta et al. (2013) defined locality as the probability of reuse, where the two types of histograms, RI and RD, give the likelihood of reuse in next-n-addresses and next-n-unique-addresses. Locality optimization has no known polynomial time solutions, whether it is to minimize the conflict misses, i.e., the Petrank-Rawitz limit (2002), or the capacity misses (Lavaee 2016). However, many effective solutions exist, including the reference affinity based on timescale co-occurrences (Liu et al. 2018; Zhang et al. 2006). We do not consider spatial locality in the relational theory.

2.6 Frequency Locality

Frequency is concise—for any n accesses to m data, the average access frequency per datum is n/m, a single number. This is one of the most commonly used approaches to measuring program locality; as such, we benefit from understanding how it relates to our previous definitions.

It is commonly known as “hotness” (Chilimbi et al. 1999; Rubin et al. 2002). Program data with a greater number of reuses are hotter. The locality is better if the “temperature” is higher. However, the ratio completely ignores the order of data access. The following three traces have the same access frequency but different locality. We name the first two following (Denning and Kahn 1975) and the last one following (Ding and Kennedy 2004):

$$\begin{aligned} cyclic &: abc\ abc \\ sawtooth &: abc\ cba \\ fused &: aa\ bb\ cc \end{aligned}$$

The locality depends on not just the frequency but also the recency of reuse. Although the three traces reuse the same data, the locality of *fused* is better than *sawtooth*, and *sawtooth* is better than *cyclic*. The closer the reuse, the better the locality.

In theory, Snir and Yu showed that the complete locality cannot be captured by a fixed size representation (Snir and Yu 2005). One way to measure locality is $mr(c)$ for all $c \geq 0$. The Snir-Yu limit implies that the frequency conversion has lost too much information—it is impossible to compute the miss ratio from a fixed number of access frequencies.

Not all locality definitions are equally usable. For the example, *fused* is optimal, because no other access order can further reduce any reuse distance. This optimality is obvious when analyzed using the reuse distance but not using the footprint or miss ratio.

3 THE RELATIONAL THEORY

3.1 The Complete Relations

Figure 4 shows the relation graph, where each node is a definition of locality, each directed edge a conversion and, if the edge has a cross (×), the assertion that no such conversion exists. An undirected edge means two directed edge in opposite directions. The conversions are injective. A

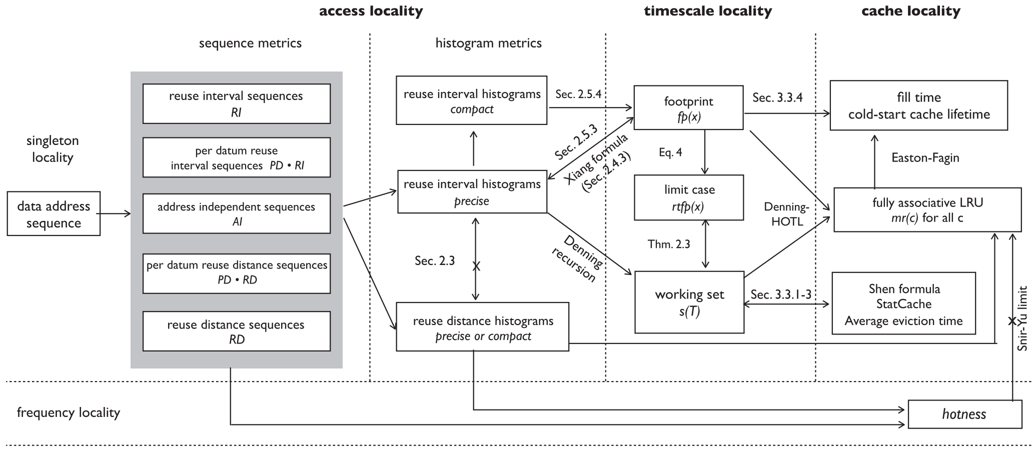
access locality
timescale locality
cache locality
sequence metrics
histogram metrics
singleton locality
data address sequence
reuse interval sequences RI
per datum reuse interval sequences PD • RI
address independent sequences AI
per datum reuse distance sequences PD • RD
reuse distance sequences RD
reuse interval histograms compact
reuse interval histograms precise
reuse distance histograms precise or compact
Sec. 2.5.4
Sec. 2.5.3
Xiang formula (Sec. 2.4.3)
Sec. 2.3
Denning recursion
footprint fp(x)
Eq. 4
limit case rtfp(x)
Thm. 2.3
working set s(T)
Sec. 3.3.4
Denning-HOTL
Sec. 3.3.1-3
fill time cold-start cache lifetime
Easton-Fagin
fully associative LRU mr(c) for all c
Shen formula StatCache Average eviction time
Snir-Yu limit
frequency locality
hotness

Fig. 4. The graph shows the relational theory as the conversion between locality metrics. For relations between sequence locality definitions (those in the gray box), see Figure 2. New contributions in this article are shown by the edges marked with a section, equation, algorithm, or theorem number.

Table 3. Comparison of Locality Categories

Category	Information stored	Strength/weakess in modeling cache
Sequence	ordered accesses	no compaction, RD → MRC, RI-RD equivalence
Histogram	unordered RI/RD	compactness, RD histogram →MRC, RI-RD nonequivalence
Timescale	unordered RI	compactness, RI histogram→MRC, composability

series of directed edges form a path. The transitive relation gives the conversion or its impossibility between every pair of metrics.

The locality metrics are grouped by categories, which are areas separated by dotted lines. Timescale locality is centrally connected: It is the hub that connects histogram and cache metrics and, through them, all other metrics.

All the metrics in the relation graph are from existing work. The contribution of the preceding sections is the connection of these metrics: In particular, the conversion and non-equivalence results that are required for all-to-all relations and were absent from past work.

Table 3 compares three categories of locality with respect to information retained and practical implications. Sequence locality stores full ordering information, while histogram and timescale locality ignore the ordering among reuses. To compute the miss-ratio curve (MRC), histogram locality needs reuse distances, while timescale locality needs just reuse intervals. There is a significant gain of space efficiency from sequence to histogram locality and of time efficiency from histogram to timescale locality. In addition, timescale locality is composable, as discussed in Section 2.5.5.

3.2 Usefulness in Practice

The relational theory helps to solve problems in practice. The first is precision. All metrics in the relation graph are defined by mathematics or algorithms, based entirely on information extracted from a reference trace, i.e., the singleton locality. Mathematics is not just precise but maintains the precision after many steps of derivation. Furthermore, it proves results for *all* programs, which are therefore universal.

The second is brevity and completeness. A metric may be computed in different ways, and this is shown by multiple paths from singleton locality. Every derivation between two locality concepts is represented by a path in the graph.

The third is modularity. A path decomposes a complex construction into single steps each represented by an edge. When there are multiple paths to derive the same metric, their overlap shows shared intermediate concepts and steps. These combinatorial choices are fully expressed without having to be enumerated.

The fourth is integration. Researchers can use multiple metrics when solving a problem. As the example in Section 2.6 shows, it is often convenient to formulate a problem using one metric and solution in another. The relational theory gives researchers the convenience in mixing these concepts in practice. Its equivalence and conversion results provide safe bridges, and non-equivalence results mark the boundaries and limitations.

As an example, consider the Denning recursion. As shown in Section 2.4.1, it computes the working set size iteratively by adding the miss ratio at each timescale. The time and space cost is linear. From the formula itself, it is unclear how it can be computed in a logarithmic cost using a compact histogram. Using the relation graph in Figure 4, we now see that this problem is easily solved. First, the footprint can be computed using a compact histogram. Second, it can derive the limit-case footprint. Finally, the limit-case footprint can compute the result of the Denning recursion. In the next section, we will show that a set of techniques are equivalent to the Denning recursion. Similar reasoning using the relation graph would show that they, too, can be computed using a compact histogram.

3.3 Formal Relations with Past Techniques

This section introduces four past techniques. The first three measure the miss ratio curve by using various types of sampling. This section focuses on how they compute the miss ratio. Using the new theory, we show that four past techniques, although independently developed with different areas of applications, produce results mathematically related to previous locality definitions and hence to each other.

3.3.1 The Shen Formula. Shen was the main inventor of a formula that converts from reuse interval to reuse distance statistically (2007). Given the reuse interval histogram, the Shen formula predicts the most likely reuse distance histogram. The conversion was 99% accurate and used by the open-source programming tool SLO (Beyls and D'Hollander 2006) and by a new tool called RDX that uses samples collected by hardware counters (Wang et al. 2019).

The formula is derived based statistical inference. The derivation is long and complex. Based on the article, it is difficult to understand why the formula should be such and why it is accurate in experiments. The authors actually admitted in the paper that their "formula is hard to interpret intuitively."

The key invention in the Shen formula is $p(w)$, which is "the probability of any given data element to appear in a time interval of length" w and is computed as follows from the reuse interval histogram:

$$p(w) = \sum_{i=1}^{w} \sum_{j=i+1}^{n-1} \frac{rt(j)}{m-1}.$$

If we take the difference $p(w+1) - p(w)$, then we see that it is equivalent to the Denning-Schwartz formula divided by $m-1$:

$$p(w+1) - p(w) = \sum_{i=w+2}^{n} \frac{rt(i)}{m-1}.$$

From Section 2.4.1, the probability is equivalent to $p(w) = s(w)/(m-1)$. It now has a clear meaning, which is that the probability an access in a window adds a distinct data item (to reuse distance) is the ratio of the working set size divided by the data size minus one. The reason for $m-1$ is to model a reuse window, where the reused datum cannot be accessed inside the window.

We can analyze the properties of the Shen formula using the new theory. For example, mathematically the probability $p(w)$ may exceed 1, because $s(w)$ is not bounded by m. The formula may use the footprint, i.e., setting $p(w) = fp(w)/(m-1)$, to avoid this problem.

3.3.2 Statcache. In 2010, Eklov and Hagersten developed Statcache and showed that it was highly accurate (98%) for computer-architecture evaluation. Statcache estimates the average reuse distance $ES(r)$ of all the accesses with the same reuse interval r. Eklov and Hagersten (2010) defined F_j as the fraction of all memory references with a reuse interval greater than j and computed the average reuse distance $ES(r)$ using the following formula:

$$ES(r) = \sum_{j=1}^{r} F_j = \sum_{j=1}^{r} \sum_{i=j+1}^{n} rt(i).$$

The purpose and the method of Statcache are similar to Shen. While the basic formula is identical to Denning-Schwartz, Statcache also developed extremely fast measurement through a novel type of random sampling (Eklov and Hagersten 2010). The subsequent application of Statstack won a Best Paper award a year later for its efficiency and accuracy (Eklov et al. 2011).

3.3.3 Cache Fill Times and Eviction Times. In the higher-order theory of locality (HOTL), Xiang et al. (2013) defined the *cache fill time*, which we denote as $ft(c)$, as the average amount of time for a program to access an amount of data equal to a cache size c, i.e., the time for an empty cache to incur c misses. They defined it as the inverse function of the footprint. Mathematically, the fill time and the footprint are opposite mappings between time and space, more specifically, between a timescale and a data (cache) size. The fill time of the footprint of timescale x is x, i.e., $ft(fp(x)) = x$. Equivalently, the footprint of the fill time of the cache size c is c, i.e., $fp(ft(c)) = c$.

Intuitively, the cache fill time is the time a program takes to fill the cache with recently accessed data. Any data block previously accessed is evicted after the cache fill time. As the cache is larger, the fill time longer, and the chance of a miss lower. Formally, an access is a miss if and only if its reuse interval is greater than or equal to the cache fill time, and the miss ratio can be computed by:

$$mr(c) = P(ri > ft(c)), \tag{10}$$

where c is the cache size, $ft(c)$ the fill time, and $P(ri)$ the distribution of reuse intervals.

Hu et al. (2018) defined an *eviction time* as the time between the last access of a data block and its eviction from the cache, and the average of all eviction times is the *average eviction time (AET)*, defined for each cache size c as $AET(c)$. The miss ratio is computed as the portion of accesses whose reuse interval is greater than the AET of the cache,

$$mr(c) = P(ri > AET(c)), \tag{11}$$

where c is the cache size, $AET(c)$ its AET, and $P(ri)$ the distribution of reuse intervals.

The fill time and the average eviction time are mathematically different. The first is based on footprint. AET is equivalent to Denning-Schwartz as shown by Hu et al. (2018). Therefore, the difference between the two has been analyzed previously in Section 2.4.4. In particular, from the WS-RTFP relation, if the fill time is computed from the limit-case Xiang formula as the inverse function of $rtfp(x) - rtfp(0)$, it is equivalent to the average eviction time. Because of its monotonicity and concavity, it implies that the inverse of the limit-case footprint is unique.

The fill time is mainly a metric for explaining the footprint theory. AET has been used in managing and optimizing the memory and storage cache (Byrne et al. 2018; Chen et al. 2018; Hu et al. 2018; Xiang et al. 2018). For static analysis, Chen et al. (2018) found it more convenient to use AET, because it required only the reuse interval, not the first- and last-access times.

3.3.4 *The Easton-Fagin Recipe.*

Easton and Fagin (1978) were among the first to study cache sharing, in particular, the effect from context switching. They defined a "cold-start cache" as one when a program is switched back and its earlier data have been all wiped out, and, to distinguish from it, a "warm-start cache" is used to refer to a regular, solo-use cache.

The cold-start miss ratio was difficult to simulate, because a program may be interrupted and then restarted at any point. The warm-start cache had no breaks in execution and was easy to simulate. Section 5 of the 1978 paper gave an ingenious solution that computes the cold-start miss ratio from the warm-start miss ratio.

The solution computes $LIFE_c^*(c)$, the time to have c misses in the cold-start cache of size c as the sum of the inter-miss time of all warm-start caches of size $0 \leq j < c$, i.e., $LIFE_c^*(c) = \sum_{j=0}^{c-1} LIFE(j)$ (Easton and Fagin 1978, Sec. 5). It expresses a simple but compelling intuition—the time from the jth miss to the next in the cold-start cache is probably similar to the inter-miss time of the warm-start cache of size j. Initially, the first access to the cold-start cache is the first miss, which is the inter-miss time of zero-size warm-start cache, $LIFE(0) = 1$.

The intuition seemed correct. Easton and Fagin found that the "estimate was almost always within 10–15 percent of the directly observed average cold-start miss ratio." They "gave a rough explanation as to why our recipe is reasonable" but "remark without proof that this need not be the case." They called the formula a recipe rather than a model.

With the relational theory, we can now derive the recipe. First, we see that $LIFE_c^*(c) = ft(c)$ and $LIFE(j) = im(j)$. Therefore, the recipe says that the fill time $ft(c)$ can be computed from the inter-miss times of the cache of all smaller sizes j or, formally, $ft(c) \approx \sum_{j=0}^{c-1} im(j)$.

We rewrite the fill time as the sum of c inter-miss times,

$$ft(c) = \sum_{j=0}^{c-1} (ft(j+1) - ft(j)) \approx \sum_{j=0}^{c-1} im(j).$$

We see that the key assumption of the recipe is that the time between the jth miss and the next is approximately the inter-miss time of cache size j. We can rewrite the inter-miss time as the inverse of the miss ratio and compute it using the Denning-HOTL conversion. In addition, we have $fp(ft(j+1)) - fp(ft(j)) = j + 1 - j = 1$. The approximation is therefore as follows:

$$ft(j+1) - ft(j) \approx im(j) = \frac{1}{mr(j)} = \frac{fp(ft(j+1)) - fp(ft(j))}{fp(ft(j)+1) - fp(ft(j))}.$$

Examining both sides of $\approx$, we see that the change in the fill time function is approximated by the change in the footprint function. It assumes a linear relation between the two changes. Mathematically, the approximation is equivalent to $f(x + \Delta) - f(x) \approx (f(x+1) - f(x))\Delta$, where $\Delta = ft(j+1) - ft(j)$ and $f(x)$ is $fp(ft(j))$. The change in ft, i.e., Δ, is proportional to the change in fp, i.e., $fp(x+1) - fp(x)$.

4 RELATED WORK

We review more related work in more detail in the following areas.

Observational Stochastics. Denning and Buzen (1978) formulated a new theory of queueing analysis now called *observational stochastics*. Conventional analysis was based on classic queuing models with idealistic assumptions such as infinite stationary processes. Observational stochastics are based on directly measurable variables and directly verifiable assumptions. The theory and applications in system and network analysis are enunciated in two recent books (Buzen 2015; Denning and Martell 2015). All locality definitions and properties in this article are based on direct measurements, do not depend on idealistic assumptions, and hence are extensions of observational stochastics.

The original timescale definition of locality is the Denning recursion, which can be derived using stochastic assumptions—that a trace is infinite and generated by a stationary Markov process, i.e., a limit value exists (Denning and Schwartz 1972). In later work, Denning and his colleagues adopted observational stochastics and used the formula on finite-length traces, with adjustments to account for boundary effects (Denning and Slutz 1978; Slutz and Traiger 1974). The footprint is also a type of observational stochastics. The equivalence theorem in Section 2.4.4 shows the mathematical relation between the two.

Cache Benchmark Synthesis. Benchmark synthesis is the construction of a synthetic program with desirable locality. It is locality metric conversion in the opposite direction to a trace. Synthesis has been used to solve two practical problems. The first is memory probing with parameterized locality to examine machine performance in multiple use scenarios. The probe program APEX-MAP can be configured to exhibit a distribution of reuse distances similar to a given target (Ibrahim and Strohmaier 2010). While APEX-MAP approximates, an algorithm by Shen and Shaw (2008) generates a trace that has the exact reuse distance histogram as specified. The second use cache behavior cloning. A system called WEST generates a stochastic trace based on the RD distribution within each cache set (Balakrishnan and Solihin 2012), while Hierarchical Reuse Distance (HRD) matches the RD distributions at multiple cache-line granularities or cache levels (Maeda et al. 2017).

Sampling. A real-world application usually generates a memory address trace of an extremely large length. Existing locality models (Hu et al. 2016; Wires et al. 2014) often employ sampling techniques to reduce the profiling overhead. There exist many sampling techniques: address sampling using hardware counters (Tam et al. 2009; Wang et al. 2019), random sampling (Eklov and Hagersten 2010; Hu et al. 2018), reservoir sampling (Beyls and D'Hollander 2006), and static sampling (Chen et al. 2018). Overall, sampling is a technique orthogonal to the definition of locality used in the analysis. The use of the new theory in sampling is beyond the scope of this article.

5 SUMMARY

This article has formalized major definitions of locality, grouped them into six categories, and showed a series of relations and properties, including the equivalence between sequence locality definitions, non-equivalence between histogram metrics, the equivalence between two timescale definitions, a formal justification of the Easton-Fagin recipe, the first solution that computes the footprint in linear time from either a precise or a compact histogram, and, from these results, a complete relational theory of locality.

APPENDIX

Here we present Algorithm 3, a more detailed version of Algorithm 1 for which we have a proof of correctness. Proof of Theorem 2.3.

Proof. The first appearance of each data item is placed correctly. Later appearances of each data item are also placed correctly, because the calculation of *nextpos* converts from reuse distance to reuse interval. After an access of e, $nextpos[e]$ is set initially by the reuse distance. Whenever there is a reuse, $nextpos[e]$ increments (Line 13). At the next access of e, $nextpos[e]$ equals the last

ALGORITHM 3: PD·RD → AI conversion (vector based)

```
1  $lastpos[1 \dots m] \leftarrow pd[1 \dots m][1]$
2  $nextpos[1 \dots m] \leftarrow pd[1 \dots m][1]$
3  $cnt[1 \dots m] \leftarrow \{1\}$
4  for $i = 1$ to $n$ do
5      $e \leftarrow 0$
6      for $e' = 1$ to $m$ do
7          if $nextpos[e'] = i$ && $(e = 0 \mid\mid lastpos[e] < lastpos[e'])$ then
8              $e \leftarrow e'$
9          end
10     end
11     for $e' = 1$ to $m$ do
12         if $lastpos[e'] < lastpos[e]$ then
13             $nextpos[e'] \leftarrow nextpos[e'] + 1$
14         end
15     end
16     $ai[i] \leftarrow e$
17     $cnt[e] \leftarrow cnt[e] + 1$
18     $lastpos[e] \leftarrow i$
19     $nextpos[e] \leftarrow i + pd[e][cnt[e]]$
20 end
```

access time plus the reuse interval. Therefore, all later occurrences of data elements are placed correctly. □

Algorithm 3 gives the conversion PD · RD → AI. The main loop of Algorithm 3, starting at Line 4, constructs the AI trace $ai[1 \dots n]$ by selecting the datum e accessed at each time i. Lines 1 to 3 initialize the auxiliary data: The last access time $lastpos[e]$ is the time of e's last access before i, $nextpos[e]$ the estimated time of its next access, and the access count $cnt[e]$ the number of times e has been accessed. Initially, for each datum e, the first access is f_e, and its access count $cnt[e] = 1$.

The main loop has two inner loops: the selection loop and the update loop. The selection loop, Lines 6–10, chooses e for $ai[i]$ if its estimated next access time is i. There may be multiple choices. The selection loop does not stop at the first such datum. It finds every such item and chooses the one with the largest last access time. This is a choice based on recency, i.e., most recent last access. Naturally, this choice is unique.

The recency choice at line 7 is necessary. Consider the AI trace $(e_1, e_2, e_3, e_2, e_1)$. When time $i = 4$, the next access times of e_1, e_2 are both estimated as 4. The selection loop must choose e_2, which is more recently accessed.

The update loop is the second inner loop. Lines 11–15 update $nextpos$ for all other elements e'. If e has been accessed after the last e', then the e access is a recurrence, so the estimated next access time of e' is increased by 1. Then, Lines 16–19 update for e: the current access is now the last access, the access count $cnt[e]$ is increased by 1, and the next access time is estimated to be the current time plus the next reuse distance $pd[e][cnt[e]]$.

ACKNOWLEDGMENTS

The authors thank the referees and staff of the journal, especially Referee 2, who suggested Algorithm 1 as a more intuitive expression of Algorithm 3. The article has been improved thanks to the comments by Noah Bertram, Jacob Brock, Dong Chen, Joel Kottas, Yechen Li, Rahman Lavaee, Hao Luo, Pengcheng Li, Colin Pronovost, and Chencheng Ye.

REFERENCES

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

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

Ganesh Balakrishnan and Yan Solihin. 2012. WEST: Cloning data cache behavior using stochastic traces. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 387–398. DOI: https://doi.org/10.1109/HPCA.2012.6169042

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*, Lecture Notes in Computer Science, Vol. 4208. Springer, 220–229.

Jacob Brock, Chen Ding, Rahman Lavaee, Fangzhou Liu, and Liang Yuan. 2018. Prediction and bounds on shared cache demand from memory access interleaving. In *Proceedings of the International Symposium on Memory Management*. 96–108. DOI: https://doi.org/10.1145/3210563.3210565

Jeffrey P. Buzen. 2015. *Rethinking Randomness: A New Foundation for Stochastic Modeling*.

Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: Miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the International Symposium on Memory Management*. 84–95. DOI: https://doi.org/10.1145/3210563.3210571

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 340–351.

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

Xi E. Chen and Tor M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 329–340.

Trishul M. Chilimbi, Bob Davidson, and James R. Larus. 1999. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 13–24.

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

Keith Cooper and Linda Torczon. 2010. *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

Peter J. Denning. 1968. The working set model for program behaviour. *Commun. ACM* 11, 5 (1968), 323–333.

Peter J. Denning and Jeffrey P. Buzen. 1978. The operational analysis of queueing network models. *Comput. Surv.* 10, 3 (1978), 225–261. DOI: https://doi.org/10.1145/356733.356735

Peter J. Denning and Kevin C. Kahn. 1975. A study of program locality and lifetime functions. In *Proceedings of the ACM Symposium on Operating System Principles*. 207–216.

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

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

Peter J. Denning and Donald R. Slutz. 1978. Generalized working sets for segment reference strings. *Commun. ACM* 21, 9 (1978), 750–759.

C. Ding and K. Kennedy. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.* 64, 1 (2004), 108–134.

Chen Ding, Xiaoya Xiang, Bin Bao, Hao Luo, Ying-Wei Luo, and Xiao-lin Wang. 2014. Performance metrics and models for shared cache. *J. Comput. Sci. Technol.* 29, 4 (2014), 692–712. DOI: https://doi.org/10.1007/s11390-014-1460-7

Malcolm C. Easton and Ronald Fagin. 1978. Cold-start vs. Warm-start miss ratios. *Commun. ACM* 21, 10 (1978), 866–872.

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.

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

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

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.

Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding. 2009. A component model of spatial locality. In *Proceedings of the International Symposium on Memory Management*. 99–108.

Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. 2013. Locality principle revisited: A probability-based quantitative approach. *J. Parallel Distrib. Comput.* 73, 7 (2013), 1011–1027. DOI: https://doi.org/10.1016/j.jpdc.2013.01.010

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*.

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

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 Trans. Stor.* 14, 2 (2018), 12:1–12:34. DOI: https://doi.org/10.1145/3185751

Khaled Z. Ibrahim and Erich Strohmaier. 2010. Characterizing the relation between Apex-Map synthetic probes and reuse distance distributions. In *Proceedings of the International Conference on Parallel Processing*. 0 (2010), 353–362. DOI: https://doi.org/10.1109/ICPP.2010.43

Bruce Jacob, Spencer W. Ng, and David T. Wang. 2010. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

S. Jiang and X. Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

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

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

Yumeng (Lucinda) Liu, Daniel Busaba, Chen Ding, and Daniel Gildea. 2018. All timescale window co-occurrence: Efficient analysis and a possible use. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. 289–292. http://dl.acm.org/citation.cfm?id=3291291.3291322.

Li Lu and Michael L. Scott. 2011. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the International Conference on Distributed Computing*. 460–474.

Hao Luo, Guoyang Chen, Fangzhou Liu, Pengcheng Li, Chen Ding, and Xipeng Shen. 2018. Footprint modeling of cache associativity and granularity. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'18)*. 232–242. DOI: https://doi.org/10.1145/3240302.3240419

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

Rafael K. V. Maeda, Qiong Cai, Jiang Xu, Zhe Wang, and Zhongyuan Tian. 2017. Fast and accurate exploration of multi-level caches using hierarchical reuse distance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 145–156. DOI: https://doi.org/10.1109/HPCA.2017.11

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.

R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.

Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn (Eds.). 2003. *Algorithms for Memory Hierarchies, Advanced Lectures*. Lecture Notes in Computer Science, Vol. 2625. Springer.

Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri E. Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

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*.

Apan Qasem and Ken Kennedy. 2005. *Evaluating a Model for Cache Conflict Miss Prediction*. Technical Report CS-TR05-457. Rice University.

S. Rubin, R. Bodik, and T. Chilimbi. 2002. An efficient profile-analysis framework for data layout optimizations. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon.

Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 53–64.

Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 279–292.

Xipeng Shen and Jonathan Shaw. 2008. Scalable implementation of efficient locality approximation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*. 202–216.

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.

Donald R. Slutz and Irving L. Traiger. 1974. A note on the calculation working set size. *Commun. ACM* 17, 10 (1974), 563–565. DOI: https://doi.org/10.1145/355620.361167

A. J. Smith. 1976. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the International Conference on Software Engineering*.

M. Snir and J. Yu. 2005. *On the Theory of Spatial and Temporal Locality*. Technical Report DCS-R-2005-2564. Computer Science Deptartment, University of Illinois at Urbana–Champaign.

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.

David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–132.

Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight reuse-distance measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 440–453.

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

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

Meng-Ju Wu and Donald Yeung. 2013. Efficient reuse distance analysis of multicore scaling for loop-based parallel programs. *ACM Trans. Comput. Syst.* 31, 1 (2013), 1. DOI: https://doi.org/10.1145/2427631.2427632

Meng-Ju Wu and Donald Yeung. 2011. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 264–275.

Meng-Ju Wu, Minshu Zhao, and Donald Yeung. 2013. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of the International Symposium on Computer Architecture*. 499–510.

Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. 2011a. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 91–102.

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

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.

Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic cache allocation with partial sharing. In *Proceedings of the EuroSys Conference*. 13:1–13:15. DOI: https://doi.org/10.1145/3190508.3190511

Chencheng Ye, Chen Ding, Hao Luo, Jacob Brock, Dong Chen, and Hai Jin. 2017. Cache exclusivity and sharing: Theory and optimization. *ACM Trans. Arch. Code Optimiz.* 14, 4, 34:1–34:26. DOI: https://doi.org/10.1145/3134437

Liang Yuan, Wesley Smith, Chen Ding, Sicong Fan, Zixu Chen, and Yunquan Zhang. 2018. Footmark: A new formulation for working set statistics. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*.

Chengliang Zhang, Chen Ding, Mitsunori Ogihara, Yutao Zhong, and Youfeng Wu. 2006. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 16–29.

Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. 2007. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Comput.* 56, 3 (Mar. 2007), 328–343.

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.* 31, 6 (Aug. 2009), 1–39.

Received August 2018; revised June 2019; accepted June 2019