Ding C, Xiang X, Bao B *et al.* Performance metrics and models for shared cache. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(4): 692–712 July 2014. DOI 10.1007/s11390-014-1460-7

Performance Metrics and Models for Shared Cache

Chen Ding[1] (丁　晨), Xiaoya Xiang[1] (向晓娅), Bin Bao[1] (包　斌), Hao Luo[1] (罗　昊), Ying-Wei Luo[2] (罗英伟) and Xiao-Lin Wang[2] (汪小林)

[1]*Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, U.S.A.*

[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

E-mail: cding@cs.rochester.edu; {sappleing, bin.bao}@gmail.com; hluo@cs.rochester.edu; {lyw, wxl}@pku.edu.cn

Received March 1, 2014; revised May 14, 2014.

Abstract Performance metrics and models are prerequisites for scientific understanding and optimization. This paper introduces a new footprint-based theory and reviews the research in the past four decades leading to the new theory. The review groups the past work into metrics and their models in particular those of the reuse distance, metrics conversion, models of shared cache, performance and optimization, and other related techniques.

Keywords memory performance metric, cache sharing, reuse distance

1 Introduction

Computing is ubiquitous in science, engineering, business, and everyday life. Most of today's applications, whether for cloud, desktop, or handheld, run on multicore processors. As a result, they interact with peer programs. It is beneficial to minimize the negative interaction. The benefit is important not just for good performance but also for stable performance, not just for parallel code but also for sequential applications running in parallel.

This paper surveys the theories and techniques to measure and improve program interaction on multicore processors. A program is either a sequential application or a parallel application being treated as a single party in interaction. Here we assume that programs do not share data or computation, but they share the hardware host. We call it a *solo-run* if a program runs by itself on a machine and a *co-run* if multiple programs run in parallel.

Cache sharing is a primary cause of co-run interference. Modern applications take most of their time to access memory, and most memory accesses — over 99% typically — happen in cache. A commodity system today has 2 to 8 processors (sockets), 2 to 6 physical cores per processor, and 2 to 4 hyperthreaded logical cores per physical core. Nearly a hundred programs can run together in parallel.

Partitioned cache solves the interference problem via program isolation. However, cache partitioning is wasteful when only one program is running and inefficient when co-run programs share data. Current multicore processors use a mix of private and shared cache. For example, Intel Nehalem has 256 K L2 cache per core and 4 MB to 8 MB L3 cache shared by all cores. IBM Power 7 has 8 cores, with 256 KB L2 cache per core and 32 MB L3 shared by all cores.

Depending on which CPU they are using, programs interact in different ways. Physical cores have private caches at the first and second levels but share the last level cache. Logical cores share the caches at all levels. Different processors do not share the caches. However, they share the memory bandwidth, and the demand of memory bandwidth depends entirely on the performance of the cache. In addition, some caching policies, e.g., inclusive cache on Intel machines, may induce indirect interaction, where a program may lose data in its private cache due to the data access by another program in the shared cache.

The advent of cache sharing the 2000s is reminiscent of the middle 1960s when time sharing was invented. Since then, the problem of memory management has been well studied and solved, and modern operating systems routinely manage memory for a large number of programs. However, the problem of cache sharing is more complex.

Survey

The work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61232008, the NSFC Joint Research Fund for Overseas Chinese Scholars and Scholars in Hong Kong and Macao under Grant No. 61328201, the National Science Foundation of USA under Contract Nos. CNS-1319617, CCF-1116104, CCF-0963759, an IBM CAS Faculty Fellowship and a research grant from Huawei. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding organizations.

Xiang has graduated and is now working at Twitter Inc. Bao has graduated and is now working at Qualcomm Inc.

Cache is managed by hardware, not the operating system. Cache has multiple levels and varying mixes of exclusivity and sharing. Events of cache accesses and replacements are orders of magnitude more frequent than memory access and paging. A single program may access cache a billion times a second and can wipe out the entire content of the cache in less than a millisecond. The intensity multiplies when more programs are run in parallel. Furthermore, the size of cache is fixed on a given machine. One cannot get online and buy more cache as one can with memory.

Cache interference is asymmetrical, non-linear, and circular. The asymmetry was shown experimentally by Zhang *et al.*[1] at Rochester and confirmed by later studies. In a pair-run experiment we conducted using Zhang's setup. One program becomes 85% slower, while its partner is only 15% slower. The interference changes from program to program. The effect depends not as much on how many programs are running as on which programs are running. Finally, the effect is circular. As a program affects its peers, it is also affected by them.

The solution to these problems requires a special theory called the theory of locality. Locality is a basic property of a computing system. Denning[2] defined locality as "a concept that a program favors a subset of its segments during extended intervals (phases)." There is a difference between the data that a program has and the data that the program is actively using. The "active" data is a subset, which Denning[3] called *the working set*.

Performance depends on how fast a computer system provides access to the active data subset. The access time of the other data is irrelevant. Locality analysis is therefore a prerequisite to memory design, for the oft quoted reason "we cannot improve what we cannot measure." In this article, we review the metrics for measuring and techniques for improving performance in shared cache.

2 Footprint Theory of Locality

2.1 Footprint

As a locality metric, the *footprint* measures the amount of active data usage. Given a program execution, we extract the data accesses as a linear sequence of memory addresses or object IDs. The sequence is called an access trace or an address string. A window is a sub-sequence of consecutive accesses. The length of a window is measured by time, either logically based on the number of accesses in the window or physically based on the time when the first and the last accesses were made.

Given a window, the footprint is the amount of data accessed in the window, i.e., the size of the "active" data. For an execution, the footprint is defined for each window length as the *average* footprint of all windows of that length. In a dynamic execution, the data usage may change in different length windows and in different windows of the same length. The footprint shows the change over all window lengths. For each length, it shows the average footprint, which is a single, unique value.

For example, consider three data blocks a, b, c. Fig.1 shows two patterns of data accesses. One has a stack access pattern, where the data block last accessed is first reused. The other has a streaming pattern, where the blocks are traversed in the same order. The footprints are shown for all length-3 windows, four in each trace. The footprint of a trace is the average. For length-3 windows, the footprint, $fp(3)$, is 2.5 in the stack trace and 3 in the streaming trace. Therefore, the streaming access has a greater data activity for that window length. The complete footprint is defined for all window lengths and would count in the amount of data access in all windows of all lengths.

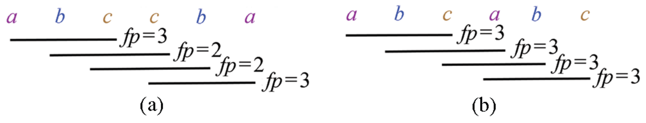


Fig.1. Amount of data accessed in length-3 windows in two access traces: (a) stack accesses and (b) streaming accesses. The footprint of a trace is the average amount. It is defined for each window length. When the length is 3, the footprint, $fp(3)$, is 2.5 in the stack trace and 3 in the streaming trace.

In practice, the footprint is too numerous to enumerate. The number of time windows is quadratic to the length of the trace①. Assuming a program running for 10 seconds on a 3 GHz processor, we have 3E10 CPU cycles in the execution and 4.5E20 distinct windows.

Brock *et al.*[4] described program analysis as a Big Data problem, and showed the scale of the problem by the number of time windows in an execution. Fig.2 shows that as the length of execution increases from 1 second to 1 month, the number of CPU cycles (n) ranges from 3E9 to 2E15, and the number of distinct execution windows $\binom{n}{2}$ from 4.5E18 to 5.8E29, that is, from 4 sextillion to over a half nonillion.

As a dynamic analysis problem, the scale quickly reaches the size of any static problem. As a comparison, the figure shows the radius of the Milky Way in centimeters, 48 sextillion, and the radius of the observable universe, 44 octillion.

①If the trace length is n, the number of windows (and hence footprints) is $\binom{n}{2} + n = \frac{n\times(n+1)}{2}$ or $O(n^2)$ asymptotically.

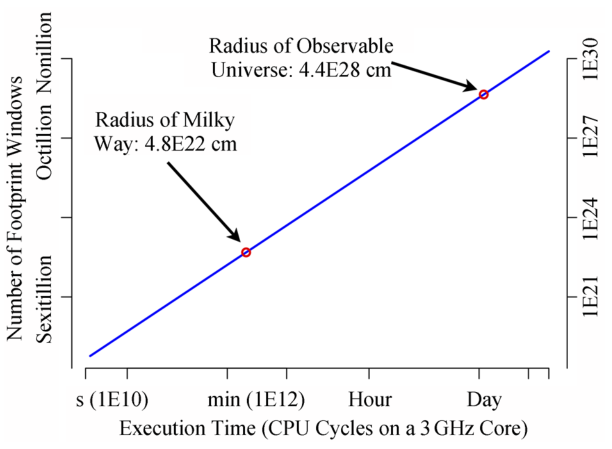


Fig.2. Scale of the problem shown by the number of footprint windows in a program execution, compared to the size of a galaxy and the universe. Reproduced from [4].

For system design, it may not be very useful to consider very large windows, since caching decisions are usually based on information on the recent execution. For programming, however, it is necessary to analyze the full execution to find opportunities of global optimization. This is shown by Zhong *et al.* in whole-program locality analysis, which analyzes the full length of reuse distances to see how it changes with the input[5], and in affinity-based data layout, which groups structure fields based on the distribution of long reuse distances[6].

The purpose of a footprint theory is to overcome the enormity of the analysis problem, characterize the active data usage in all windows, and make it useful for system analysis and optimization.

2.2 Footprint Theory

For locality analysis, the basic unit of information is a data access, and the basic relation is a data reuse. The theory of locality is concerned with the fundamental properties of data accesses and reuses, just as the graph theory is with nodes and their links.

The footprint theory consists of a set of formal definitions, algorithms, and properties based on the concept of the footprint. This subsection introduces the four components of the theory and their supporting techniques, based on the material published in a series of papers[7-12].

Footprint Measurement. The enormous scale of all-window analysis is tackled by a series of three algorithms. Each is two orders of magnitude more efficient than the previous one.

• Footprint distribution analysis, which enumerates all $O(n^2)$ footprints in $O(n \log m)$ time, where n is the length of the trace and m the maximal footprint.

• Average footprint analysis, which reduces the cost to linear time $O(n)$ by computing the average without enumerating all footprints.

• Footprint sampling, which samples limited-size windows and further reduces the cost.

The distribution analysis is the first algorithm to measure the all-window footprint. As it actually enumerates all footprints, it finds the largest, smallest, median, average, and any percentile footprint for each window length. However, the cost is sometimes thousands of times slowdown compared to the speed of the original program.

The second algorithm computes just the average footprint, and the cost is reduced from a thousand times slowdown to about 20 times. Being a linear time algorithm, it is scalable in that the cost increases proportionally to the length of the program execution.

The cache on a real machine has a finite size, so an analysis does not have to consider windows whose footprint is greater than the cache size. In addition, the behavior of a long running program tends to repeat itself. Furthermore, on modern processors, the analysis can be carried out on a separate core in parallel with the analyzed execution. Footprint sampling specializes and parallelizes the analysis for a specific machine and program. The average cost is reduced to 0.5% of the running time of the unmodified execution.

The algorithmic development attains immense gains in both computational complexity and implementation efficiency. As the baseline, the distribution analysis is the first viable solution for precise all-window analysis. The second and the third algorithm each improves efficiency by another order of magnitude, eventually making it fast enough for real-time analysis. This has a beneficial impact elsewhere, because the footprint can be used to compute other locality metrics, as we will see in the third part of the footprint theory.

Composability. A locality metric is *composable* if the metric of a co-run can be computed from the metric of solo-runs. If co-run programs do not share data, the footprint is composable. Let the average footprint of a program be $prog.fp(x)$ for window length x. If we have k programs $prog_1, prog_2, \ldots, prog_k$ actively sharing the cache, the aggregate footprint is the sum of the individual footprints.

$$corun.fp(x) = \sum_{i=1}^{k} prog_i.fp(x).$$

In comparison, the miss ratio is not composable. We will prove it later in Subsection 3.6.3. Intuitively, the co-run miss ratio will change compared to the solo-run miss ratio, since each program has now a fraction instead of the whole cache. The change in miss ratio, as mentioned earlier, is asymmetrical, non-linear, and affected by circular feedback. As a result, we cannot directly add the solo-run miss ratio to compute the co-run miss ratio, as we can with the footprint.

Another locality metric is reuse distance. Reuse distance does not depend on cache parameters, but as we will explain in Subsection 3.2, neither is it composable.

As mentioned earlier, we can measure the average as well as the distribution of footprints. The average footprint is immediately composable. The distribution, although composable, requires a convolution which is expensive to compute and difficult to visualize. In the following, the term "footprint" means the average footprint.

The next question is whether the footprint composability can help in analyzing the miss ratio and other locality metrics in shared cache. This is solved in the third part of the new theory.

Locality Metrics Conversion. Locality has different measurements, just like temperature can be measured in different scales, Celsius or Fahrenheit. For locality, the two most common metrics are miss ratio for hardware design and reuse distance for program optimization.

Central to a locality theory is the conversion between different metrics. The footprint theory shows that the footprint is convertible with a number of other metrics. Let $mr(c)$ be the miss ratio for cache size c. It can be computed from the footprint using the following formula[10]:

$$mr(c) = \frac{fp(x+\Delta x) - fp(x)}{\Delta x},$$

where $c = fp(x)$. If these are continuous functions, we would say that the miss ratio is the derivative of the footprint.

The higher order mathematics implies mathematical properties. Since the derived metric, the miss ratio, is non-decreasing, the source metric, the footprint, must be not just non-decreasing, but also concave. Indeed, the monotonicity and concavity were proved in two successive papers[9-10].

The conversion is reversible. If we have the miss ratios of all cache sizes, we can reverse the formula and compute the average footprint. The reverse process is the analog of integration for a discrete function.

Combining footprint composition and metrics conversion, we can see immediately that if the co-run miss ratio (miss ratio seen by the shared cache) can be computed from the aggregate footprint. Fig.3 shows the derivation by adding the individual footprints and then converting the sum into the co-run miss ratio.

Since the conversion formula is reversible, we can switch between the footprint and the miss ratio and compose the latter indirectly through the former. First, we compute the individual footprint from the individual miss ratios (of all cache sizes). Then we add the individual footprints and finally compute the co-run miss ratio in the shared cache (of all sizes). Fig.3 shows this type of deduction and others that are made possible by composition and conversion. In particular, the figure shows how to compose another locality metric, the reuse distance. We use the terms private reuse distance (PRD) and concurrent reuse distance (CRD), as introduced by [13-14].

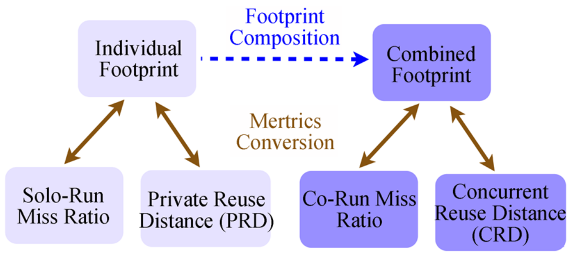


Fig.3. Joint use of two theoretical properties: composition (dotted line) and conversion (solid lines).

The solution of composition raises the problem of decomposition. The co-run miss ratio does not tell us the contribution from each program. To see the individual effects, we need more elaborate models.

Composable Locality Models. We say a model is composable if the co-run result can be computed from solo-run results, not just for the co-run group as a whole, but also for the co-run effect on each individual program. In other words, a composable model must be both composable and decomposable.

As a composable metric, the footprint has the following useful traits:

• *Machine Independent.* The analysis is based on data accesses, not cache misses. It takes a single pass to analyze a trace for all cache sizes, and the result is not affected by program instrumentation. In comparison, it is inescapable for direct measurement to be affected by instrumentation.

• *Clean-Room Statistics.* The footprint of one program can be measured in a co-run environment, unperturbed by other programs. The clean-room effect solves the chicken-or-egg problem of direct measurement: the behavior of one program depends on its peer, but the peer behavior in turn depends on itself.

• *Peer Independent.* The footprint of a program is independent of co-run peers. The analysis of cache sharing does not require actual cache sharing. The co-run effect is computed rather than measured.

• *Statically Composable.* There are 2^P co-run combinations for P programs. The footprint model can predict the interference in these 2^P runs by testing P single-program runs. For the P sequential runs, we can choose to run them one by one or some of them in parallel to increase speed. The composition is static if there is no actual co-run; otherwise we say the composition is dynamic. Here dynamic composition means parallel testing, while static composition does not need parallel testing at all.

To compute the co-run effect on each individual program, this dissertation describes three models. The models solve the decomposition problem as a composition problem: how one program is affected by its peers.

• *Composition by Reuse Distance and Footprint.* Variations of this model were invented by Thiebaut and Stone[15] and Suh *et al.*[16] for time-sharing systems (time-switched cache sharing) and Chandra *et al.*[17] for multicore (continuous cache sharing). These studies estimated the footprint since there was no feasible ways to measure it. After the invention of the fast measurement, the cost of the model became limited by the time required for reuse distance measurement[9-10].

• *Composition by Footprint Only.* The second model converts the footprint into reuse distance, so it no longer needs to measure the reuse distance and can be hundreds of times faster[10].

• *Composition by Program Pressure and Sensitivity.* The last model is as fast as the second model but more intuitive and easier to use. It characterizes the behavior of a program by two factors, pressure and sensitivity. The two can be visualized as two curves. Performance composition is as simple as looking up related values on the two curves[12].

The composable models provide answers to a number of long-standing questions about shared cache, including a machine independent way to compare programs by their shared cache behavior, the correlation between a program's cache interference and its miss ratio, and the performance of cache sharing compared with cache partitioning[12].

These models are theoretical, and they are appealing partly due to the generality. The footprint is defined on a program trace without knowing co-run peers or machine parameters (other than having shared cache). There are many sources of error due to the fact that the basic models do not consider the effect of cache associativity, program phase behavior, the time dilation due to interference, the filtering effect in a multi-level cache hierarchy, and the impact of the prefetcher. A theory must be validated to be practically relevant. The past studies have used experiments on real systems to evaluate the theoretical models and compare their predictions with actual miss counts measured by hardware counters[8-10,12]. They also showed extensions of the models to consider time dilation[8,12], cache associativity, and program phases[10].

3 Locality Theory from 1968

Locality was started as an observation that programs do not use all their data at all times[2]. After decades of research, it has been developed into an important scientific field. At its foundation are locality metrics, so the concept and its effect can be measured. Among the basic problems are the measurement speed and accuracy of these metrics.

3.1 Miss Ratio and Execution Time

The metric of miss ratio was first used by Belady[18] to find out how often individual policies caused page faults. It was challenging at that time to measure page traces and simulate the various policies on them. Today, the hardware performance counters on modern machines enable a tool to measure program speed and count cache misses in real time with little cost. The performance of a single program or a group of programs can be observed directly. However, direct observation has difficulties in characterizing the locality cleanly due to dependences on the observation environment. These dependences include:

• *Machine Dependence.* Different machines have different memory hierarchies and processors, so we cannot compare the locality in different programs entirely based on their performance.

• *Instrumentation Dependence.* The analysis code itself consumes processor and cache resources. It may not be possible to completely separate the effect of the instrumentation.

• *Peer Dependence.* It is unknown how the performance has changed due to cache sharing. It would have required another test on an unloaded system. It is also unknown how the performance will change if the peer programs change.

The effect of cache on performance is often disruptive. This phenomenon was first discussed by Denning[19] and stated as the *thrashing*, which happens when the sum of the working sets exceeds the available memory. Chilimbi once compared the phenomenon to strolling leisurely until suddenly falling over a cliff②. The danger is greater in a shared environment. As more programs are added, the combined working set grows. When it exceeds the size of the shared cache, sharp performance drops would ensue. Being peer and machine dependent, direct testing cannot foresee a pending calamity. What is worse, it cannot even tell whether a

②Trishul Chilimbi made this analogy in a presentation in 2002[20].

given parallel mix is efficient or not without testing them individually first.

3.2 Reuse Distance

The most common metric in program characterization is the reuse distance. For each memory access in a trace, the reuse distance is the number of distinct data elements accessed between this and the previous access to the same data. Mattson *et al.* first defined the concept (to model the performance of an LRU stack) and called it the LRU stack distance[21]. LRU is a cache management method that favors recently used data. Recognizing it as a measure of recency, Jiang and Zhang[22] called the metric the inter-reference recency (IRR).

For example, the reuse distance shows the locality of stack access and streaming access traces in Fig.4. When a block is first accessed, the reuse distance is infinite. When the block is reused, the reuse distance is the number of distinct blocks accessed between the previous access and the reuse. The reuse would miss in (fully associative LRU) cache if and only if its reuse distance is greater than the cache size. The figure shows that the stack trace can reuse the data in cache when the cache size is less than 3 but the streaming trace cannot.

Reuse	∞	∞	∞	1	2	3	∞	∞	∞	3	3	3
Distance	*a*	*b*	*c*	*c*	*b*	*a*	*a*	*b*	*c*	*a*	*b*	*c*

Fig.4. The locality of two traces, stack accesses on the left and streaming accesses on the right, measured by the reuse distance of each memory reference. An access is a miss in fully associative LRU cache if and only if its reuse distance is greater than the cache size.

The reuse distance quantifies the locality of every memory access. The locality of a program, or a loop or a function inside it, is the collection of all its reuse distances. The collective result can be represented as a distribution. It is called a *locality signature*[5] and *locality profile*[14].

3.2.1 Relation with Cache Performance

In the absence of cache sharing, the capacity miss ratio can be written as the fraction of the reuse distance that exceeds the cache size[21]. Let the test program be A and cache size be C; we have

$$P(\text{capacity miss by} A) = P(A\text{'s reuse distance} > C). \quad (1)$$

The reuse distance is machine independent but can give the capacity miss ratio for cache of all sizes, as the formula shows. The locality signature can be viewed as a discrete probability density function, showing the probability of a memory access having a certain degree of locality. The miss ratio is then the probability function, showing the probability of the access being a miss for a given cache size. A probabilistic adjustment invented by Smith can estimate the effect of cache conflicts in set-associative cache[23-25]. Combining the reuse distance and the Smith formula, we can compute the miss ratio in the cache of all sizes.

Miss Ratio Curve (*MRC*). The miss ratio curve (MRC) shows the miss ratio of all its cache sizes as a discrete function. It is easy to visualize and show directly the trade-off between performance and cache size. For fully associative LRU cache, the miss ratio curve is equivalent to the reuse distance distribution, as the preceding formula shows. The problem is equivalent in theory to the argument whether it is measuring the miss ratio curve or the reuse distance. In practice, the miss ratio curve is defined for only practical cache sizes, i.e., powers of two between some range, e.g., 32 KB and 8 MB. The reuse distance has the full range between 1 and the size of program data.

The full range of reuse distance represents the complete temporal locality. The miss ratio curve is a projection of the full information on a subset of cache sizes. The two would be equivalent if the miss ratio is defined for all cache sizes between 1 and infinity. The unbounded size of the representation is necessary, as shown by the theoretical result of Snir and Yu[26] that temporal locality cannot be fully encoded using a bounded number of bits. In the following, we review the prior work on both the reuse distance and the miss ratio curve.

3.2.2 Locality Analysis and Optimization

Reuse distance has found many uses. The locality signature shows how the cache behavior changes with the program input, and the changes can be predicted by whole-program locality analysis[5,25,27], which was used to predict the miss ratio of all inputs and cache sizes[28]. Fang *et al.* modeled locality signature for each memory reference and used it to find critical memory loads and important program paths[27,29]. Marin *et al.*[25] modeled the locality signature at reference, loop, and function levels to predict performance across different computer architectures. Beyls and D'Hollander[30-31] built a program tuning tool *SLO*, which identifies the cause of long distance reuses and gives improvement suggestions for restructuring the code. In addition to cache misses, reuse distance has been used to analyze the response time in server systems[32] and the usage pattern in web reference streams[33]. Zhong *et al.*[5] classified these and other uses of reuse distance as "Five Dimensions of Locality" and reviewed the analysis techniques for program input, code, data, execution phase, and program interaction.

Reuse distance provides a common foundation to model program behavior, predict machine performance, and guide program optimization. Locality analysis and profiling are to infer, measure, and decompose reuse distances, and locality optimization is to shorten long reuse distances. The analysis and the optimization are free of machine, instrumentation, and peer dependences. The downside, however, is the complexity of measuring reuse distance.

3.2.3 Direct Measurement

Reuse distance is one of the stack distances defined in the seminal paper in 1970 by Mattson *et al.*[21] The stack algorithm in the paper needs $O(nm)$ time to profile a trace with n accesses to m distinct data. The efficiency has been steadily improved over the past four decades. In 1975, Bennett and Kruskal[34] organized the trace as a tree and reduced the cost to $O(n \log n)$. In 1980, Olken[35] made the tree compact and reduced the cost further to $O(n \log m)$.

The Olken algorithm has been the most efficient asymptotic solution (for full reuse distance measurement) until 2003, when Ding and Zhong gave an approximation algorithm[5,36]. The approximation guarantees a relative precision, e.g., 99%, and takes $O(n \log \log m)$ time, which is effectively linear to n for any practical data size m. Zhong *et al.* also gave an algorithm that guarantees a constant error bound and does not reduce the asymptotic cost[37]. In an independent implementation, Schuff *et al.*[38] reported that the average cost of the $O(n \log \log m)$ method is as high as several thousand times slowdown.

Kim *et al.*[39] gave a linear-time algorithm to measure the miss ratio for a fixed number of cache sizes, which may be used to approximate reuse distance.

There are practical improvements to the Olken algorithm. *Cheetah* implemented the Olken algorithm using a splay-tree[40]. It became part of the widely used SimpleScalar simulator[41]. Almasi *et al.*[42] used a different tree representation to further improve the efficiency. A greater efficiency can be obtained through sampling and parallelization (Subsections 3.2.6 and 3.2.7).

Zhong *et al.* gave a lower bound result showing that the space cost of precise measurement is at least $O(n \log n)$, indicating that reuse distance is fundamentally a harder problem than streaming, i.e., counting the number of 1's in a sliding window, which can be done using $O(n)$ space[5].

3.2.4 Approximation by Reuse Time

While the reuse distance counts the number of distinct memory accesses, the reuse time counts all accesses. It is simply the difference in logical time between the previous access and the current reuse and can be measured quickly in $O(n)$ time. The working set theory uses the reuse time (inter-reference gap) to compute the time-window miss rate[43]. If we take time-window miss rate as an approximation of the LRU miss rate, we may say that the working set theory is the first approximation technique.

Two series of more recent studies have used the reuse time to compute the reuse distance. The first is StatCache and StatStack by Hagersten and his students[44-47],[③], and the second is time-based locality approximation[48-50]. For brevity, we name the latter technique after its lead author and call it the *Shen conversion*.

Berg and Hagersten solved the following equation for the miss ratio R[44]. Let N be the length of the trace, $h(t)$ be the number of accesses whose reuse time is t, and $f(k)$ be the probability that a cache block is replaced after k misses. The cache is assumed to have random replacement, so $f(k) = 1 - (1 - \frac{1}{C})^k$, for cache with C blocks. The total number of misses can be computed in two ways, and they should be equal:

$$NR = \sum_{t=1} Nh(t)ft(R).$$

StatCache solves the implicit equation for the miss ratio R using numerical methods.

In the Shen conversion[48,51], the key measure is the interval access probability $p(\Delta)$, which is the probability of a randomly chosen datum v being accessed during a time interval Δ. For a reuse at time distance Δ, below is the probability that its reuse distance is k:

$$p(k, \Delta) = \binom{N}{k} p(\Delta)^k (1 - p(\Delta))^{N-k}.$$

The formula computes the probability for k distinct data items to appear in a Δ-long interval. It assumes a binomial distribution given the interval access probability $p(\Delta)$, which is computed as

$$p(\Delta) = \sum_{t=1}^{\Delta} \sum_{\delta=t+1}^{T} \frac{1}{N-1} p_{T}(\delta), \qquad (2)$$

where $p_t = \frac{h(t)}{N}$ is the probability that an access has the time distance t. The derivation for $p(\Delta)$ can be found in a technical report[51].

[③]Berg and Hagersten used the term reuse distance for what we mean by reuse time[44].

The two statistical techniques are successful in predicting performance. StatCache was used to model first private cache[44-45] and then shared cache[46-47]. The Shen conversion was used first for sequential code[48-49] and then multi-threaded code[50,52].

Although both using statistical analysis, StatCache and the Shen conversion are fundamentally different: one models the random cache, and the other the LRU cache. Next we explore the difference between random and LRU modeling in greater depth.

3.2.5 *Random vs LRU*

Any statistical analysis of locality invariably makes some assumptions about randomness. We examine three such assumptions.

The first is random access to a cache set, which means that a data access can happen at any cache set with equal probability. The Smith formula uses the assumption to calculate the contention in a cache set and the effect of cache associativity[23].

The second is random cache replacement, which means that a miss may evict any cache block with equal probability. Under the assumption of random replacement, the lifetime of a block in cache is binomially distributed over the number of cache misses. Not knowing the miss rate, StatCache uses the relation to compute the miss rate from the reuse time[44]. Knowing the miss rate, West *et al.* computed the cache occupancy[53]. Fedorova *et al.* devised a fair scheduling policy based on the assumption that a set of applications divide the cache equally if they had the same miss rate[54].

Since real cache does not use random replacement, the accuracy of the assumption needs to be examined. For cache occupancy, West *et al.* compared the prediction with the actual measurement (through cache simulation) and found that the prediction is accurate for caches using random replacement but less so for caches using LRU[53].

Random has two other differences from LRU. One is well known, which is that the random replacement cache is fully associative by definition. The other is less recognized, which is that the cache performance is not deterministic as the replacement decisions change randomly every time a program is run. Fortunately, the problem is recently solved. Zhou gave an ingenious algorithm to compute the average miss ratio in a single pass, without having to simulate multiple times to compute the average[55].

The way to model LRU is using reuse distance. Knowing the reuse distance, the Smith formula uses it to model the LRU replacement within a cache set[23]. Not knowing the reuse distance, the Shen conversion needs a way to compute it[48,51]. It assumes a third type of randomness — in a time window, each data block is uniformly randomly accessed. By computing the reuse distance, the Shen conversion models LRU rather than random cache replacement.

Cache models can be divided by the replacement policy: LRU or random. There is a second dimension to compare them: the metrics used to measure window-based locality. For random replacement, we want to know the number of misses in a time window. There is a (trivial) linear relation between the miss count and the window length. For LRU, we want to know the footprint in a window. The relation is non-linear, and it is the main source of complexity in the Shen conversion in particular the derivation of the interval access probability.

Cache models use two types of window-based locality: the miss count and the footprint. The miss count is linear but cache size dependent. In comparison, the footprint is non-linear but cache size independent. For example, StatCache has to solve its equation for every cache size, while the Shen conversion produces the reuse distance and the miss ratio for all cache sizes. The past solutions represent different trade-offs between modeling simplicity and power. With the footprint theory, we have a new option, which is to compute the reuse distance using the footprint, which we can measure as accurately as we can with the miss count.

The three modeling methods, StatCache, the Shen conversion, and the footprint conversion, are not guaranteed to always give the correct reuse distance. Indeed, a precise linear-time solution is unlikely given the lower bound result in Zhong *et al.*[5] Among the three, the footprint theory is unique in formulating the condition for correctness, which is the reuse hypothesis[10].

3.2.6 *Sampling Analysis*

Sampling is usually effective to reduce the cost of profiling. Choosing a low sampling rate may reduce the amount of profiling work by factors of hundreds or thousands. In program analysis, bursty tracing is widely used, where the execution alternates between short sampling periods and long hibernation periods[20,56-57]. During hibernation, the execution happens in the original code and has no analysis overhead.

Locality sampling, however, is tricker. Locality is about the time of data reuse, but the time is unknown until the access actually happens. The uncertainty has two consequences. First, the length of the sampling period cannot be bounded if it is to cover a sampled data reuse pair. Second, the analyzer has to keep examining every data access. Complete hibernation is effectively impossible.

The problem of locality sampling is addressed by a series of studies, including the publicly available SLO tool[30], continuous program optimization[58], bursty reuse distance sampling[59], and multicore reuse distance analysis[38]. Sampling can drastically reduce the cost if sampled windows accurately reflect the behavior of other windows[45-47].

SLO has been developed by Beyls and D'Hollander[30]. It instruments a program to skip every k accesses and take the next address as a sample. A bounded number of samples are kept in a sample reservoir — hence the name reservoir sampling. To capture the reuse, SLO checks each access to see if it reuses some sample data in the reservoir. The instrumentation code is carefully engineered in GCC to have just two conditional statements for each memory access (one for address and the other for counter checking). Reservoir sampling reduces the time overhead from 1000-fold slow-down to only a factor of 5 and the space overhead to within 250 MB extra memory. The sampling accuracy is 90% with 95% confidence. The accuracy is measured in reuse time, not reuse distance or miss rate.

To accurately measure reuse distance, a record must be kept to count the number of distinct data that appeared in a reuse window. Zhong and Chang[59] developed the bursty reuse distance sampling, which divides a program execution into sampling and hibernation periods. In the sampling period, the counting uses a tree structure and costs $O(\log\log M)$ per access. If a reuse window extends beyond a sampling period into the subsequent hibernation period, counting uses a hash-table, which reduces the cost to $O(1)$ per access. Multicore reuse distance analysis by Schuff *et al.*[38] uses a similar scheme for analyzing multi-threaded code. Its fast mode improves over hibernation by omitting the hash-table access at times when no samples are being tracked. Both methods track reuse distance accurately.

StatCache by Berg and Hagersten[45] is based on unbiased uniform sampling. After a data sample is selected, StatCache puts the page under the OS protection (at page granularity) to capture the next access to the same datum. It uses the hardware counters to measure the time distance until the reuse. OS protection is limited by the page granularity. Two other systems, developed by Cascaval *et al.*[58] and Tam *et al.*[60], use the special support on IBM processors to trap accesses to specific data addresses. To reduce the cost, these methods use a small number of samples. Cascaval *et al.*[58] used the Hellinger Affinity Kernel to infer the accuracy of sampling. Tam *et al.*[60] predicted the miss rate curve in real time.

3.2.7 *Parallel Analysis*

Schuff *et al.*[38] combined sampling and parallel analysis for parallel code on multicore. At the IPDPS conference in 2012, three groups of researchers reported that they made the analysis of even sequential programs many times faster with parallel algorithms. Niu *et al.*[61] parallelized the analysis to run on a computer cluster, while Cui *et al.*[62] and Gupta *et al.*[63] parallelized it for GPU.

Unlike the reuse distance, the footprint can be easily sampled and analyzed in parallel using shadow profiling[64-65]. By measuring the footprint and converting it to reuse distance, we have shown the equivalent of parallel sampling analysis for reuse distance, which can be done in near real-time, with just 0.5% visible cost on average[10]. We note that the accuracy of footprint conversion is conditional[10], but direct (parallel) measurements are always accurate.

3.2.8 *Compiler Analysis*

Reuse distance can be analyzed statically for scientific code. Cascaval and Padua[66] used the dependence analysis[67], and Beyls and D'Hollander[68] defined *reuse distance equations* and used the Omega solver[69]. While they analyzed conventional loops, Chauhan and Shei[70] analyzed MATLAB scripts using dependence analysis. Unlike profiling whose results are usually input specific, static analysis can identify and model the effect of program parameters. Beyls and D'Hollander[68] used the reuse distance equations for cache hint insertion, in particular, conditional hints, where the caching decision is based on program run-time parameters. Shen *et al.*[71] used static and lightweight reuse analysis in the IBM compiler for array regrouping and structure splitting.

Using the static reuse distance analysis and the footprint theory, Bao and Ding demonstrated a compiler technique for analyzing the program footprint and discussed the potential use in peer-aware program optimization[72-73]. In [72], they used the tiled matrix multiply (Fig.5) as an example to show the reuse distance computed at the source level (Table 1). They also

```
for ($jj = 0$; $jj < N$; $jj = jj + B_j$)
  for ($kk = 0$; $kk < N$; $kk = kk + B_k$)
    for ($i = 0$; $i < N$; $i = i + 1$)
      for ($j = jj$; $j < min(jj + B_j, N)$; $j = j + 1$)
        for ($k = kk$; $k < min(kk + B_k, N)$; $k = k + 1$)
          $C[i][j] = beta \times C[i][j] + alpha \times A[i][k] \times B[k][j]$;
```

Fig.5. Loop nest of tiled matrix multiply.

Table 1. Reuse Distance as a Function of the Loop Bounds

Loop	Array	Reuse Distance (Bytes)
k	$C[i][j]$	8×3
j	$A[i][k]$	$8 \times 1 + 8 \times B_k + 8 \times B_k$
i	$B[k][j]$	$8 \times B_j + 8 \times B_k + 8 \times B_k \times B_j$
kk	$C[i][j]$	$8 \times N \times B_j + 8 \times N \times B_k + 8 \times B_k \times B_j$
jj	$A[i][k]$	$8 \times N \times B_j + 8 \times N \times N + 8 \times N \times B_j$

showed the use of the conversion theory (Subsection 2.2) to compute the miss ratio curve and a measure of shared-cache friendliness called the fill time.

3.2.9 Domain-Specific Modeling

To model graph algorithms, Yuan *et al.*[74] defined the notion *vertex distance* and used statistical analysis to derive the reuse distance. The study examines random graphs and scale-free graphs. It shows the dual benefits of domain-specific analysis. On the one hand, the structure of a graph facilitates locality analysis. On the other hand, locality analysis reveals the relation between the properties of a graph, e.g., edge density, and the efficiency of its computation.

3.2.10 Discussion

Reuse distance is a powerful tool for program analysis. It quantifies the locality of every program instruction. For a single sequential execution, the metric is composable. For example, the composition can happen structurally to show the locality of larger program units such as loops, functions, and the whole program, or it can happen temporally to show program executions as (integer valued) signals.

There are at least two limitations. First, reuse distance is insufficient to analyze program interaction. While programs interact at all times in the shared cache, reuse distance provides locality information for only reuse windows, not all windows. Second, precise reuse distance is still costly to measure. Despite all of the advances in sampling and parallelization, the asymptotic cost is still more than linear. These problems will be addressed indirectly through the study of another locality metric, the footprint.

3.3 Early Footprint

Measuring footprint requires counting distinct data elements, and the result depends on observation windows. The problem has long been studied in measuring various types of reuse distances as discussed before. However, footprint measurement is a more difficult problem than reuse distance measurement. Given a trace of length n, there is only $O(n)$ reuse windows but in total $O(n^2)$ footprint windows. This subsection focuses on the measurement problem, which prior work solved by either selecting a window subset to measure or constructing a model to approximate.

Direct Counting for Subset Windows. Agarwal *et al.*[75] counted the number of cold-start misses for all windows starting from the beginning of a trace (*cumulative cold misses*). Cumulative cold misses, together with warm-start region misses, were used to evaluate cache performance degradation caused by operation system and multiprogramming activity.

The footprint in single-length execution windows can be computed in linear time. On time-shared systems, the window of concern is the scheduling quantum. On these systems, the cached data of one process may be evicted by data brought in by the next process. Thiebaut and Stone computed what is essentially the single-window footprint by dividing a trace by the fixed interval of CPU scheduling quantum and taking the average amount of data access of each quantum[15].

Ding and Chilimbi[7] gave a sampling solution. At each access, it measures the footprint of a window ending at the current access. The length of the measured window is chosen at random.

For an execution of length n, direct counting measures the footprint in $O(n)$ windows. If we use direct counting to estimate all-window footprint, we have a sampling rate $O(\frac{1}{n})$. The sampling rate may be too low to be statistically meaningful, or it may be sufficient in practice. Without a solution for all-window analysis, we would not have a way to evaluate the accuracy of direct counting.

Footprint Equations. Suh *et al.*[16] and Chandra *et al.*[17] used a recursive equation to estimate the footprint. As a window of size w is increased to $w+1$, the change in the footprint depends on whether the new access is a miss. The equation is as follows: consider a random window w_t of size t being played out on some cache of infinite size. As we increase t, the footprint increases with every cache miss. Let $E[w_t]$ be the expected footprint of w_t, and $M(E[w_t])$ be the probability of a miss at the end of w_t. For window size $t+1$, the footprint either increases by an increment of one or stays the same depending on whether $t+1$ access is a cache miss.

$$E[w_{t+1}] = E[w_t](1-M(E[w_t]))+(E[w_t]+1)M(E[w_t]).$$

The term $M(E[w_t])$ requires simulating sub-traces of all size t windows, which is impractical. Suh *et al.*[16] solved it as a differential equation and made the assumption of linear window growth when the range of window sizes under consideration is small. On the other hand, Chandra *et al.*[17] computed the recursive relation bottom up. Neither method can guarantee a bound on the accuracy, i.e., how the estimate may deviate from the actual footprint.

In addition, these approaches produce the average footprint, not the distribution. The distribution can be important. Consider two sets of footprints, A and B. One tenth of A has size $10N$ and the rest has size 0. All of B has size N. A and B have the same average footprint N, but their different distribution can lead to very different types of cache interference. With the footprint distribution analysis[8], we now have a way to evaluate whether the average footprint produces the same composition results as the footprint distribution.

The past solutions on reuse distance often make similar estimates because the reuse distance is the footprint in a reuse window. These techniques[45,48-50,76] were mentioned in Subsection 3.2. They do not guarantee the precision of the estimation.

3.4 Analytical Models

Instead of measuring the reuse distance or footprint, a mathematical model may be used to characterize the cache performance. Apex-Map uses a parameterized model and a probe program to quickly find the model parameter for a program and a machine[77]. Ibrahim and Strohmaier[78] compared the result of synthetic probing and that of reuse distance profiling, while He *et al.*[79] used a fractal model to estimate the miss rate curve through efficient online analysis.

There was much work earlier on analytical models for memory paging performance. An extensive survey can be found in [2]. Saltzer[80], a designer of the Multics system, gave one simple formula (Subsection 3.6.1). He explained that "Although it is only occasionally that a mathematically tractable model happens to exactly represent the real-world situation, often an approximate model is good enough for many engineering calculations. The challenge ... is to maintain mathematical tractability in the face of obvious flaws and limitations in the range of applicability and yet produce a useful result." Saltzer's formula has been used by Strecker[81] in cache modeling.

Another type of analytical models is the *independent reference model*. Given a program with n pages, each has an independent access probability p that adds to 1, King[82] showed that a steady miss rate exists for fully associative caches managed by LFU, LRU, and FIFO replacement policies. Later studies gave efficient approximation methods for LRU and FIFO[83-84]. Gu and Ding[85] proved a simple relation between random access and the reuse distance distribution (which is uniform). The method of Dan and Towsley[84] can be used to analyze a more general case where data is divided into multiple groups and has different (random access) probabilities. It is a type of composable model.

3.5 Metrics Conversion and Denning's Law

Footprint is a form of working set. The working set theory is the scientific basis as much for memory management as it is for cache management. Denning defined the working set precisely as "the set of distinct pages referred to in a backward window of fixed size T."④ The average footprint for window length T is the average working set size for all size T windows.

A breakthrough in this area is a simple formula discovered by Denning④ and first published in 1972[43]. It shows the relation between the working set size, which is difficult to measure, and the frequency and interval of data reuses, which are easy to measure. The formula converts between two locality metrics. Metrics conversion is at the heart of the science of locality, because it shows that memory behavior and performance are different displays of the same underlying property.

While the proof of Denning and Schwartz[43] depends on idealized conditions in infinitely long executions, subsequent research has shown that the working set theory is accurate and effective in managing physical memory for real applications.

There are three ways to quantify the working set: as a limit value in Denning's original paper[3], as the time-space product defined by Denning and Slutz[86], and as the all-window footprint just defined in Subsection 3.3 (initially in [7]). The equation Denning discovered holds in all three cases. In our 2013 paper[10], we stated it as a law of locality and named it after its discoverer:

Denning's Law of Locality 1. *The working set is the second-order sum of the reuse frequency, and conversely, the reuse frequency is the second-order difference of the working set.*

The footprint theory subsumes the infinitely long case in the original working set theory and proves Denning's law for all executions. It gives a theoretical explanation to the long observed effectiveness of the working set theory in practice.

Easton and Fagin[87] gave another important formula for the conversion between the cold-start and warm-start miss ratios. The authors called it their "recipe". The recipe reveals that the (cold-start) lifetime in cache size C is the sum of the inter-miss times of the (warm) cache for sizes smaller than C. They found that their "estimate was almost always within 10∼15 percent of the directly observed average cold-start miss ratio." They further quoted the analysis of [88] as corroborating evidence. In these studies, as in the work of Denning and Schwartz[43], a program is assumed to be a stationary stochastic process. In the footprint theory, the Easton-Fagin formula can be derived directly, and the theory shows the correctness condition when it is used for finite-length program executions.

④Personal communication, December 17, 2013.

3.6 Locality Models of Shared Cache

3.6.1 Early Models

There are two types of cache sharing: the sharing between multiple time-switched programs, and the sharing between the instruction and data of the same program. Easton and Fagin[87] studied the former, comparing the difference between cold-start and warm-start miss ratios and computing the effect of task interruptions as a weighted average of expected cold-start miss ratios. Thiebaut and Stone[15] defined a precise measure called the *reload transient*. For a departing process, the reload transient is the amount of its cached data lost when it returns after another process is run. To compute the reload transient, Thiebaut and Stone[15] defined *cache footprint*, which is the number of data blocks a program has in cache. Given two programs A, B, the reload transient of A after B is the overlap between their cache footprints.

To compute footprints and their overlap, Thiebaut and Stone[15] assumed that a program has an equal probability of accessing any cache block. The probability is independent and identically distributed. The overlap is then computed from expectations of binomial distributions.

Instead of discrete probabilistic models, Strecker[81] put forward an intuitive notion that a program is a *continuous flow* and fills the cache at the rate that is the product of two probabilities: the chance of a miss and the chance that the miss results in a new location in the cache being filled. A differential equation was constructed since the fill rate is the derivative of the footprint over time. To compute the miss ratio, Strecker[81] used an analytical formula by Saltzer[80]. Saltzer[80] computed the inter-miss time in which he called the *headway* as the number of hits between successive misses.

The second type of cache sharing happens between the instruction and the data of a program. Stone *et al.*[89] investigated whether LRU produces the optimal allocation. Assuming that the miss rate functions for instruction and data are continuous and differentiable, the optimal allocation happens at the points "when miss-rate derivatives are equal"[90]. The miss rate functions, one for instruction and one for data, were modeled instead of measured. The authors showed that LRU is not optimal, but left open a question as to whether there is a bound on how close LRU allocation is to optimal allocation. The footprint theory can be used to compute the effective cache allocation (LRU allocation) among any group of programs.

As a component of the Wisconsin Wind Tunnel (WWT) project, Falsafi and Wood[91] developed a performance model for cache. They used the formulation of Thiebaut and Stone[15] but computed the overlap using a queuing model. In implementation, they measured the cold-start miss rate and used a reverse mapping to estimate the footprint. Since WWT ran the concurrent processes of a parallel program, the instruction code was shared between processes. The sharing was modeled as the shared footprint in the overall process footprint.

Falsafi and Wood[91] revised the terminology of Thiebaut and Stone[15] and redefined the *footprint* as the set of unique data blocks a program accesses. The *projection* of the footprint is the set of data blocks that the program leaves in cache. Viewed in another way, the footprint is the program data in an infinite cache, and the projection is the data in a finite cache. The footprint theory uses their definition of the word *footprint.*

3.6.2 Reuse Distance in Parallel Code

Reuse distance measures the locality of a program directly and does not rely on the assumptions that are necessary for analytical models. In a parallel program, we have two types of reuse distance. One considers only the accesses of a single task, and the other considers the interleaved accesses of all tasks. Using the terminology of Wu and Yeung[13] and Jiang *et al.*[50], we call them private reuse distance (PRD) and concurrent reuse distance (CRD). The new problem in analyzing the parallel locality is the relation between PRD and CRD.

Recent work has studied several solutions. Ding and Chilimbi[76] built models of data sharing and thread interleaving to compose CRD. Jiang *et al.*[50] tackled the composition problem using probabilistic analysis, in particular, the interval access probability based on [48], discussed in Subsection 3.2.

Multicore reuse distance by Schuff *et al.*[38] measures CRD directly using improved algorithms made efficient by sampling and parallelization. For loop-based code, Wu and Yeung gave a scaling model to predict how the reuse distance, both PRD and CRD, changes when the work is divided by a different number of threads[13]. These modeling techniques have found uses in co-scheduling[52] and multicore cache hierarchy design[13-14,92].

3.6.3 Non-Composability of Reuse Distance

A model is composable if the locality of a parallel execution can be computed from the locality of individual tasks. However, the reuse distance is insufficient to build composable models.

We illustrate this limitation by a counter example, first published in [8]. Fig.6 shows three short program traces. Programs A, B have the same set of private reuse distances (PRD). However, when running with a third program C, the pair A, C produces a different set of concurrent reuse distances (CRD) than the pair B, C. Assuming that the cache size is 4, the pair A, C has no capacity miss, but B, C has. The example also shows the same limitation for miss ratio. With identical reuse distances, A, B have the same number of misses in the private cache. But in the shared cache co-running with the same program C, they incur a different number of cache misses.

Fig.6 is a disproof by counterexample. It shows conclusively that PRD is not enough to compute CRD, and the solo-run miss ratio is not enough to compute the co-run miss ratio.

In the example, the reason for the different co-run locality is the different interaction based on the time span of a reuse. Consider the data accesses to a in A, B. They have the same private reuse distance, 2, but very different (logical) reuse times, 3 in A and 7 in B. When co-running with C, the reuse distance is lengthened because of the data accesses by C. Since the reuse in B spans over a longer time, it is affected more by cache sharing. As a result, the concurrent reuse distance for a is 4 in the A, C but 5 in the B, C co-run.

Chandra *et al.*[17] described three models of cache sharing. A simple one is the composition of reuse distance, called (*LRU*) *stack distance competition* (*SDC*). Since the model uses the reuse distance as the only input, it would have given the same prediction in our example for A, C and B, C. Therefore, it is a flawed model. A number of earlier studies have reached the same conclusion through experiments[93-95].

3.6.4 *Classic Composition Model*

Let A, B be two programs that share the same cache but do not share data. The effect of B on the locality of A is:

$$P\,(\text{capacity miss by } A \text{ when running with } B)$$
$$= P(A\text{'s reuse distance} + B\text{'s footprint} \geqslant \text{cache size}).$$

In this model, the cache interference (i.e., CRD) is computed by combining the footprint (i.e., the interference), and the reuse distance, i.e., the per-task locality. Specialized versions of this model were first developed by Suh *et al.*[16] for time-sharing systems and Chandra *et al.*[17] for multicore cache sharing. While Chandra[17] described and evaluated the composition for two programs, Chen and Aamodt[96] improved the accuracy when analyzing more programs with a greater number of cache conflicts. A later study by Jiang *et al.*[52] gives the general form of the classic model not tied to cache parameters such as associativity.

In the work of Suh *et al.*[16] and Chandra *et al.*[17], the footprint equation is iterative (see Subsection 3.3), while in the work of Jiang *et al.*[52], the footprint equation is statistical (see Subsection 3.2). Another footprint equation is the conversion formula by Denning and Schwartz[43]. These equations are not completely constrained, so the solution is not unique and depends on modeling assumptions.

The classic model is not simple as presented in the previous publications. In the work of Chandra *et al.*[17], hardware and program factors were considered together. Xie and Loh[97] noted that the model by Chandra *et al.* "is fairly involved; the large number of complex statistical computations would be very difficult to directly implement in hardware." In addition, the model has a high cost. It was not used in the comparison study of Zhuravlev *et al.*[94], because it was not "computationally fast enough to be used in the robust scheduling algorithm."

There is another weakness in usability. The two inputs, reuse distance and footprint, do not have a simple effect on the composed output. The complexity hinders the use of composable model in practice. As introduced in Subsection 2.2, the footprint theory shows many equivalent methods of composition. The subsection lists two other methods that are faster and easier to use.

3.7 Performance and Optimization

Cache is one of the factors in machine performance. Locality models show the frequency of cache hits and

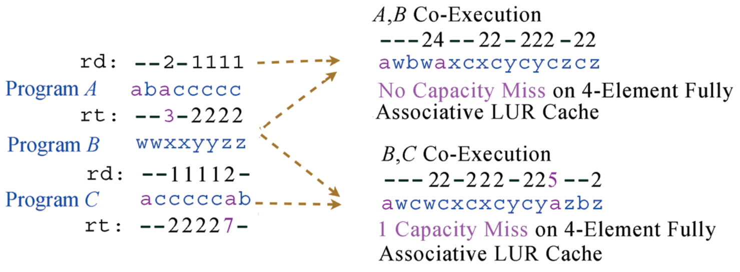


Fig.6. Non-composability of reuse distance. Programs A, B have the same set of reuse distances ("—" means *infty*), but A, C co-run has a different set of reuse distances than B, C co-run does.

misses. For performance analysis, the next question is the combined effect on the execution time, and the ultimate question is the limit to which the performance can be improved.

3.7.1 From Cache Misses to CPU Cycles

The effect of cache on execution time is traditionally given by two metrics, AMP (average miss penalty) and AMAT (average memory access time), which is the number of cycles necessary for respectively, a cache miss and a memory access on average[98].

On modern processors, the timing effect is increasingly complex. A recent analysis was conducted by Sun and Wang[99], who explained that AMAT is affected by the processor techniques for improving instruction-level parallelism, including pipelining, multiple functional units, out-of-order execution, branch prediction, speculation, and by the techniques for improving memory performance, including pipelined, multi-port, multi-bank cache, non-blocking cache, and data prefetching. The increasing complexity motivates the development of new metrics such as APC (access per cycle) studied in their paper[99].

Much of the timing delay is caused by events outside the CPU and the cache, in particular, the memory controller, the memory bus and the DRAM modules.

Zhao *et al.*[100] developed a model of pressure that includes both cache and memory bandwidth sharing using regression analysis to identify a piece-wise linear correlation between the memory latency and the memory bandwidth utilization. The model is not peer specific. The same utilization may be caused by one program or a group of programs. Wang *et al.*[101] gave an event model called DraMon to capture the probability of DRAM hits, misses and conflicts and the effect of contention and concurrency at the level of a DRAM bank. The event model was shown to be more accurate than linear and logarithmic regression[102].

It is important to manage contention and sharing at the memory layer, as shown by two recent techniques, bus-cycle modulation for execution throttling[103] and memory partitioning to reduce bank-level interference[104]. Next we turn the attention back to cache and review the techniques for reducing the cache interference.

3.7.2 Characterization of Interference

Xie and Loh[97] gave an animalistic classification of program interference. Based on the behavior in shared cache, a program belongs to one of the four animal classes. A turtle has little use of shared cache. A rabbit and a sheep both have a low miss rate. A rabbit is sensitive and tends to be affected by co-run peers, but a sheep does not. Both programs have small impacts on others. The last class is a devil, which has a high miss rate and impairs the performance of others but is not affected by others.

Other classifications include coloring of miss intensity, dual metrics of cache partitioning, and utility of cache space to performance. These are reviewed by Xie and Loh[97].

Jiang *et al.*[52] classified programs along two locality dimensions. The *sensitivity* is computed from the classic composition model (Subsection 3.6.4). It shows how a program is affected by others. The *competitiveness* is distinct data blocks per cycle (DPC), which is equivalent to the average footprint. If we divide each locality dimension into two halves, we have four classes, which we may call *locality classes*. Locality classes are not the same as animal classes. For example, a program can be extremely competitive, i.e., devilish, but may also be either sensitive or insensitive. This phenomenon was observed by Zhuravlev *et al.*[94], who showed that "devils were some of the most sensitive applications".

3.7.3 Optimal Co-Scheduling

Given a set of programs, co-scheduling divides them into co-run groups, where each group is run together. The goal is to minimize the interference within these groups, so to maximize resource utilization and co-run throughput. The interference depends mostly on the memory hierarchy, and the effect is non-linear and asymmetric.

While a locality model may predict the cache interference, the impact on performance depends on many other factors including the CPU speed, the effect of prefetching, the available memory bandwidth, and, if a program is I/O intensive, the speed of the disk or the network. Direct testing can most accurately measure the performance interference. Complete testing, however, has an exponential cost, since the number of subsets in an n-element set is 2^n. Note that solo executions are needed to compute the slowdown in group executions.

For pairwise co-runs, the interference can be represented by a complete graph where nodes are programs and edges have weights equal to pair-run slowdowns. Jiang *et al.*[105],⑤ showed that the optimization is min-weight perfect matching, and the problem is NP-hard. They gave an approximation algorithm that produces near-optimal schedules.

The throughput is often not the only goal. Other desirable properties include fairness, i.e., no program is

⑤First published by Jiang *et al.*[106]

penalized disproportionally due to unfair sharing, and quality of service (QoS), i.e., a program must maintain a certain level of performance.

As inputs, an optimal solution requires accurate prediction of co-run degradation. Prior solutions are either locality based (see Subsection 3.6) or performance based (this subsection). It is difficult for them to produce accurate prediction without expensive testing. For co-run miss rates, the footprint gives near real-time prediction, with an accuracy similar to exhaustive testing[10].

3.7.4 *Heuristics-Based Co-Scheduling*

In symbiotic scheduling (SOS), Snavely and Tullsen[107] used a sampling phase to test a number of possible co-run schedules and select the best one from these samples for the next (symbiosis) phase. They showed that a small number of possible schedules (instead of exhaustive testing) is sufficient to produce good improvements. The system was designed and tested for simultaneous multi-threading. Symbiotic scheduling assumes that program co-run behavior does not vary significantly over time, so the sampling phase is representative of performance in the remaining execution. Testing does not require program instrumentation.

Fedorova *et al.*[54] addressed the problem of performance isolation by suspending a program execution when needed. They gave a cache-fair algorithm to ensure a program runs at least at the speed with fair cache allocation. The technique is based on the assumption that if two programs have the same frequency of cache misses, they have the same amount of data in cache. In locality modeling, the assumption means uniform distribution of the access in cache. While the assumption is not always valid, the model is efficient for use in an OS scheduler to manage cache sharing in real time.

The two techniques are dynamic and do not need off-line profiling. However, on-line analysis may not be accurate and cannot predict interference in other program combinations. Furthermore, non-symbiotic pairing (during sampling) and throttling (for fairness) do not maximize the throughput.

Blagodurov *et al.*[95],⑥ developed the *Pain* classification. The degree of pain that application A suffers while it co-runs with B is affected by A's cache sensitivity, which is computed using the reuse distance profile (PRD), and B's cache intensity, which is measured by the number of last level cache accesses per million instructions. The Pain model is similar to the classic composition model described in Subsection 3.6.4 except that Pain uses the miss frequency rather than the footprint. The choice is partly for efficiency. Other online techniques also use the last-level cache miss rate as cache use intensity[108-109].

Pain is an offline solution. This idea is extended into an online solution called Distributed Intensity (DI), which uses only the miss rate. An application is classified as intensive if it is above the average miss rate and non-intensive otherwise. The scheduler then tries to group high-resource-intensity program(s) with low-resource-intensity program(s) on a multicore to mitigate the conflicts on shared resources[3,18,93-95,110-111].

Cache misses represent only a (small) subset of program accesses. In comparison, the footprint includes the effect of all cache accesses. Furthermore, the miss frequency depends on co-run peers and has the effect of circular feedback, since the peers are affected by the self. The result of counter-based modeling is specific to one grouping situation and may not be usable in other groupings. In comparison, footprint analysis collects "clean-room" statistics, unaffected by co-run peers program instrumentation or the analyzer code and usable for interference with any peers (which may be unknown at the time of footprint analysis). With the new theory in this thesis, footprint can be obtained with near real-time efficiency.

In an offline solution, Jiang *et al.*[105] defined the concept of *politeness* for a program as "the reciprocal of the sum of the degradations of all co-run groups that include the job." The politeness is measured by the effect on the execution time, not just the miss ratio, and is used to approximate optimal job scheduling.

In an online solution, the high cost of co-run testing is addressed in a strategy called Bubble-Up[112]. The strategy has two steps. First, a program is co-run against an expanding bubble to produce a sensitivity curve. The bubble is a specially designed probe program. In the second step, the pressure of the program is reported by another probe and probing run. Bubble-Up is a composable strategy, since each program is tested individually without testing all program combinations. Bubble-Up extracts the factors that determine the program execution time. In comparison, the footprint theory has a narrower scope, which includes just the factors that determine the program behavior in cache.

Two recent solutions use machine learning. Delimitrou and Kozyrakis[113] built a data center scheduler called Paragon. The design of Paragon identifies 10 sources of interference. It uses offline training (through probe programs) to build parameterized models on their performance impact. During online use, Paragon feeds the history information to a learning algorithm

⑥Journal version of [93-94].

called collaborative filtering. Collaborative filtering supports sparse learning. Based on a small amount of past data, it can predict application-application interference and application-machine match.

Statistical techniques have had many uses in performance analysis of parallel code, including clustering, factoring, and correlation[114], linear models (with nonlinear components)[115], queuing models[116], directed searches[117], and analytical models[118].

Machine learning is general and can consider different types of resources together. It is also scalable as more factors can be added by having additional learning. Paragon's learning technique observes the co-run results but has to be given the solo-run speed to compute the co-run slowdown. The cache model complements performance models, which can include the specialized model as a component. Locality metrics such as the footprint can be used as an input to a learning algorithm. While the strength of machine learning is the breadth and the general framework, the strength of the locality theory is the depth and the focused formulation. As a benefit of the latter, we now can understand the shared cache with mathematically tractable models and derive precise co-run miss ratios.

3.7.5 Performance Scaling Models

Using the PRD/CRD model[13], Wu *et al.*[14] conducted experiments on a wide range of symmetric multithreaded benchmarks on modest problem size and core counts and used their scaling framework to study the performance (average memory access time AMAT) over cache hierarchy scaling for large problem sizes on large-scale (LCMPs). The study focuses on the effect of hardware characteristics such as core counts, cache sizes, and cache organizations on different programs and program inputs, but not on hardware independent program characterization.

3.8 Related Techniques

3.8.1 Input-Centric Analysis

The early work in profiling examines multiple executions to identify what behavior is common. For example, Wall compared the hot variables and functions found in different executions of the same program[119]. Recent work has gone one step further to identify the patterns of change and predict how the behavior will differ from run to run. Shen called it *input-centric analysis*[120].

Input-centric analysis covers the intermediate ground between dynamic analysis, which is for a single execution, and static analysis, which is for all executions. For problems such as reuse distance and footprint, dynamic analysis is too specific, because the result is limited to what happens in one execution. Static analysis is too general, since it assumes all code paths are possible. Input-centric analysis provides a way to overcome these limitations.

Imperative to input-centric analysis is a metric whose results can be compared between different executions. The access of a memory location, for example, is not comparable because a program may allocate the same datum to different locations in different runs. Neither is the instruction making the access, since the same access may be made from different codes in different runs. Reuse distance is the first metric to enable input-centric analysis, since it is not tied to specific memory allocation or control flow and can be compared between different runs.

The first group of work studied how the reuse distance changes in different runs and developed statistical models of locality prediction (called whole-program locality)[5,36,121], miss-rate prediction[28], performance prediction (not just cross-input but also cross-architecture)[25,122], critical load instruction prediction[29], and locality phases[123-125]. Zhong *et al.* surveyed these and other techniques and categorized them as behavior (rather than code) based analysis, analogous to observation and prediction in the physical and biological sciences[5].

More recent work combined behavior and code analysis, in particular, showed how to predict the loop bounds in different runs. To characterize program inputs, Mao and Shen defined an extensible input characterization language (XICL)[126]. Jiang *et al.* defined the notion of seminal behavior, which is the smallest set of program values that collectively determine the iteration count of all loops[127]. Learning techniques such as classification trees were used to identify the seminal behavior[126,128]. Wu *et al.* later expanded the loop analysis to capture sequence patterns[129].

Input-centric analysis has been used to improve the feedback-driven program optimization (FDO) in the IBM XL C compiler[127] and the just-in-time (JIT) compiler in Java virtual machines[120,130-131]. Profiles from different inputs are routinely used in feedback-driven and iterative compiler optimization. The quality of optimization depends on the quality of profiles. The dependence has been examined using statistics[132-133].

3.8.2 Profiling and Performance Monitoring

The term profiling broadly refers to techniques that extract and analyze a subset of events in a program execution. Locality profiling extracts and analyzes the sequence of memory accesses. It does so by program instrumentation. At each memory reference, it inserts a call to pass the memory address to an analyzer. The instrumentation can be done at source or binary level. Source level instrumentation is made by a compiler such as GCC, Open64, and LLVM, usually at the level of the intermediate code. Binary instrumentation is by a binary rewriting tool. Both can be done statically, i.e., without running a program. Binary rewriting can also be done dynamically when a program is running.

The main problem of profiling is the cost of the instrumentation. A compiler can optimize the instrumented code statically. Another advantage is that the instrumentation tool is portable if a compiler is portable. In comparison, binary rewriting is architecture specific. For example, ATOM instruments only Alpha binary[134], and Pin x86 binary[135]. On the other hand, Pin can instrument dynamically loaded library, which a static tool cannot do.

Profiling does not model the timing effect, for which we need to either monitor an execution on actual hardware or reproduce it in a simulator.

Performance monitoring for parallel code has a long history[136-138]. Modern processors provide hardware counters to monitor hardware events with little or no run-time cost. The events related to memory performance include the frequency of cache misses, cache coherence misses, and various cycle counts, including stalled cycles. When many events are being monitored in a large system over a long execution, the large volume of results presents two problems. The first is the time and space cost of collecting and storing these results. The second is analysis — how to identify high-level information from low-level measurements.

These problems are solved by monitoring and visualization tools, including commercial ones such as Intel VTune Amplifier, AMD CodeAnalysist, and CrayPat, and open-source projects such as PAPI library[139], HPCToolkit[140], TAU[141], and Open|SpeedShop[142]. The aggregation of information is usually code centric, which shows performance in program functions and instructions. Vertical profiling identifies performance problems across a software stack[143]. Continuous program optimization (CPO) not only finds performance problems but also optimizes performance automatically[58,60,144-145]. In recent work, data-centric aggregation is used to pin-point locality problems more effectively, for issues of not just cache misses but also non-uniform memory access (NUMA) latency[146-148].

Bursty Sampling and Shadow Profiling. Arnold and Ryder pioneered a general framework to sample Java code, i.e., the first few invocations of a function or the beginning iterations of a loop[56]. It has been adopted for hot-stream prefetching in C/C++ in bursty sampling [20] and extended to sample both static and dynamic bursts for calling context profiling[149]. Shadow profiling pauses a program at preset intervals and forks a separate process to profile in parallel with the base program[64-65]. The reuse distance analysis is not a good target for these techniques because of the uncertain length of the reuse windows. However, the footprint can be easily sampled using shadow profiling. Reuse distance can then be computed using the conversion theory.

4 Conclusions

In this paper we have described the recent footprint theory of locality, including the definition and formal properties especially the footprint composition and the conversion between window-based statistics, i.e., the footprint, and reuse-based statistics, e.g., the miss ratio. We have surveyed a large literature, more than 140 publications over the past four decades, focusing on the working set theory, which lays the foundation of this research field, and recent performance models, which address the complex challenges posed by the modern multicore memory hierarchy. Through the review, we have appraised their strengths and weaknesses and pointed out the relation with the new footprint theory.

Nicholas Wirth titled his 1976 book "Algorithms + Data Structures = Programs" to emphasize the core subjects and their relations. We would modernize the figurative equation for use on today's machines and say "(Algorithms + Data Structures) × Locality = Efficient Programs". In theory, locality is fundamental in understanding the nature of computation. In practice, memory optimization is necessary in the design and use of every computing system. Locality research has made tremendous progress and immense impacts. This review focuses on the growing body of research to uncover the essential aspects of program behavior in shared cache and as a result enhance our ability to understand and manage program interaction on multicore systems.

Acknowledgment We thank Peter Denning and Xipeng Shen for always patiently and promptly answering our questions about their work, for the many people who worked with us at Rochester, and for the reviewers and the organizers of this special issue. Given the scope and depth of the past research, it is inevitable that the presentation fails to be complete and completely precise. We apologize for any omission and misrepresentation. Any error is entirely ours. We appreciate reader feedback, which can be sent to the email address listed in the first page of the paper.

A Chinese version of the first two sections have been co-authored with Yuan Liang and published in the Journal of Computer Engineering and Science[150].

References

[1] Zhang X, Dwarkadas S, Shen K. Towards practical page coloring-based multicore cache management. In *Proc. the EuroSys Conference*, April 2009, pp.89-102.

[2] Denning P J. Working sets past and present. *IEEE Transactions on Software Engineering*, 1980, 6(1): 64-84.

[3] Denning P J. The working set model for program behaviour. *Communications of the ACM*, 1968, 11(5): 323-333.

[4] Brock J, Luo H, Ding C. Locality analysis: A nonillion time window problem. In *Proc. Big Data Analytics Workshop*, June 2013.

[5] Zhong Y, Shen X, Ding C. Program locality analysis using reuse distance. *ACM TOPLAS*, 2009, 31(6): 1-39.

[6] Zhong Y, Orlovich M, Shen X, Ding C. Array regrouping and structure splitting using whole-program reference affinity. In *Proc. PLDI*, June 2004, pp.255-266.

[7] Ding C, Chilimbi T. All-window profiling of concurrent executions. In *Proc. the 13th PPoPP* (Poster Paper), Feb. 2008, pp.265-266.

[8] Xiang X, Bao B, Bai T, Ding C, Chilimbi T M. All-window profiling and composable models of cache sharing. In *Proc. PPoPP*, Feb. 2011, pp.91-102.

[9] Xiang X, Bao B, Ding C, Gao Y. Linear-time modeling of program working set in shared cache. In *Proc. PACT*, Oct. 2011, pp.350-360.

[10] Xiang X, Ding C, Luo H, Bao B. HOTL: A higher order theory of locality. In *Proc. ASPLOS*, March 2013, pp.343-356.

[11] Xiang X, Bao B, Ding C, Shen K. Cache conscious task regrouping on multicore processors. In *Proc. the 12th CCGrid*, May 2012, pp.603-611.

[12] Xiang X. A higher order theory of locality and its application in multicore cache management [Ph.D. Thesis]. Computer Science Dept., Univ. of Rochester, 2014.

[13] Wu M, Yeung D. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proc. PACT*, Oct. 2011, pp.264-275.

[14] Wu M, Zhao M, Yeung D. Studying multicore processor scaling via reuse distance analysis. In *Proc. the 40th ISCA*, June 2013, pp.499-510.

[15] Thiébaut D, Stone H S. Footprints in the cache. *ACM Transactions on Computer Systems*, 1987, 5(4): 305-329.

[16] Suh G E, Devadas S, Rudolph L. Analytical cache models with applications to cache partitioning. In *Proc. the 15th ICS*, June 2001, pp.1-12.

[17] Chandra D, Guo F, Kim S, Solihin Y. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. the 11th HPCA*, Feb. 2005, pp.340-351.

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

[19] Denning P J. Thrashing: Its causes and prevention. In *Proc. AFIPS Fall Joint Computer Conference, Part* 1, Dec. 1968, pp.915-922.

[20] Chilimbi T M, Hirzel M. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. PLDI*, June 2002, pp.199-209.

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

[22] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proc. SIGMETRICS*, June 2002, pp.31-42.

[23] Smith A J. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proc. the 2nd ICSE*, Oct. 1976, pp.286-292.

[24] Hill M D, Smith A J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 1989, 38(12): 1612-1630.

[25] Marin G, Mellor-Crummey J. Cross architecture performance predictions for scientific applications using parameterized models. In *Proc. SIGMETRICS*, June 2004, pp.2-13.

[26] Snir M, Yu J. On the theory of spatial and temporal locality. Technical Report, DCS-R-2005-2564, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2005.

[27] Fang C, Carr S, Önder S, Wang Z. Path-based reuse distance analysis. In *Proc. the 15th CC*, Mar. 2006, pp.32-46.

[28] Zhong Y, Dropsho S G, Shen X, Studer A, Ding C. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 2007, 56(3): 328-343.

[29] Fang C, Carr S, Önder S, Wang Z. Instruction based memory distance analysis and its application to optimization. In *Proc. PACT*, Sept. 2005, pp.27-37.

[30] Beyls K, D'Hollander E H. Discovery of locality-improving refactorings by reuse path analysis. In *Proc. the 2nd Int. Conf. High Performance Computing and Communications*, Sept. 2006, pp.220-229.

[31] Beyls K, D'Hollander E H. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proc. the 3rd ACM Conference on Computing Frontiers*, May 2006, pp.373-382.

[32] Kelly T, Cohen I, Goldszmidt M, Keeton K. Inducing models of black-box storage arrays. Technical Report, HPL-2004-108, HP Laboratories Palo Alto, 2004.

[33] Almeida V, Bestavros A, Crovella M, de Oliveira A. Characterizing reference locality in the WWW. In *Proc. the 4th International Conference on Parallel and Distributed Information Systems* (*PDIS*), December 1996, pp.92-103.

[34] Bennett B T, Kruskal V J. LRU stack processing. *IBM Journal of Research and Development*, 1975, 19(4): 353-357.

[35] Olken F. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report, LBL-12370, Lawrence Berkeley Laboratory, 1981.

[36] Ding C, Zhong Y. Predicting whole-program locality through reuse distance analysis. In *Proc. PLDI*, June 2003, pp.245-257.

[37] Zhong Y, Ding C, Kennedy K. Reuse distance analysis for scientific programs. In *Proc. Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, March 2002.

[38] Schuff D L, Kulkarni M, Pai V S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. the 19th PACT*, Sept. 2010, pp.53-64.

[39] Kim Y H, Hill M D, Wood D A. Implementing stack simulation for highly-associative memories. In *Proc. SIGMETRICS*, May 1991, pp.212-213.

[40] Sugumar R A, Abraham S G. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical Report, University of Michigan, August 1993.

[41] Burger D, Austin T. The SimpleScalar tool set, version 2.0. Technical Report, CS-TR-97-1342, Department of Computer Science, University of Wisconsin, June 1997.

[42] Almasi G, Cascaval C, Padua D A. Calculating stack distances efficiently. In *Proc. the ACM SIGPLAN Workshop on Memory System Performance*, June 2002, pp.37-43.

[43] Denning P J, Schwartz S C. Properties of the working set model. *Communications of the ACM*, 1972, 15(3): 191-198.

[44] Berg E, Hagersten E. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proc. ISPASS*, March 2004, pp.20-27.

[45] Berg E, Hagersten E. Fast data-locality profiling of native execution. In *Proc. SIGMETRICS*, June 2005, pp.169-180.

[46] Eklov D, Hagersten E. StatStack: Efficient modeling of LRU caches. In *Proc. ISPASS*, March 2010, pp.55-65.

[47] Eklov D, Black-Schaffer D, Hagersten E. Fast modeling of shared caches in multicore systems. In *Proc. the 6th*

HiPEAC, Jan. 2011, pp.147-157.

[48] Shen X, Shaw J, Meeker B, Ding C. Locality approximation using time. In *Proc. the 34th POPL*, Jan. 2007, pp.55-61.

[49] Shen X, Shaw J. Scalable implementation of efficient locality approximation. In *Proc. the 21st LCPC Workshop*, July 31-August 2, 2008, pp.202-216.

[50] Jiang Y, Zhang E Z, Tian K, Shen X. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proc. the 19th CC*, Mar. 2010, pp.264-282.

[51] Shen X, Shaw J, Meeker B, Ding C. Locality approximation using time. Technical Report, TR 901, Department of Computer Science, University of Rochester, December 2006.

[52] Jiang Y, Tian K, Shen X. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proc. HiPEAC*, Jan. 2010, pp.201-215.

[53] West R, Zaroo P, Waldspurger C A, Zhang X. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 2010, 44(4): 19-29.

[54] Fedorova A, Seltzer M, Smith M D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. the 16th PACT*, Sept. 2007, pp.25-38.

[55] Zhou S. An efficient simulation algorithm for cache of random replacement policy. In *Proc. the IFIP Int. Conf. Network and Parallel Computing*, Sept. 2010, pp.144-154.

[56] Arnold M, Ryder B G. A framework for reducing the cost of instrumented code. In *Proc. PLDI*, June 2001, pp.168-179.

[57] Hirzel M, Chilimbi T M. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[58] Cascaval C, Duesterwald E, Sweeney P F, Wisniewski R W. Multiple page size modeling and optimization. In *Proc. the 14th PACT*, Sept. 2005, pp.339-349.

[59] Zhong Y, Chang W. Sampling-based program locality approximation. In *Proc. the 7th ISMM*, June 2008, pp.91-100.

[60] Tam D K, Azimi R, Soares L, Stumm M. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. the 14th ASPLOS*, Mar. 2009, pp.121-132.

[61] Niu Q, Dinan J, Lu Q, Sadayappan P. PARDA: A fast parallel reuse distance analysis algorithm. In *Proc. IPDPS*, May 2012.

[62] Cui H, Yi Q, Xue J, Wang L, Yang Y, Feng X. A highly parallel reuse distance analysis algorithm on GPUs. In *Proc. the 26th IPDPS*, May 2012, pp. 1284-1294.

[63] Gupta S, Xiang P, Yang Y, Zhou H. Locality principle revisited: A probability-Based quantitative approach. In *Proc. the 26th IPDPS*, May 2012, pp.995-1009.

[64] Moseley T, Shye A, Reddi V J, Grunwald D, Peri R. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. CGO*, March 2007, pp.198-208.

[65] Wallace S, Hazelwood K. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. CGO*, Mar. 2007, pp.209-220.

[66] Cascaval C, Padua D A. Estimating cache misses and locality using stack distances. In *Proc. the 17th ICS*, June 2003, pp.150-159.

[67] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers, 2001.

[68] Beyls K, D'Hollander E H. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2005, 51(4): 223-250.

[69] Pugh W, Wonnacott D. Eliminating false data dependences using the Omega test. In *Proc. PLDI*, June 1992, pp.140-151.

[70] Chauhan A, Shei C Y. Static reuse distances for locality-based optimizations in MATLAB. In *Proc. the 24th ICS*, June 2010, pp.295-304.

[71] Shen X, Gao Y, Ding C *et al.* Lightweight reference affinity analysis. In *Proc. the 19th ICS*, June 2005, pp.131-140.

[72] Bao B, Ding C. Defensive loop tiling for shared cache. In *Proc. CGO*, Feb. 2013, pp.1-11.

[73] Bao B. Peer-aware program optimization [Ph.D. Thesis]. Computer Science Dept., Univ. of Rochester, January 2013.

[74] Yuan L, Ding C, Štefankovič D, Zhang Y. Modeling the locality in graph traversals. In *Proc. the 41st ICPP*, Sept. 2012, pp.138-147.

[75] Agarwal A, Hennessy J L, Horowitz M. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 1988, 6(4): 393-431.

[76] Ding C, Chilimbi T. A composable model for analyzing locality of multi-threaded programs. Technical Report, MSR-TR-2009-107, Microsoft Research, August 2009.

[77] Strohmaier E, Shan H. APEX-Map: A parameterized scalable memory access probe for high-performance computing systems. *Concurrency and Computation: Practice and Experience*, 2007, 19(17): 2185-2205.

[78] Ibrahim K Z, Strohmaier E. Characterizing the relation between Apex-Map synthetic probes and reuse distance distributions. In *Proc. ICPP*, Sept. 2010, pp.353-362.

[79] He L, Yu Z, Jin H. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *Proc. IPDPS*, May 2012, pp.1341-1351.

[80] Saltzer J H. A simple linear model of demand paging performance. *Communications of the ACM*, 1974, 17(4): 181-186.

[81] Strecker W D. Transient behavior of cache memories. *ACM Transactions on Computer Systems*, 1983, 1(4): 281-293.

[82] King W F. Analysis of demand paging algorithms. In *Proc. IFIP Congress*, August 1971, pp.485-490.

[83] Fagin R, Price T G. Efficient calculation of expected miss ratios in the independent reference model. *SIAM Journal of Computing*, 1978, 7(3): 288-297.

[84] Dan A, Towsley D F. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. SIGMETRICS*, May 1990, pp.143-152.

[85] Gu X, Ding C. Reuse distance distribution in random access. Technical Report, URCS #930, University of Rochester, January 2008.

[86] Denning P J, Slutz D R. Generalized working sets for segment reference strings. *Communications of the ACM*, 1978, 21(9): 750-759.

[87] Easton M C, Fagin R. Cold-start vs. warm-start miss ratios. *Communications of the ACM*, 1978, 21(10): 866-872.

[88] Shedler G, Tung C. Locality in page reference strings. *SIAM Journal on Computing*, 1972, 1(3): 218-241.

[89] Stone H S, Turek J, Wolf J L. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 1992, 41(9): 1054-1068.

[90] Thiébaut D, Stone H S, Wolf J L. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 1992, 41(6): 665-676.

[91] Falsafi B, Wood D A. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 1997, 7(1): 104-130.

[92] Wu M J, Yeung D. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proc. the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, June 2012, pp.2-11.

[93] Fedorova A, Blagodurov S, Zhuravlev S. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 2010, 53(2): 49-57.

[94] Zhuravlev S, Blagodurov S, Fedorova A. Addressing shared resource contention in multicore processors via scheduling. In *Proc. ASPLOS*, March 2010, pp.129-142.

[95] Blagodurov S, Zhuravlev S, Fedorova A. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 2010, 28(4): Article No.8.

[96] Chen X E, Aamodt T M. A first-order fine-grained multithreaded throughput model. In *Proc. HPCA*, Feb. 2009, pp.329-340.

[97] Xie Y, Loh G H. Dynamic classification of program memory behaviors in CMPs. In *Proc. CMP-MSI Workshop*, June 2008.

[98] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach (4th edition). Morgan Kaufmann, 2006.

[99] Sun X H, Wang D. APC: A performance metric of memory systems. *ACM SIGMETRICS Performance Evaluation Review*, 2012, 40(2): 125-130.

[100] Zhao J, Feng X, Cui H *et al.* An empirical model for predicting cross-core performance interference on multicore processors. In *Proc. PACT*, Sept. 2013, pp.201-212.

[101] Wang W, Dey T, Davidson J W *et al.* DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *Proc. HPCA*, Feb. 2014.

[102] Kim M, Kumar P, Kim H, Brett B. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *Proc. IPDPS*, May 2012, pp.1318-1329.

[103] Zhang X, Zhong R, Dwarkadas S, Shen K. A flexible framework for throttling-enabled multicore management (TEMM). In *Proc. ICPP*, Sept. 2012, pp.389-398.

[104] Liu L, Cui Z, Xing M *et al.* A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proc. PACT*, Sept. 2012, pp.367-376.

[105] Jiang Y, Tian K, Shen X, Zhang J, Chen J, Tripathi R. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE Trans. Parallel and Distributed Systems*, 2011, 22(7): 1192-1205.

[106] Jiang Y, Shen X, Chen J, Tripathi R. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proc. PACT*, Oct. 2008, pp.220-229.

[107] Snavely A, Tullsen D M. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. ASPLOS*, Nov. 2000, pp.234-244.

[108] Shen K. Request behavior variations. In *Proc. ASPLOS*, Mar. 2010, pp.103-116.

[109] Knauerhase R, Brett P, Hohlt B, Li T, Hahn S. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 2008, 38(3): 54-66.

[110] Denning P J. Equipment configuration in balanced computer systems. *IEEE Transactions on Computers*, 1969, C-18(11): 1008-1012.

[111] Wulf W A. Performance monitors for multi-programming systems. In *Proc. the ACM Symposium on Operating System Principles*, Oct. 1969, pp.175-181.

[112] Mars J, Tang L, Skadron K, Soffa M L, Hundt R. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 2012, 32(3): 88-99.

[113] Delimitrou C, Kozyrakis C. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS*, March 2013, pp.77-88.

[114] Ahn D H, Vetter J S. Scalable analysis techniques for microprocessor performance counter metrics. In *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 2002.

[115] Rodríguez G, Badia R M, Labarta J. Generation of simple analytical models for message passing applications. In *Proc. Euro-Par.*, Aug. 31-Sept. 3, 2004, pp.183-188.

[116] Jacquet A, Janot V, Leung C *et al.* An executable analytical performance evaluation approach for early performance prediction. In *Proc. IPDPS*, April 2003.

[117] Miller B P, Callaghan M D, Cargille J M *et al.* The Paradyn parallel performance measurement tool. *IEEE Computer*, 1995, 28(11): 37-46.

[118] Kerbyson D J, Hoisie A, Wasserman H J. Modelling the performance of large-scale systems. *IEE Proceedings - Software*, 2003, 150(4): 214-222.

[119] Wall D W. Predicting program behavior using real or estimated profiles. In *Proc. PLDI*, June 1991, pp.59-70.

[120] Tian K, Jiang Y, Zhang E Z, Shen X. An input-centric paradigm for program dynamic optimizations. In *Proc. OOPSLA*, Oct. 2010, pp.125-139.

[121] Shen X, Zhong Y, Ding C. Regression-based multi-model prediction of data reuse signature. In *Proc. the 4th Annual Symposium of the Los Alamos Computer Science Institute*, Oct. 2003.

[122] Marin G, Mellor-Crummey J. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proc. the Symposium of the Los Alamos Computer Science Institute*, Oct. 2005.

[123] Shen X, Ding C. Parallelization of utility programs based on behavior phase analysis. In *Proc. the International Workshop on Languages and Compilers for Parallel Computing*, Oct. 2005, pp.425-432.

[124] Shen X, Zhong Y, Ding C. Locality phase prediction. In *Proc. ASPLOS*, Oct. 2004, pp.165-176.

[125] Shen X, Zhong Y, Ding C. Predicting locality phases for dynamic memory optimization. *Journal of Parallel and Distributed Computing*, 2007, 67(7): 783-796.

[126] Mao F, Shen X. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proc. CGO*, Mar. 2009, pp.92-101.

[127] Jiang Y, Zhang E Z, Tian K *et al.* Exploiting statistical correlations for proactive prediction of program behaviors. In *Proc. the 8th CGO*, April 2010, pp.248-256.

[128] Cavazos J, Moss J E B. Inducing heuristics to decide whether to schedule. In *Proc. PLDI*, June 2004, pp.183-194.

[129] Wu B, Zhao Z, Shen X, Jiang Y, Gao Y, Silvera R. Exploiting inter-sequence correlations for program behavior prediction. In *Proc. OOPSLA*, Oct. 2012, pp.851-866.

[130] Arnold M, Welc A, Rajan V T. Improving virtual machine performance using a cross-run profile repository. In *Proc. OOPSLA*, Oct. 2005, pp.297-311.

[131] Tian K, Zhang E Z, Shen X. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proc. OOPSLA*, Oct. 2011, pp.445-462.

[132] Chen Y, Huang Y, Eeckhout L *et al.* Evaluating iterative optimization across 1000 datasets. In *Proc. PLDI*, June 2010, pp.448-459.

[133] Wu B, Zhou M, Shen X *et al.* Simple profile rectifications go a long way — Statistically exploring and alleviating the effects of sampling errors for program optimizations. In *Proc. the European Conference on Object-Oriented Programming*, July 2013, pp.654-678.

[134] Srivastava A, Eustace A. ATOM: A system for building customized program analysis tools. In *Proc. PLDI*, June 1994, pp.196-205.

[135] Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V J, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, June 2005, pp.190-200.

[136] Wagner Meira Jr., LeBlanc T, Poulos A. Waiting time analysis and performance visualization in Carnival. In *Proc. ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[137] Reed D A, Elford C L, Madhyastha T M, Smirni E, Lamm S E. The next frontier: Interactive and closed loop performance steering. In *Proc. ICPP Workshop*, Aug. 1996, pp.20-31.

[138] Darema-Rogers F, Pfister G F, So K. Memory access patterns of parallel scientific programs. In *Proc. SIGMETRICS*, May 1987, pp.46-58.

[139] Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 2000, 14(3): 189-204.

[140] Adhianto L, Banerjee S, Fagan M, Krentel M, Marin G, Mellor-Crummey J, Tallent N R. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010, 22(6): 685-701.

[141] Shende S, Malony A D. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 2006, 20(2): 287-311.

[142] Schulz M, Galarowicz J, Maghrak D, Hachfeld W, Montoya D, Cranford S. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 2008, 16(2/3): 105-121.

[143] Hauswirth M, Sweeney P F, Diwan A. Temporal vertical profiling. *Software: Practice and Experience*, 2010, 40(8): 627-654.

[144] Childers B, Davidson J, Soffa M L. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proc. Symp. Parallel and Distributed Processing*, April 2003.

[145] Cascaval C, Duesterwald E, Sweeney P F, Wisniewski R W. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 2006, 50(2/3): 239-248.

[146] McCurdy C, Vetter J S. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. ISPASS*, March 2010, pp.87-96.

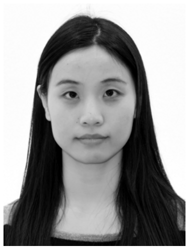
[147] Liu X, Mellor-Crummey J M. Pinpointing data locality problems using data-centric analysis. In *Proc. the 9th CGO*, April 2011, pp.171-180.

[148] Liu X, Mellor-Crummey J. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proc. the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2014, pp.259-272.

[149] Zhuang X, Serrano M J, Cain H W, Choi J. Accurate, efficient, and adaptive calling context profiling. In *Proc. PLDI*, June 2006, pp.263-271.

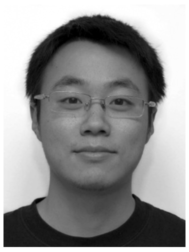
[150] Ding C, Yuan L. Program interaction on multicore: Theory and applications. *Computer Engineering and Science*, 2014, 36(1): 1-5. (In Chinese)

Chen Ding received his Ph.D. degree from Rice University, M.S. degree from Michigan Technological University, and B.S. degree from Beijing University, all in computer science before joining University of Rochester in 2000. His research received young investigator awards from NSF and DOE. He co-founded the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPC) and was a visiting researcher at Microsoft Research and a visiting associate professor at MIT. He is an external faculty fellow at IBM Center for Advanced Studies.



Xiaoya Xiang graduated in 2005 from Huazhong University of Science and Technology with a B.S. degree in computer science and technology and at the same time from Wuhan University with a B.S. degree in finance. She got her M.S. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2008. She earned her Ph.D. degree in computer science at the University of Rochester in 2013. She is now a software engineer at Twitter Inc., where her main focus is the runtime performance of the Twitter services in a cloud environment.

Bin Bao is a senior software engineer at Qualcomm Technologies, Inc. Prior to joining Qualcomm in 2013, Bin spent one year at Adobe Inc. as a computer scientist. He received his Ph.D. degree in computer science from University of Rochester in 2013, M.S. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences in 2007, and B.S. degree in software engineering from the University of Science and Technology of China in 2004. His current research interests include program analysis and compilation for graphics processors.



Hao Luo is a third year Ph.D. student in the Department of Computer Science, University of Rochester. His research interest lies on performance modeling of multithreaded applications, locality-aware task management, and program behavior analysis.

Ying-Wei Luo received his Ph.D. degree in computer science from Peking University in 1999. He is a full professor of computer science in the School of Electronics Engineering and Computer Science (EECS) in Peking University. His research interests include operating system, system virtualization, and cloud computing.

Xiao-Lin Wang received his Ph.D. degree in computer science from Peking University in 2001. He is now an associate professor of computer science in the School of EECS in Peking University. His research interests include operation system, system virtualization, and cloud computing.