J. Parallel Distrib. Comput. 73 (2013) 1011–1027

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Locality principle revisited: A probability-based quantitative approach☆

Saurabh Gupta *, Ping Xiang, Yi Yang, Huiyang Zhou

Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, USA

ARTICLE INFO

Article history:
Received 5 September 2012
Received in revised form
15 January 2013
Accepted 19 January 2013
Available online 5 February 2013

Keywords:
Locality of references
Probability
Memory hierarchy
Last level cache
Cache replacement policy
Data prefetching
Locality optimizations

ABSTRACT

This paper revisits the fundamental concept of the locality of references and proposes to quantify it as a conditional probability: in an address stream, given the condition that an address is accessed, how likely the same address (temporal locality) or an address within its neighborhood (spatial locality) will be accessed in the near future. Previous works use reuse distance histograms as a measure of temporal locality. For spatial locality, some ad hoc metrics have been proposed as a quantitative measure. In contrast, our conditional probability-based locality measure has a clear mathematical meaning and provides a theoretically sound and unified way to quantify both temporal and spatial locality. We showcase that our quantified locality measure can be used to evaluate compiler optimizations, to analyze the locality at different levels of memory hierarchy, to optimize the cache architecture to effectively leverage the locality, and to examine the effect of data prefetching mechanisms.

1. Introduction

The locality principle has been widely recognized and leveraged in many disciplines [9]. In computer systems, locality of references is the fundamental principle driving the memory hierarchy design. Specifically, memory hierarchy relies on two types of locality, temporal and spatial locality. Temporal locality indicates that after an address is referenced, the same datum is likely to be re-referenced in the near future. Spatial locality, on the other hand, indicates that some neighbors of the referenced address are likely to be accessed in the near future. Given the significance of the locality principle, previous works have attempted to quantify the locality to better understand the reference patterns and to guide the compiler and architecture design to enhance/exploit program locality. Between the two types of locality, the histogram of reuse distances [28] or LRU (least recently used) stack distances [19] computed from an address trace is used as a measure of temporal locality. For spatial locality, however, there is a lack of consensus for such a quantitative measure and several ad-hoc metrics [4,5,14,26] are proposed based on intuitive notions. In this paper, we revisit the concept of locality.

First, we propose to use conditional probabilities as a formal and unified way to quantify both temporal and spatial locality. For any address trace, we define spatial locality as the following conditional probability: *for any address, given the condition that it is referenced, how likely an address within its neighborhood will be accessed in the near future*. From this definition, we can see that the conditional probability depends on the definition of two parameters, the size of the neighborhood and the scope of near future. Therefore, spatial locality is a function of these two parameters and can be visualized with a 3D mesh. Temporal locality is reduced to a special case of spatial locality when the neighborhood size is zero bytes.

Second, we examine the relationship between our probability-based temporal locality and the reuse distance histogram, which is used in previous works for quantifying temporal locality. We derive that our proposed temporal locality (using one particular definition of near future) is the cumulative distribution function of the likelihood of reuse in the near future while the reuse histogram is the probability distribution function of the same likelihood, thereby offering theoretical justification for reuse distance histograms. The difference function of our proposed spatial locality over the parameter of the near future window

☆ This paper is an extended version of the conference paper titled "Locality Principle Revisited: A Probability-Based Quantitative Approach" published in the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012) May, 2012. It adds a new section (Section IV-F) "Spatial Locality vs. Temporal Locality in Last Level Caches" into the original paper.

* Correspondence to: c/o Huiyang Zhou, Department of Electrical and Computer Engineering, Box 7911, NC State University, Raleigh, NC 27695-7911, USA.
E-mail address: sgupta12@ncsu.edu (S. Gupta).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2013.01.010

size provides reuse distance histograms for different neighborhood sizes. In comparison, the difference function of our proposed spatial locality over the parameter of the neighborhood size generates stride histograms for different near future window sizes.

Third, we use the proposed locality to analyze the effect of locality optimizations, sub-blocking/tiling in particular. The quantitative locality presented in 3D meshes visualizes the locality changes for different near future windows and neighborhood sizes, which helps us to evaluate the effectiveness of code optimizations for a wide range of cache configurations.

Fourth, we use the proposed locality to analyze and optimize cache hierarchies. We show that interesting insights can be revealed from the locality measure of access streams at different levels of memory hierarchy, which can then be used to drive the cache architecture optimization. We also showcase that our locality computation can be used to easily evaluate the effect of data prefetchers. Furthermore, we show how the locality correlates to the cache replacement policy. Given the importance of last-level caches (LLCs) in multi-core architectures, we also present a detailed locality analysis to make a case for LLCs with large block sizes and relatively simple replacement policies.

Our proposed locality measure clearly presents the characteristics of a reference stream and helps to reveal interesting insights about the reference patterns. Besides locality analysis, another commonly used way to understand the memory hierarchy performance of an application is through cache simulation. Compared to cache simulation, our proposed measure provides a more complete picture since it provides locality information at various neighborhood sizes and different near future window scopes while one cache simulation only provides one such data point.

The remainder of the paper is organized as follows. Section 2 defines our probability-based locality, derives the difference functions, and highlights the relationship to reuse distance histograms. Section 3 focuses on quantitative locality analysis of compiler optimizations for locality enhancement. Section 4 discusses locality analysis of cache hierarchies and presents how the visualized locality reveals interesting insights about locality at different levels of cache hierarchy and about data prefetching mechanisms. We also highlight how the locality information helps to drive memory hierarchy optimizations. Section 5 presents a GPU-based parallel algorithm for our locality computation. Section 6 discusses related work. Finally, Section 7 summarizes the key contributions of the paper.

2. Quantifying locality of reference using conditional probabilities

2.1. Temporal locality

Temporal locality captures the characteristics of the same references being reused over time. We propose to quantify it using the following conditional probability: given the condition that an arbitrary address, A, is referenced, the likelihood of the same address to be re-accessed in the near future. Such conditional probability can be expressed as $P(X_n = A,\ \exists n \text{ in near future} \mid X_0 = A)$, where X_0 is the current reference and X_n is a reference in the near future. Since an address stream usually contains more than one unique address, the temporal locality of an address stream can be expressed as a combined probability, as shown in Eq. (1), where M is the number of unique addresses in the stream.

$$\begin{aligned} TL &= P\,(X_n = A,\ \exists n \text{ in near future} \mid X_0 = A) \\ &= \sum_{i=1}^{M} P(X_n = A_i, \exists n \text{ in near future} \mid X_0 = A_i) \\ &\quad \times P(X_0 = A_i). \end{aligned} \tag{1}$$

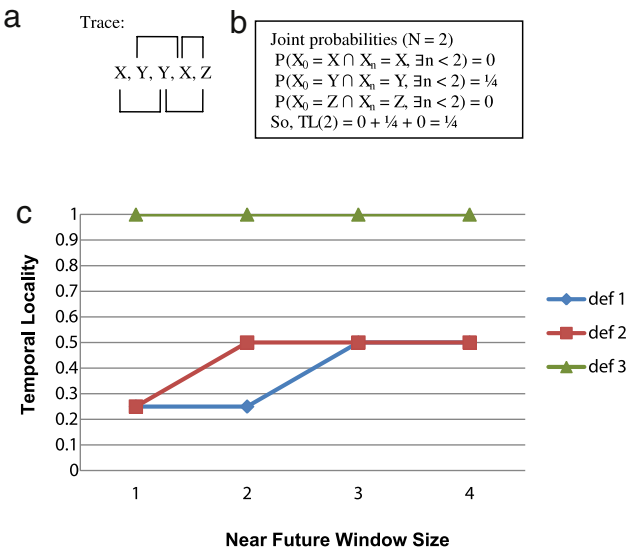


Fig. 1. Calculating temporal locality. (a) Reference trace (also showing a moving future window of size 2); (b) Joint probability using Def. 1 (next-n-address) of near future; (c) Temporal locality $TL(N)$ using different definitions of near future.

Eq. (1) can be further derived using Bayes' theorem, shown as Eq. (2).

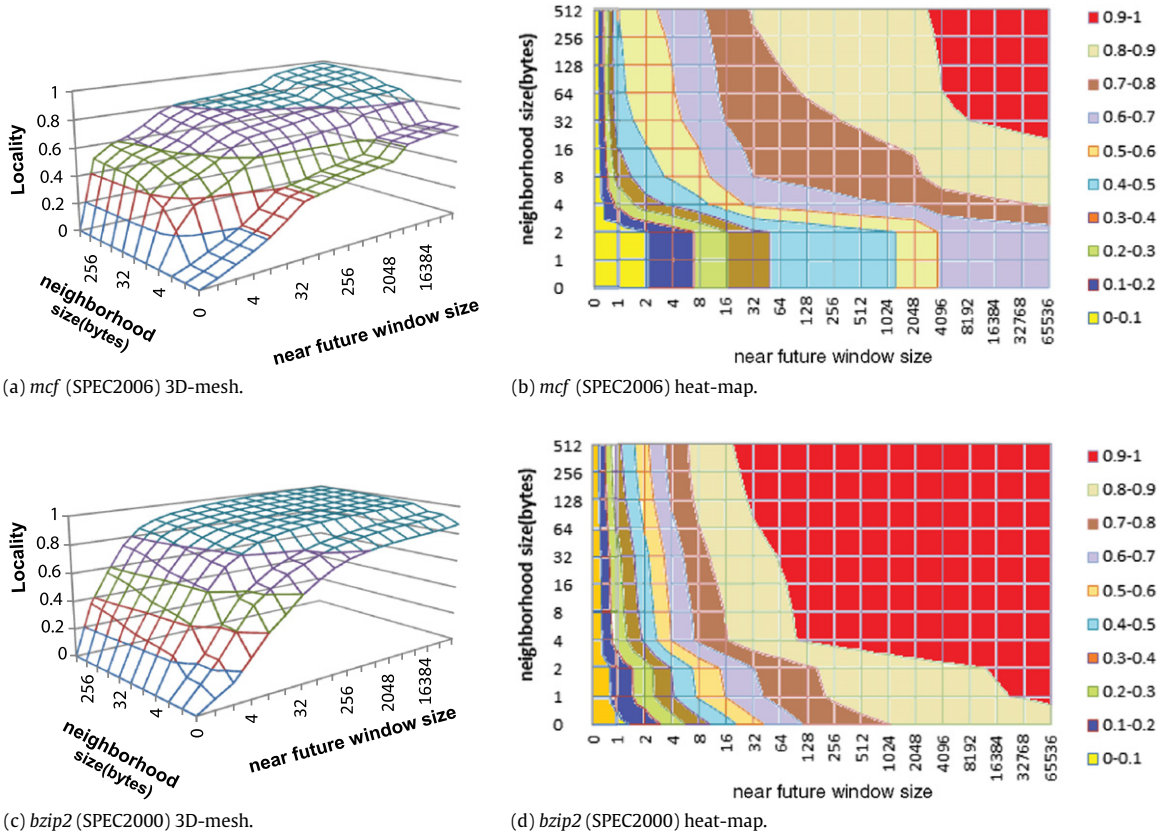
$$TL = \sum_{i=1}^{M} P\,(X_n = A_i,\ \exists n \text{ in near future} \cap X_0 = A_i)\,. \tag{2}$$

To calculate the joint probabilities, $P\,(X_n = A_i, \exists n$ in near future $\cap\, X_0 = A_i)$, we need to formulate a way to define the term 'near future'. We consider three such definitions and represent temporal locality as $\boldsymbol{TL}(\boldsymbol{N}) = P(X_n = A,\ \exists n < N \mid X_0 = A)$, where N is the scope of the near future or the future window size.

- *Def.* 1 of near future (*Next-n-address*): the next N addresses in the address stream.
- *Def.* 2 of near future (*Next-n-unique-address*): the next N *unique* addresses in the address stream.
- *Def.* 3 of near future (*Next-n-unique-block*): the next N unique *block* addresses in the address stream given a specific cache block size. In other words, we ignore the block offset when we count the next N unique addresses.

Based on the definition of the near future, the joint probability $P(X_0 = A \cap X_n = A,\ \exists n < N)$ can be computed from an address trace with a moving near future window. In a trace of the length S, for each address (i.e., the event $X_0 = A$), if the same address is also present in the near future window (i.e., the event $X_n = A,\ \exists n < N$), we can increment the counter for the joint event $(X_0 = A \cap X_n = A,\ \exists n < N)$. After scanning the trace, the ratio of this counter over $(S - 1)$ is the probability of $P(X_0 = A \cap X_n = A,\ \exists n < N)$. The reason for $(S - 1)$ is to account for the fact that there are no more references (or any potential reuse) after the last one in the trace. During the scan through the trace, if Def. 1 (next-n-address) of near future is used, the moving window has a fixed size while Def. 2 (next-n-unique-address) and Def. 3 (next-n-unique-block) require a window of variable sizes. We illustrate the process of calculating temporal locality with an example shown in Fig. 1.

Considering the trace shown in Fig. 1, for each address (except the last one), we will examine whether there is a reuse in its near future window. If the near future window size is 2 and we use Def. 1 (next-n-address) of near future, we will use a window of size 2 to scan through the trace as shown in Fig. 1(a). Based on the joint probabilities, the temporal locality, $TL(2)$, is $1/4$ as shown in Fig. 1(b). If we follow Def. 2 (next-n-unique-address) of near future,



(a) *mcf* (SPEC2006) 3D-mesh. (b) *mcf* (SPEC2006) heat-map.

(c) *bzip2* (SPEC2000) 3D-mesh. (d) *bzip2* (SPEC2000) heat-map.

Fig. 2. The spatial locality plots of the benchmarks, *mcf* (SPEC2006) and *bzip2* (SPEC2000) when using the Def. 2 (next-*n*-unique-address) of near future and Def. 1 (modulo-based) of neighborhood. Temporal locality is a special case of spatial locality with neighborhood size as 0.

the temporal locality, $TL(2)$, is 2/4 since each near future window will contain 2 unique addresses. If we use Def. 3 (next-*n*-unique-block) of near future and assume that X, Y, and Z are in the same block, the temporal locality, $TL(2)$, is 1. The temporal locality with various near future window sizes is shown in Fig. 1(c) for these three definitions of near future.

2.2. Spatial locality

Spatial locality captures the characteristics of the addresses within a neighborhood being referenced over time. We propose to quantify it as the following probability: given the condition that an arbitrary address, A, is referenced, the likelihood of an address in its neighborhood to be accessed in the near future. Such conditional probability can be expressed as $P(X_n \text{ in } A\text{'s neighborhood}, \exists n < N \mid X_0 = A)$, where X_0 is the current reference and X_n is a reference in the near future. Similar to temporal locality, the spatial locality of an address stream can be computed as the combined probability of all the unique addresses in the trace and it can be expressed as Eq. (3), where M is the number of unique addresses in the stream.

$$\begin{aligned} SL &= P\,(X_n \text{ is in the neighborhood of } A,\ \exists n < N \mid X_0 = A) \\ &= \sum_{i=1}^{M} P(X_n \text{ is in the neighborhood of } A_i, \\ &\quad \exists n < N \mid X_0 = A_i) \times P\,(X_0 = A_i) \\ &= \sum_{i=1}^{M} P(X_n \text{ is in the neighborhood of } A_i, \\ &\quad \exists n < N \cap X_0 = A_i). \end{aligned} \tag{3}$$

To compute the joint probabilities, $P\,(X_n \text{ is in the neighborhood of } A_i,\ \exists n < N \cap X_0 = A_i)$, we need to define the meaning of the term 'neighborhood' in addition to near future. We consider two definitions for neighborhood and represent spatial locality as $\boldsymbol{SL}(\boldsymbol{N}, \boldsymbol{K})$, where N is the near future window size and K is a parameter defining the neighborhood size.

- *Def.* 1 of neighborhood (*modulo-based*): an address X is within the neighborhood of Y if $|X - Y| < K$, where $2 * (K - 1)$ is the size of the neighborhood. The reason for $2 * (K - 1)$ rather than $(K - 1)$ is to account for the absolute function of $(X - Y)$.
- *Def.* 2 of neighborhood (*block-based*): two addresses X and Y are in the same cache block given the cache block size as K. In other words, X is within the neighborhood of Y if $X - (X \bmod K) = Y - (Y \bmod K)$.

Based on the definition of neighborhood, the spatial locality of a reference trace can be computed by moving a near future window through the trace, similar to temporal locality calculation. The difference is that for temporal locality, we require reuses of the same addresses while for spatial locality, the addresses in the window are inspected to see whether there exists a reference X_n, $n < N$, falling in the neighborhood of X_0. If we change the definition of neighborhood size to zero (i.e., $K = 1$), spatial locality is reduced to temporal locality.

From Eq. (3), we can see that spatial locality is a function of two parameters, the neighborhood size, K, and the near future window size, N. As such, we can visualize the spatial locality of an address stream with a 3D mesh or a 2D heat-map for different neighborhood and near future window sizes. Fig. 2 shows the spatial locality, using the Def. 2 (next-*n*-unique-address) of near future and Def. 1(modulo-based) of neighborhood, of the benchmarks SPEC2006 *mcf* and SPEC2000 *bzip2* (see the experimental methodology in Section 4.1). Temporal locality is the curve where the neighborhood size is zero.

Locality visualized in 3D meshes or heat-maps clearly shows the characteristics of reference streams. First, a contour in the

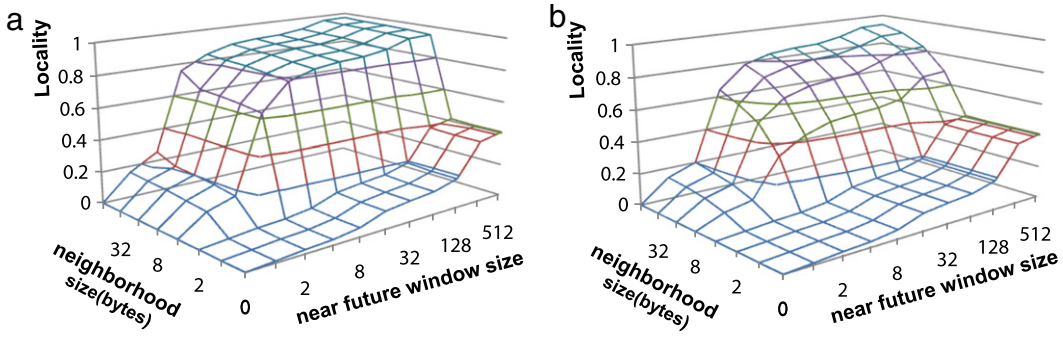


Fig. 3. Spatial locality of the benchmark *sphinx* for neighborhood definitions. (a) Definition 1 (modulo based) (b) definition 2 (block based).

3D mesh/heat-map at a certain locality score (e.g., 0.9) can be used to figure out the sizes of an application's working data sets. Comparing Fig. 2(b)–(d), it is evident that *bzip2* has a much smaller working data set than *mcf*. We can also see that the size of the working set varies for different neighborhood sizes. For example, $SL(8192, 32)$ of *mcf* is close to 0.9, indicating the working set size is slightly bigger than $2 * 32\ \text{B} * 8192 = 512$ kB when the neighborhood size is 64 (=$2 * 32$) bytes. On the other hand, $SL(65\,536, 16)$ of *mcf* is little less than 0.9, showing that the working set is greater than $(2 * 16\ \text{B} * 65\,536) = 2$ MB when the neighborhood size is 32 (=$2 * 16$) bytes.

Second, the quantified locality shows how it can be exploited. From Fig. 2(a), we can see that for *mcf* the reuse of the same data at the byte/word granularity (i.e., temporal locality) is limited even with a large near-future window. In contrast, there exists significant spatial locality that can be exploited with a moderate near future window, which also explains why the working set sizes vary significantly for different neighborhood sizes. *Bzip2*, on the other hand, shows much better temporal and spatial locality, which can be captured effectively with a relatively small neighborhood size. Note that the function $SL(N, K)$ is monotonous along either the N or K direction since our conditional probability definitions are accumulative. In other words, a reuse within a small neighborhood (or close future) is also a reuse for a larger neighborhood (or not-so-close future).

Third, our proposed locality can be used to reveal interesting insights which are typically not present through cache simulations because one cache simulation only provides one data point in this 3D mesh. As one example, the benchmark, *hmmer*, shows steps in the locality scores (Fig. 5(a)) when the near future size is at 32 and 8192, which implies that there is a good amount of data locality that we cannot leverage until we can explore the reuse distance of 8192. Such observations from the 3D mesh enable us to understand why and whether a particular cache configuration performs much better than the other ones.

Between heat maps and 3D meshes, locality presented in heat-maps better reveals the working set information. On the other hand, 3D-meshes are more useful in showing how the locality changes along either the neighborhood size or the near future window size, which indicates how the locality can be exploited.

Different definitions of 'near future' lead to different near future window sizes as illustrated in Fig. 1. Such differences, however, turn out to have little impact when we compute locality for most SPEC CPU benchmarks. Between the two definitions of neighborhood, although block-based definition models the cache behavior more accurately, it fails to capture spatial locality across the block boundary. For example, for a perfect stride pattern, 1, 2, 3, 4, 5, …, the spatial locality, using modulo-based definition of neighborhood, will be 1 for any neighborhood size and any future window size greater than 0. If we use block-based definition of neighborhood, the spatial locality will be dependent upon the neighborhood (or cache block) size. If the neighborhood/block size is 2 bytes, the spatial locality is 0.5 for any future window size. If the neighborhood/block size is B bytes, the spatial locality becomes $(B-1)/B$ for any future window size greater than or equal to B. In Fig. 3, we show the spatial locality for the benchmark *sphinx* with neighborhood defined using modulo-based definition (Fig. 3(a)) and block-based definition (Fig. 3(b)). From the figure, we can observe the similar behavior, i.e., increasing locality values for larger neighborhood sizes in Fig. 3(b). In contrast, Fig. 3(a) shows a stepwise increase in locality values. Based on these observations, we choose to use next-n-unique-address definition of near future and modulo-based definition of neighborhood for locality analysis. Whenever the objective is to estimate the cache performance, the next-n-unique-block definition of near future and the block-based definition of neighborhood can be used instead.

2.3. The relationship between spatial and temporal locality

Based on our probability-based definitions of spatial and temporal locality, temporal locality can be viewed as a special case of spatial locality. Beyond this observation, there also exist some intriguing subtleties between the two types of locality. In particular, we can change the neighborhood definition in spatial locality so that the exact same addresses are excluded from the neighborhood. In other words, the neighborhood definition becomes $0 < |X - Y| < K$. In this case, an interesting question is: for any address trace, is the sum of its spatial locality and its temporal locality always less than or equal to 1?

An apparent answer is yes since the reuses from the same addresses are excluded from the neighborhood definition. This is also our initial understanding until our locality computation results show otherwise. After a detailed analysis, it is found that the events for temporal and spatial locality are not always disjoint and therefore the sum can be larger than 1. In other words, a memory access can exhibit both spatial locality and temporal locality. To elaborate, we use Def. 1 (next-n-address) of near future and modulo-based definition of neighborhood with the modification that same addresses being excluded. Other definitions of the two terms can be applied similarly. Following Eqs. (1) and (3), temporal and spatial locality can be expressed as the following two conditional probabilities: $P(X_n = A,\ \exists n < N \mid X_0 = A)$ and $P(0 < |X_n - A| < K,\ \exists n < N \mid X_0 = A)$, respectively. Both probabilities can be computed using the sum of the joint probabilities $\sum_{i=1}^{M} P(X_n = A_i,\ \exists n < N \cap X_0 = A_i)$ and $\sum_{i=1}^{M} P(0 < |X_n - A_i| < K,\ \exists n < N \cap X_0 = A_i)$. Since each A_i is unique, the events $X_0 = A_i$ are disjoint for different A_i. So, we can focus on one such address, e.g., A_1. We can use the Venn diagram to capture the relationship among the events: $X_0 = A_1, X_1 = A_1 + \Delta$, and $X_2 = A_1$, as shown in Fig. 4. From Fig. 4, we can see that $(X_0 = A_1 \cap X_2 = A_1)$ and $(X_0 = A_1 \cap X1 = A_1 + \Delta)$ are not disjoint. An intuitive explanation is that for an address sequence

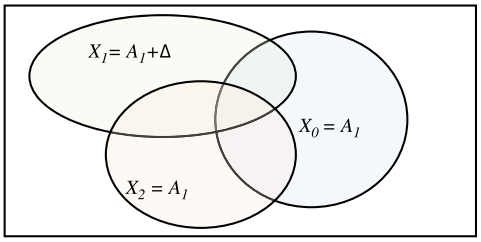


Fig. 4. The Venn diagram showing the relationship among events $X_0 = A_1, X_1 = A_1 + \Delta$, and $X_2 = A_1$.

such as $A_1, A_1 + 1, A_1$, the third reference (A_1) contributes to both temporal locality of the first reference and spatial locality of the second reference. Therefore, the events for temporal and spatial locality are not disjoint although the references to same addresses are excluded from the neighborhood definition.

2.4. Difference functions of locality measures and the relationship to reuse distance histograms

Next, we derive the relationship of our probability-based temporal locality to reuse distance histograms, which are commonly used in previous works to quantify temporal locality. In sequential execution, reuse distance is the number of distinct data elements accessed between two consecutive references to the same element. The criterion to determine the 'same' element is based on the cache-block granularity, i.e., the block offsets are omitted when determining reuses. The reuse distance provides a capacity requirement for fully associative LRU caches to fully exploit the reuses. The histogram of reuse distances is used as the quantitative summary/signature of locality present in a trace since it captures all the reuses at different distances.

Based on our probability-based measure, temporal locality $TL(N)$ represents the probability $P(X_n = A,\ \exists n < N \mid X_0 = A)$. As this probability is a discrete function of N, we can take a difference function of temporal locality $\Delta TL(N) = TL(N) - TL(N-1) = P(X_{N-1} = A \mid X_0 = A)$. This function represents the frequency of reuses at the exact reuse distance of $(N-1)$. If we use Def. 3 (next-n-unique-block) of the term near future, the difference function becomes: $\Delta TL(N) = P(X_{N-1} = block\_addr(A) \mid X_0 = A)$ where $block\_addr(A) = A - (A \bmod block\_size)$ and it is essentially the same as the reuse distance histogram. Therefore, we conclude that the reuse histogram is the *probability distribution function* of the event $(X_{N-1} = A \mid X_0 = A)$ and our proposed temporal locality is *the cumulative distribution function* of the same event.

As spatial locality, $SL(N, K)$, is a function of two parameters, the near future window size N and the neighborhood size K, we can compute the difference function of $SL(N, K)$ over either parameter and both difference functions reveal interesting insights of the access patterns.

Similar to temporal locality, the difference function of spatial locality over N, $SL(N, K) - SL(N-1, K) = P((X_{N-1} - X_{N-1} \bmod K) = (A - A \bmod K) \mid X_0 = A)$, shows the reuse distance histograms with different block sizes, K, if we follow modulo-based definition of neighborhood.

The difference function over the neighborhood size parameter, K, can be derived as follows: $SL(N, K) - SL(N, K-1) = P(|X_n - A| = K - 1,\ \exists n < N \mid X_0 = A)$. This probability function represents that how often in an address trace a stride pattern with a stride of $(K-1)$ happens in a near future window of size N. For different N and K, this function essentially provides a histogram of different strides for any given near future window size. In the context of processor caches, this difference function helps to reason about the relationship among the cache sizes, which determine the maximum reuse distance or near future window size, the block sizes, which determine the neighborhood sizes, and the performance potential of stride prefetchers.

In Fig. 5, we present the spatial locality information (Fig. 5(a)) of the benchmark, *hmmer*, and its two difference functions (Fig. 5(b)

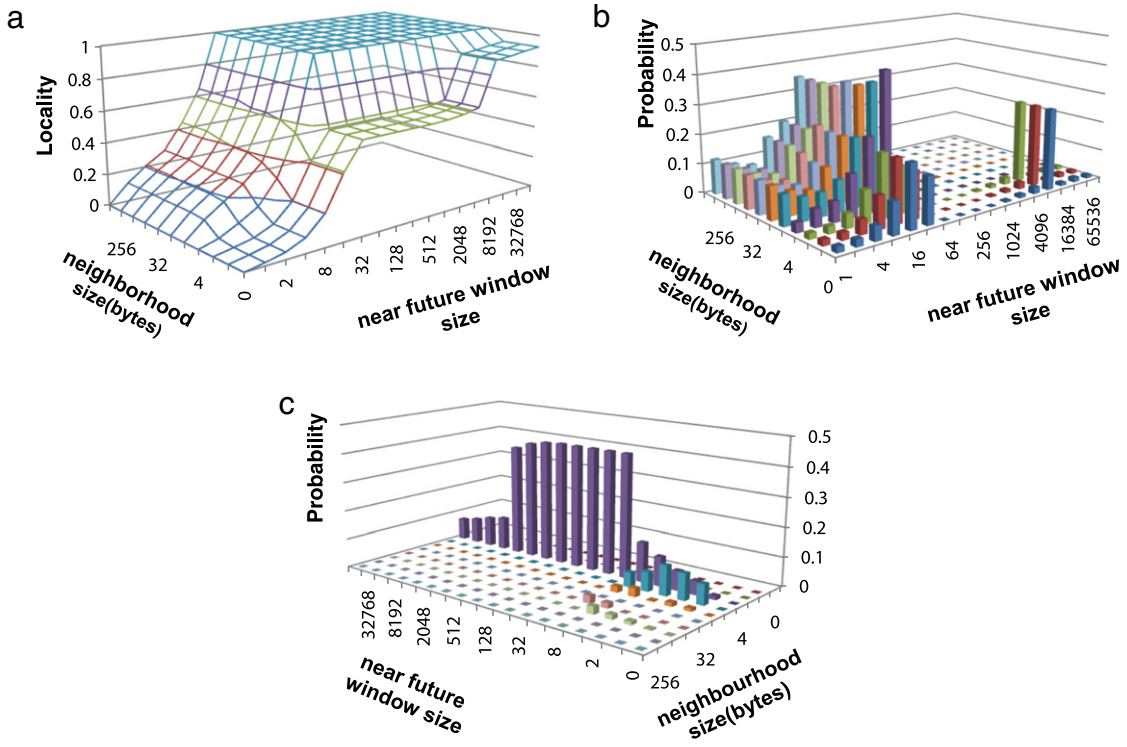


Fig. 5. Locality and the difference functions for the benchmark *hmmer*. (a) Spatial locality. (b) Difference function over the near future window size N, which is the same as reuse distance histograms. (c) Difference function over neighborhood size K, which shows a histogram of stride access patterns.

and (c)). From the Fig. 5(b), we can see how the reuse distance histograms vary for different block sizes. For small block sizes, many reuses can only be captured with long reuse distances. For large block sizes, the reuse distance becomes much smaller. This clearly indicates that one reuse histogram is not sufficient to understand the reuse patterns. Based on Fig. 5(c), we can see that for small near future windows, there are limited stride accesses. When the near future window size increases (larger than 32), the stride access pattern (stride of 8 bytes) is discovered. It implies that although the application has strong spatial locality in the forms of stride access patterns, in order to exploit such spatial locality, the cache size and set associativity need to be large enough (thereby providing the desired reuse distances) to prevent a cache block from being replaced before it is re-accessed. In addition, the single dominant stride suggests that a stride prefetcher can be an effective way to improve the performance for this benchmark.

2.5. Sub-trace locality

The locality defined in Eqs. (1) and (3) includes reuses of all the addresses in a reference stream. Therefore, it can be viewed as 'whole-trace' locality. We can add constraints to focus on certain reuses of interest, e.g., the locality of one memory access instruction or accesses to one data structure, within the address trace. We refer to such locality as sub-trace locality as not all the reuses in the trace will be considered. Note that sub-trace locality is usually not the same as the locality computed from the filtered trace, e.g., the addresses generated by one instruction, because the filtered trace will miss the constructive cross-references from other instructions. For example, consider an access sequence A(I0), A(I1), B(I0), B(I1), C(I0), C(I1) where A, B and C are different addresses while I0 and I1 are two distinctive instructions issuing these addresses. The filtered stream of I1 is A, B, C, which does not show any locality and does not account for the temporal reuses from instruction I0's accesses. The sub-trace locality is introduced to address this issue and is computed from the 'whole' trace instead of just the filtered trace. Mathematically, sub-trace spatial locality (STSL) can be defined and derived as the sum of the joint probabilities as shown in Eq. (4). Sub-trace temporal locality (STTL) can be derived similarly.

$$\begin{aligned}\text{STSL} &= P(X_n \text{ is in the neighborhood of } A,\\ &\quad \exists n < N,\ X_n \in SetX \mid X_0 = A,\ X_0 \in SetY)\\ &= \sum_{i=1}^{M} P(X_n \text{ is in the neighborhood of } A_i,\\ &\quad \exists n < N \cap X_0 = A_i \cap X_0 \in SetY \cap X_n \in SetX). \end{aligned} \tag{4}$$

From Eq. (4), it can be seen that compared to whole trace locality, sub-trace locality is modeled with the additional joint events, $(X_0 \in SetY \cap X_n \in SetX)$, in the joint probability computation. The definition of *SetX* and *SetY* determines the event of interest. For example, to compute the locality of one particular memory access instruction, we can define *SetX* as {the addresses generated by the instruction of interest} and define *SetY* as the whole sample space Ω or {addresses generated by all instructions}, meaning that for all the addresses (i.e., $(X_0 = A_i \cap X_0 \in SetY)$), we need to check whether there is a spatial reuse in the near future by the instruction of interest (X_n is in the neighborhood of A_i, $\exists n < N \cap X_n \in SetX$). If we change *SetY* to *SetX*, the equation calculates the locality of the address trace generated by the instruction of interest (i.e., the filtered trace).

We can make a slight change to Eq. (4) so that it can be computed more efficiently. Instead of looking for reuses in a near future window (defined by next-*n*-addresses), we can use a near past window, as shown in Eq. (5). This way, we can compute the locality of an instruction of interest as follows, for each address generated by the instruction (i.e., the event: $X_0 = A_i \cap X_0 \in SetX$), we search in a near past window for spatial reuse (i.e., the event: X_n is in the neighborhood of A_i, $\exists n > \ -N,\ n < 0 \cap X_n \in SetY$).

$$\begin{aligned}\text{STSL} = \sum_{i=1}^{M} P(&X_n \text{ is in the neighborhood of } A_i,\\ &\exists n > -N,\ n < 0 \cap X_0 = A_i \cap X_0 \in SetX \cap X_n \in SetY). \end{aligned} \tag{5}$$

Other than analyzing accesses from a particular PC or to a particular data structure, another use of sub-trace locality is to analyze the effectiveness of data prefetching. For example, consider the following address trace containing both demand requests (underlined addresses) and prefetch requests: $\underline{\text{A}}$, A + 32, A + 64, A + 96, A + 128, $\underline{\text{A + 64}}$, $\underline{\text{A + 128}}$, A + 160, A + 192. In this case, we can define *SetX* = {demand requests} and *SetY* = {combined demand and prefetch requests}. Then, sub-trace locality provides the information that for a demand request, how likely are there prefetch requests (or demand requests) for the same address in a near past window. The locality of either the demand request trace or the combined demand and prefetch request trace does not provide such information.

3. Locality analysis for code optimizations

In this section, we use sub-blocking/tiling to show how our proposed measure can be used to analyze and visualize the locality improvement of code optimizations.

To analyze the locality improvement of the sub-blocking/tiling optimization, we choose to use the matrix multiplication kernel $C = A \times B$ and we use the Pin tool [18] to instrument the binaries to generate the memory access traces for our locality computation. We used Def. 1 (next-*n*-address) of near future and modulo-based definition of neighborhood in our locality computation. Fig. 6 shows the locality information collected for different versions of the kernel, including untiled, tiled with the tile size of 32×32, and the tiled with the size 64×16. The matrix size is 256×256.

In the untiled matrix multiplication kernel, the matrix *A* is accessed along each row in the sequence. Therefore, it exhibits spatial locality when the neighborhood size ≥ 4 bytes, which is the size of each data element. Since the accesses to the matrices *A* and *B* are interleaved and *B* is accessed column-wise (i.e., no spatial locality at this neighborhood size), such spatial locality shows up when the near future window size ≥ 2 and the locality score is 0.5. The kernel keeps using the same row from *A* until it is done with all the columns of matrix *B*. Consequently, after every 256 accesses along the row from *A*, the same elements are re-accessed. Again, due to the interleaved accesses of *A* and *B* as well as the accesses to the product matrix *C*, the reuse distance actually is more than 512 (256 from matrix *A* + 256 from matrix *B* + 1 from matrix *C*) and hence the temporal locality appears at near future window size of 1024 in Fig. 6(a). Accesses to the matrix *B* do not repeat until $256 \times 256 \times 2 = 128k$ accesses, which is why temporal locality for the maximum near future value in Fig. 6(a) is still around 50%. Since the matrix *B* is accessed column-wise, with the near future window size of 1024 and neighborhood size of 4, accesses to the matrix *B* start to enjoy spatial locality (i.e., $B[i][j]$ and $B[i][j+1]$ show in the same window) and the locality becomes very close to 1 as shown in Fig. 6(a).

With the loop tiling optimization, both matrices *A* and *B* are divided into tiles/sub-blocks. This reduces the reuse distance of the data elements as we can see from Fig. 6(b) and (c). For a tile size of 32×32, the inner loops perform matrix multiplication of two 32×32 matrices. As a result, the sub-blocks of *A* get reused when the near future window is larger than 64 and the sub-blocks of *B* get reused when the near future window is 2048 ($=32 \times 32 \times 2$).

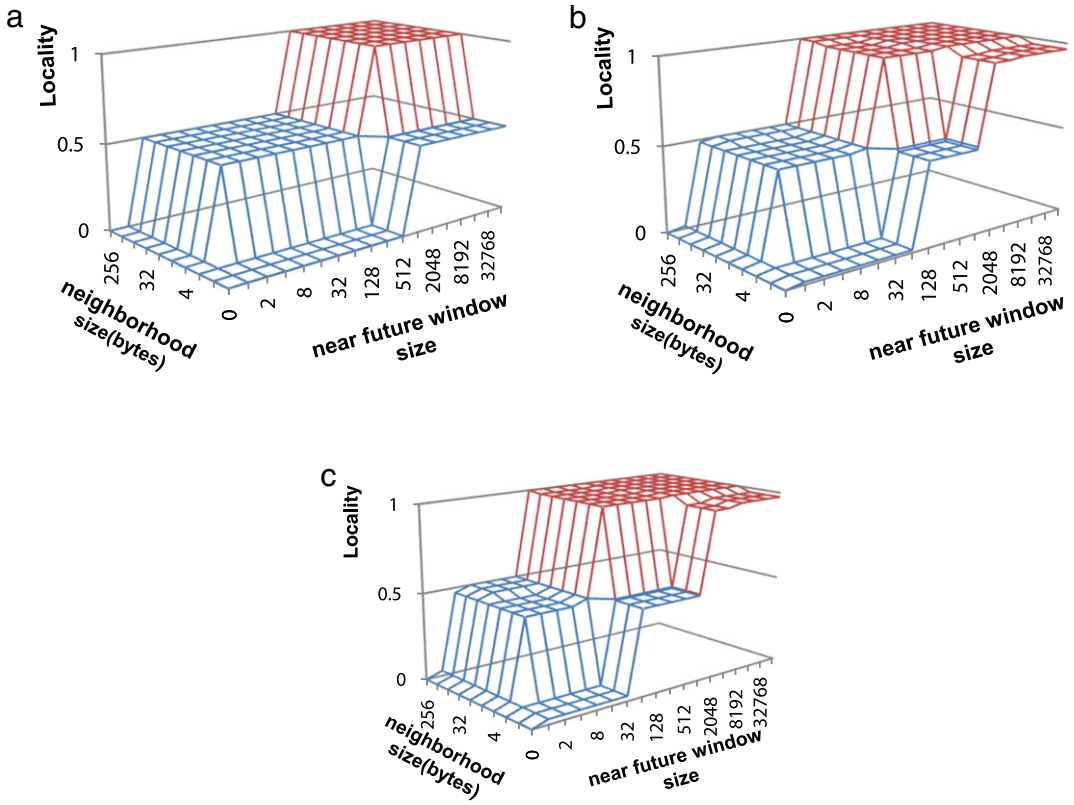


Fig. 6. Locality for different matrix multiplication kernels (a) untiled (b) 32 × 32 tile (c) 64 × 16 tile.

The sub-blocks of matrix A are accessed row-wise and show 0.5 spatial locality when the neighborhood size is 4 bytes and near future window is smaller than 64. The sub-blocks of the matrix B show spatial locality when the near future window is larger than 64, which makes $B[ii][jj]$ and $B[ii][jj+1]$ show up in the same window. The 3D locality mesh shown in Fig. 6(b) visualizes these changes in both temporal and spatial locality and confirms the observations. In the case of a tile size of 64 × 16, the locality information shown in Fig. 6(c) represents the locality behavior of matrix multiplication of a 64 × 16 matrix with a 16 × 64 matrix.

Based on the locality information shown in Fig. 6, we can easily see how much and where the locality improvement has been achieved with code optimizations. Furthermore, if we use Def. 3 (next-n-unique-block) of near future and block-based definition of neighborhood, we can also use our locality function to infer how much locality a specific cache can capture. For example, an 8 kB cache with a 16-byte block size will be able to exploit the future window up to 512 (8 kB/16 bytes) and neighborhood range of 16 bytes. Therefore, $SL(512, 16)$ will show the hit rate of this cache if it is fully associative. Mathematical models proposed in [8,16,24] can also be used to estimate miss rates for set-associative caches from the fully associative ones.

4. Locality analysis for memory hierarchy optimizations

Our proposed locality measure provides quantitative locality information at different neighborhood sizes and near future scopes. In this section, we show that it reveals interesting insights to drive memory hierarchy optimizations.

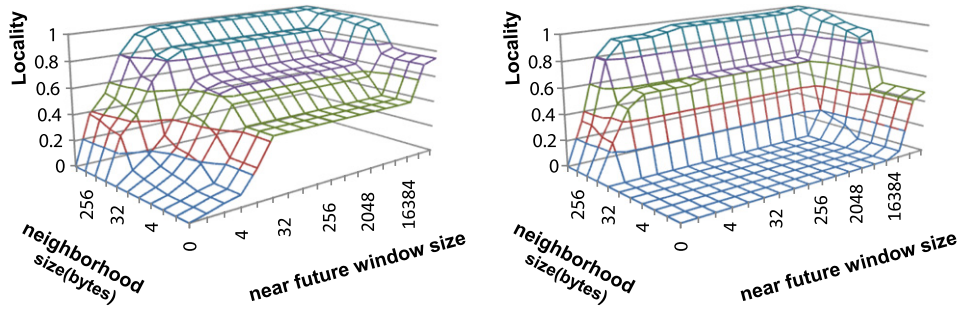
4.1. Experimental methodology

In this section, we use an in-house execution driven simulator developed based on SimpleScalar [6] for our study, except stated otherwise in Sections 4.4 and 4.6. In our simulator, we model a 4-way issue MIPS R10K style out-of-order (OOO) pipeline with a 64-entry active list. The memory hierarchy contains a 32 kB 4-way set-associative L1 data cache with the block size of 64 bytes (3 cycle hit latency), a 32 kB 2-way set associative L1 instruction cache, and a unified 8-way set associative 1 MB L2 cache (non-inclusive) with the block size of 64 bytes (30 cycle hit latency) and a banked DRAM main memory system based on the DRAMsim2 [23] framework. All the caches are non-blocking to support memory-level parallelism. The DRAM system consists of 4 GB of DDR3 modules with 8 banks (per bank row buffer size = 8 kB) and the DRAM transactions are handled on the 64-byte granularity. Address mapping used by the memory controller is as follows: rank(2-bits): row(14-bits): bank(3-bits): column(10-bits).

We include all the SPEC2000 and SPEC2006 benchmarks that we were able to compile and run for SimpleScalar ISA (PISA), 16 from SPEC2000 and 8 from SPEC2006. Benchmarks *mcf* and *bzip2* from SPEC2006 are abbreviated as *mcf*-2006 and *bzip2*-2006. For each benchmark, we use a representative 100 M Simpoint [15] for our trace collection and simulations. In our locality computation, we use the Def. 2 (next-n-unique-address) of near future and Def. 1 (modulo-based) of neighborhood.

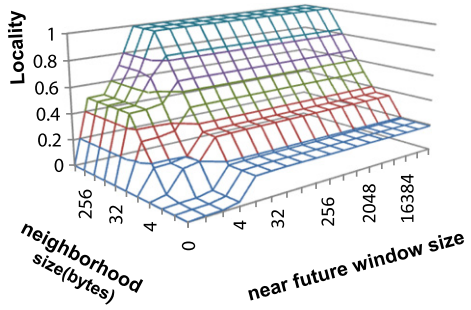
4.2. Locality at different memory hierarchy levels

In this experiment, we examine the locality of the L1 data cache access trace and the L2 cache access trace. We use the benchmarks *art, mcf* (SPEC2000) and *bzip2* (SPEC2000) as our case studies. Fig. 7 shows the locality of L1 and L2 access traces for *art, bzip2*, and *mcf*. By comparing Fig. 7(a)–(f), we can see that the L1 cache effectively exploits the spatial locality with neighborhood size less than 64 bytes ($K = 32$), which is the L1 block size. Also, the capacity of the L1 cache enables it to exploit a near future window of 512 (= L1 size/block size). As a result, at the L2 level, there is

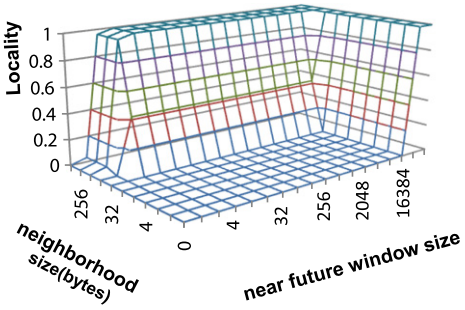


(a) *art* L1 access trace.

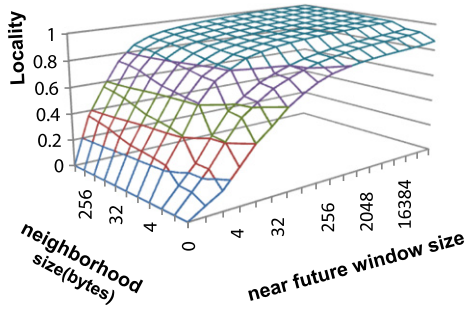
(b) *art* L2 access trace.



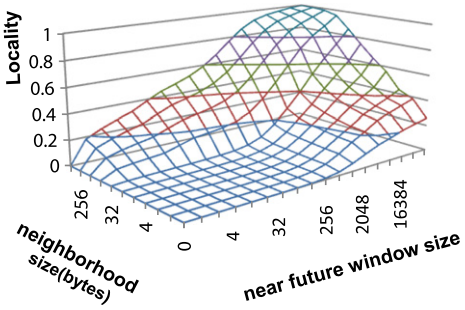
(c) *mcf* L1 access trace.



(d) *mcf* L2 access trace.



(e) *bzip2* L1 access trace.



(f) *bzip2* L2 access trace.

Fig. 7. The spatial locality of the benchmarks, *art*, *mcf* (SPEC2000), and *bzip2* (SPEC2000) at different cache levels.

limited locality present in this range. So, in order for L2 to become effective, it has to support much larger near future windows (by a large L2 cache size) and/or larger neighborhood sizes (by a large L2 block size). For the benchmark *art*, we can see from Fig. 7(a) that the locality does not increase much as we increase the near future window size up to 32 768. If we keep the L2 cache block size the same as the L1 block size, the gains of increasing the cache size are very limited until it can explore the near future window size of 32 768 (implying a L2 cache size of 32 768 $\times$ 64 B $=$ 2 MB) as shown in Fig. 7(a). While making the cache block size 128 B will significantly improve the spatial locality exploited by L2 cache. For the benchmark *mcf*, similar observations can be made from the locality curve. For the benchmark *bzip2*, the spatial locality of the L2 accesses is lower than the other two benchmarks in Fig. 7.

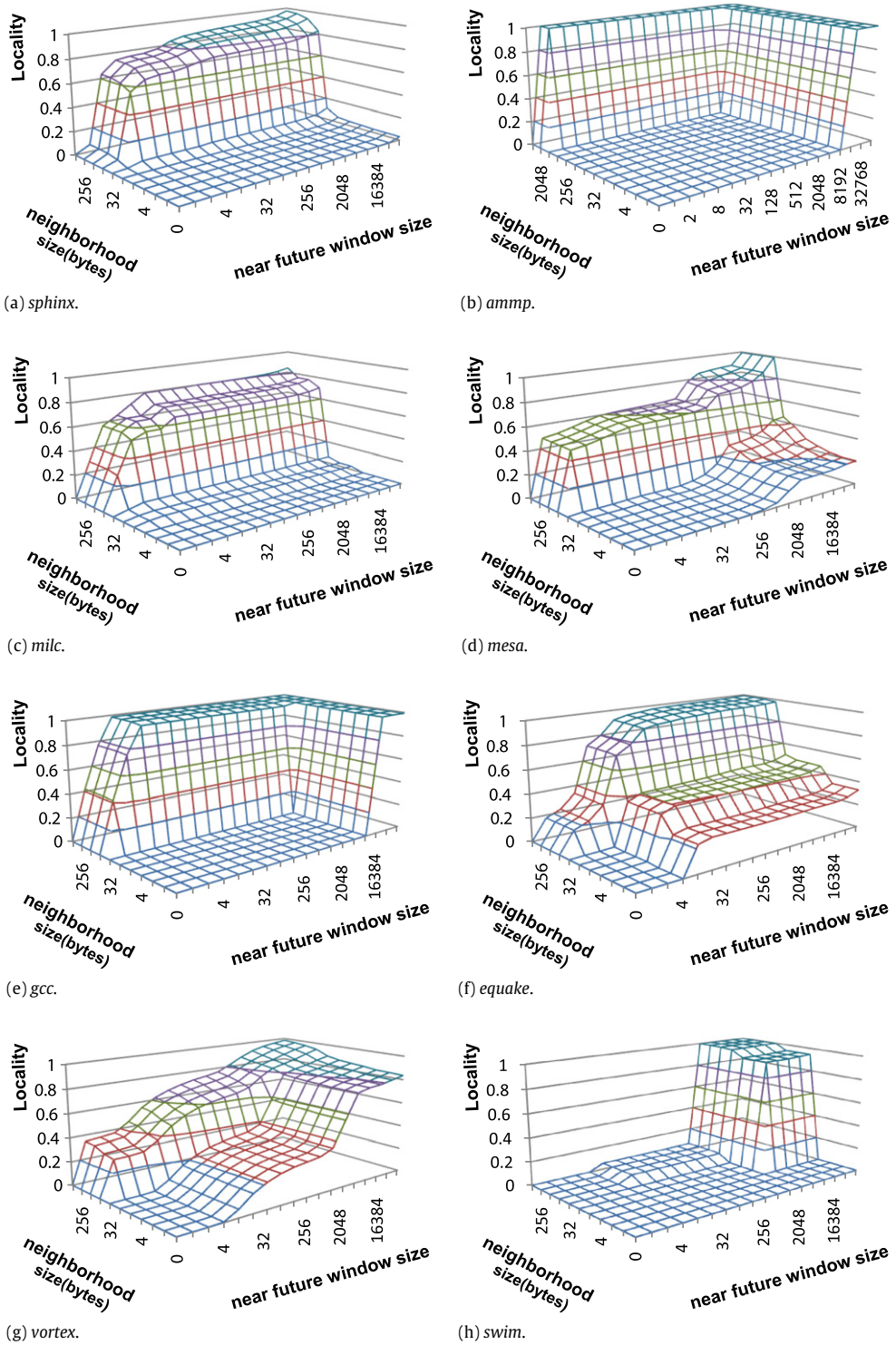
4.3. Memory hierarchy optimizations

4.3.1. Optimizing last-level caches (LLCs)

In order to improve the performance of the last-level cache, the L2 cache in our simulator, we examine the locality information of the L2 access stream. The L2 access locality of *art* and *mcf* is shown in Fig. 7(b) and (d). In Figs. 8 and 9(a), we report the locality information of L2 access streams of the SPEC benchmarks, *sphinx*, *ammp*, *milc*, *mesa*, *gcc*, *equake*, *vortex*, *swim* and *wupwise*. Two important observations can be made from the locality results. First, a common theme of all the locality information is that there exists strong spatial locality when the neighborhood size is larger than 32 bytes, corresponding to a cache block size of 64 bytes given our neighborhood definition 1. The behavior is expected as the L1 cache block size is 64 bytes.

However, in many current commercial processors, the same cache block size (e.g., 64 bytes in Intel Core i7) is used for different level of caches to simplify the cache coherence management. From the locality information shown in Figs. 8 and 9(a), we can see that many benchmarks require much larger neighborhood sizes to take advantage the spatial locality. Among these benchmarks, *wupwise*, *milc*, *mesa*, *equake*, *vortex* and *gcc* would need the neighborhood size of 64 bytes (i.e., 128-byte cache block) while *sphinx*, *mcf* and *art* exhibit additional spatial locality for increasing neighborhood sizes. The benchmark, *ammp*, requires a neighborhood size of 2048 bytes due to its streaming access pattern with a stride of more than 1024 bytes.

Second, for all these benchmarks, the spatial locality can be exploited within a small near future window (or reuse distance) when a large block is used. This indicates that a small cache with relatively large block sizes can be more effective than a large cache with small block sizes. For example, for benchmark *gcc*, $SL(4, 64) = 0.97$ (corresponding to a 4 $\times$ 128 $=$ 512 byte cache



(a) *sphinx*. (b) *ammp*.

(c) *milc*. (d) *mesa*.

(e) *gcc*. (f) *equake*.

(g) *vortex*. (h) *swim*.

Fig. 8. The spatial locality of L2 access traces for various benchmarks.

with the 128-byte block size) is higher than $SL(8192, 32) = 0.008$ (corresponding to a $8192*64 = 512$ kB cache with a 64-byte block size) and is very close to $SL(16\,384, 32) = 0.99$ (corresponding to a $16\,384*64 = 1$ MB cache with a 64-byte block size). Note that the locality scores presented here are computed based on the modulo-based definition of neighborhood. As discussed in Section 2.2 there is a correlation between the two definitions of neighborhood. When using the block-based definition, which is closely related to the cache hit rate, $SL(4, 64) = 0.491$ and $SL(8192, 32) = 0.005$. Either neighborhood definition shows, a 512-byte cache with a 128-byte block size has much higher performance than a 512 kB cache with a 64-byte block size. In addition, since most of the locality can be captured within a small near future window, it means that the LRU replacement policy will work well for caches with large blocks, thereby reducing the benefit from more advanced replacement policies (more discussion in Section 4.6). Furthermore, the limited near future window indicates that the required set associativity is small as well.

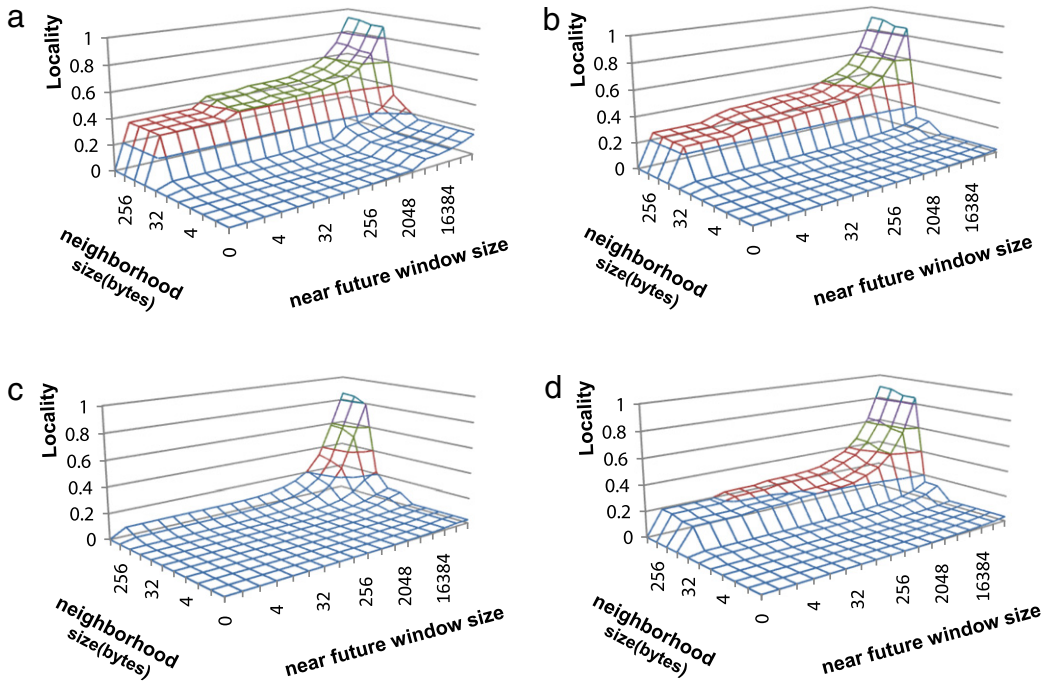


Fig. 9. The spatial locality for *wupwise* (a) L2 access trace (b) L2 miss stream for 1 MB L2 cache with a 64 B block size (c) L2 miss stream for 1 MB L2 cache with a 128 B block size (d) L2 miss stream with a stream buffer.

Different approaches can be used to achieve the effect of large-block caches. The first method is to directly increase the cache block size. The advantage is that it can reduce the required cache size as discussed above. The downside is that it may not work for all applications because different applications prefer different block sizes. The second method is to emulate the big block size by augmenting the cache with next-n-line/previous-n-line prefetching. The way to determine the value 'n' in next/previous-n-line prefetching is based on our locality measure. With the current block size of 64 bytes, if there is high spatial locality at block size of 128 bytes, e.g., *gcc*, next/previous line prefetch is used (i.e., $n = 1$). If the spatial locality shows up at a much larger block size, e.g., 2 kB for *ammp*, n is set as 2 kB/64 B $= 32$. In other words, our locality measure can be used a profiling tool to control the next/previous-n-line prefetching. The advantage of this approach is that it can emulate different cache block sizes without changing the cache configurations (i.e., cache size, set associativity, and block size). The third way is to use a stream buffer to detect stride patterns and prefetch data blocks into caches.

Next, we evaluate and compare the effectiveness of these different approaches using our locality measure with a case study of the benchmark *wupwise*. Here we use the locality measure of the L2 demand miss stream as it shows the locality that the L2 cache fails to exploit. In Fig. 9(b)–(d), we show the L2 miss locality measure for *wupwise* when the L2 cache uses a 64-byte block size, a 128-byte block size, and a stream-buffer. The next/previous line prefetch has very similar results to the cache with the 128-byte block size. From Fig. 9(b), we can see that the L2 cache with a 64-byte block size fails to exploit the spatial locality at large neighborhood sizes. In contrast, Fig. 9(c) shows that the 128-byte block cache with the same size exploits the spatial locality effectively. The stream prefetcher does not work as well for *wupwise* as the stride access patterns are not common in this benchmark (Fig. 9(d)). Our timing simulation also confirms the similar trend in performance improvement for *wupwise*: an execution-time reduction of 5% using the stream buffer while the 128-byte block size and next line prefetching achieve the execution-time reduction of 11% to 14%, respectively. Fig. 10 shows performance improvements measured in IPC (instructions per cycle) speedups for large block sizes, next/previous-n-line prefetching, and stream buffer prefetcher. Among the benchmarks, *equake, gap, hmmer, mcf (SPEC2000), sphinx, parser, perl*, and *wupwise* favor large block sizes (or next/previous-line prefetching). On average, 10.0% (14.9%) IPC improvement is achieved on average across all the benchmarks using large block sizes (next/previous-n-line prefetching). On the other hand, the benchmarks *ammp, art, gcc, gromacs, hmmer, milc*, and *swim* benefit more from the streaming buffer and it shows 15.9% performance gains on average across all the benchmarks.

Benchmark *art* experiences a slowdown when bigger block size or next/previous-n-line prefetching is used. The reason is that *art* is highly memory intensive and issues bursts of memory references. A bigger block may only provide hits at miss handling status registers but does not save much latency for a burst of accesses. This suggests that our proposed locality measure can be refined with timing information of each access. Also, the benchmark *art* has two different stride patterns with a stride of 64 bytes and 256 bytes respectively. Therefore, the 128 byte block size is not much helpful.

The next/previous-n-line prefetcher uses the locality information to determine the more suitable 'n' value to be used. For most benchmarks, we use $n = 1$ to emulate the block size of 128 bytes. For the benchmark *ammp*, as it shows a stride pattern with a stride of 2048 bytes, $n = 32$ (=2048 bytes/64 bytes) is used. For benchmarks *lbm, mcf* and *sphinx*, their locality measures show significant higher locality when the neighborhood size is increased from 128 bytes to 256 bytes. Therefore, we chose $n = 3$ for these benchmarks (i.e., 3 neighboring blocks around the missing one to emulate a 256 byte block).

4.3.2. Optimizing memory controller

Even with the spatial locality being leveraged by large blocks or stream buffers, when we examine the locality of the L2 demand miss stream, we observe that there is residual locality to be leveraged. As an example, Fig. 11(a) shows the locality information

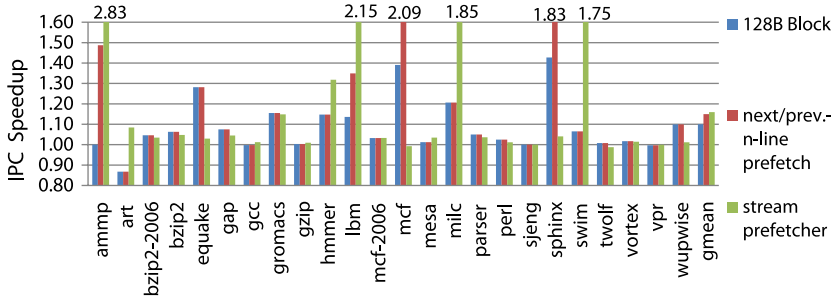
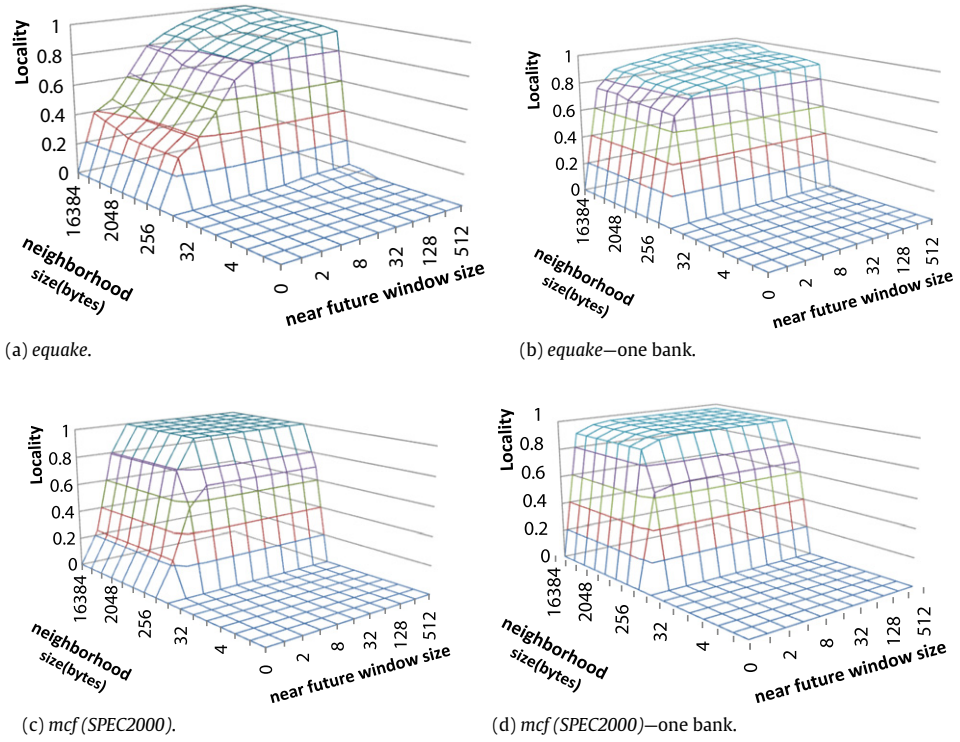


Fig. 10. Speedups of larger block sizes, next-*n*-line prefetching and stream-buffer.



(a) *equake*.

(b) *equake*—one bank.

(c) *mcf (SPEC2000)*.

(d) *mcf (SPEC2000)*—one bank.

Fig. 11. The spatial locality of L2 miss stream for the benchmarks (a) *equake*, (b) *equake* one bank, (c) *mcf* (SPEC2000), and (d) *mcf* (SPEC2000) one bank.

for the demand memory access stream of the benchmark *equake* when the L2 cache has the cache line size of 128 bytes. From the locality measure, it can be seen that there exists strong spatial locality when the neighborhood size is larger than 256 bytes ($K = 128$). Although the DRAM system can exploit some of this locality in the DRAM row-buffer, where a row-buffer hit has significantly lower latency than a row-buffer miss, our locality measure reveals further insights on how to better exploit the locality. From Fig. 11(a), we can observe that with a short near future window of 1 reference, the locality score $SL(1, 256)$ is 0.29 for *equake*. When increasing the near future window size to 4 references, the locality score $SL(16, 256)$ for *equake* is increased to 0.80. Similarly, for the benchmark *mcf*, as shown in Fig. 11(c), the locality score $SL(1, 512)$ is 0.23 and $SL(4, 512)$ is 0.99. Based on this observation, we propose to add a small buffer at the memory controller level. This buffer has large block sizes (e.g., 1–4 kB) but with just a few entries (e.g., 4–16).

Compared to the row buffer in the DRAM, which can be viewed as an off-chip single-entry buffer in each bank with a block size of 8 kB, it has two benefits: (1) a multi-entry buffer is able to exploit the significantly higher locality present beyond the near future scope of 1 reference, and (2) a buffer located at the on-chip memory controller has much lower latency than the off-chip DRAM row buffer. For the benchmarks showing no such spatial locality, we can simply power-gate this small buffer. We confirm our observations on bank-level row buffers with locality measure computed from filtered accesses to one of the banks, which is presented in Fig. 11(b) and (d) for *equake* and *mcf*, respectively. From Fig. 11(b), it can be seen that $SL(1, 8192) = 0.75$ and $SL(2, 1024) = 0.84$, which means that a multi-entry row buffer with small entry size can exploit higher locality than a single-entry large row buffer.

In our experiment, we use a 16-entry buffer with each entry caching 1024 bytes of data. A 2-cycle hit latency is assumed for this buffer, which is accessed after an L2 cache read miss. This buffer uses a write-through write-not-allocate policy for write requests.

Fig. 12 shows performance results of our timing simulations for the benchmarks which show benefits from using this buffer. From the figure, we can see that many benchmarks benefit from caching the data in a buffer at the memory controller level. The baseline system here already deploys a cache block size of 128 bytes at the L2 cache level. On average, a 16-entry buffer with a block size of 1 kB improves the performance by 8.2% across these benchmarks.

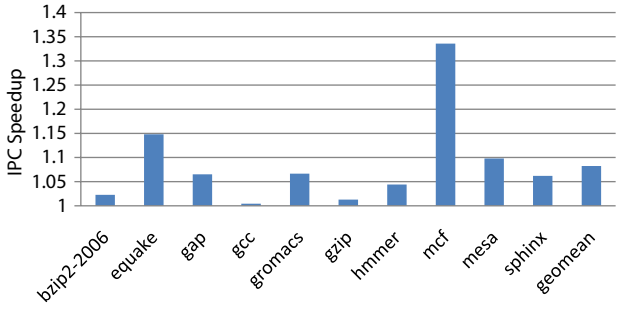


Fig. 12. Performance improvements from the proposed buffer at the memory controller level.

4.4. Locality improvement from data prefetching

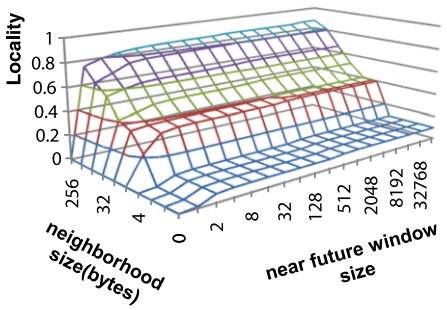
As discussed in Section 2.5, we can use our proposed sub-trace locality measure to examine the locality improvement of data prefetching mechanisms. With the locality scores present for various neighborhood sizes and near future window scopes, we can tell how well a prefetcher under study works in a wide range of cache configurations. In this experiment, we use the simulation framework for JILP Data Prefetching Contest [25] and select one of the top performing prefetchers [10] to illustrate how to interpret its effectiveness from our locality measure.

We examine two benchmarks, *mcf* and *soplex*, as either enjoys significant performance gains from the prefetcher (CPI reduced from 3.17 to 0.71 for *mcf* and from 0.61 to 0.43 for *soplex* when the L1 cache is 32 kB and L2 cache is 2 MB and the block size is 64 B) and exhibits interesting locality behavior. As the data prefetcher under study is an L1 cache prefetcher, we examine the locality improvement at both the L1 and L2 caches. For either benchmark, the locality of the L1 (demand) access stream without the prefetcher and the sub-trace locality of the L1 demand accesses within the combined demand and prefetch stream show very similar locality pattern and the improvement is limited (less than 0.05 in the locality score or 5% in L1 cache hit rate). This shows that the performance improvements are *not* from the locality improvement at the L1 cache level. In contrast, the locality at the L2 cache level is highly improved, revealing the reason for the significant performance gains. In Fig. 13, we compare the locality of the L2 demand access stream (Fig. 13(a), (c)) without the prefetcher to the sub-trace locality of the L2 access stream with the prefetcher being enabled (Fig. 13(b) and (d)) for *mcf* and *soplex*.

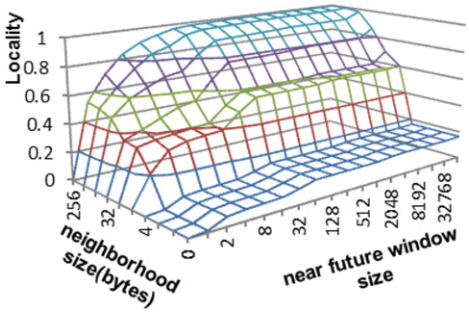
From Fig. 13, we can see that for *soplex* the locality is improved for small near future windows (from near future window = 1 onwards in Fig. 13(d)) which implies that the prefetches are issued very close to the demand accesses. For *mcf*, the locality improvement shows up for after near future window of 32. This explains why the prefetcher is more effective for *mcf* compared to *soplex*. Also, the locality improvements for a zero byte neighborhood size show that the prefetcher is able to predict the byte addresses accurately sometimes. Another interesting observation from Fig. 13(b) and (d) is that even with a quite effective data prefetcher, large neighborhood sizes still show higher locality scores. It indicates that although the data prefetcher effectively exploits regular access patterns such as strides, there remains spatial locality, which is not captured by the prefetcher but can be leveraged with large cache block sizes.

4.5. Understanding the replacement policy

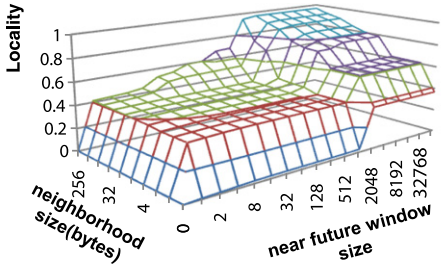
It has been well understood that some applications work well with the least recently used (LRU) replacement policy while others do not. We examine this question using our probability-based locality measure. We first focus on L1 cache access streams. Fig. 7(e) (or Fig. 2(c)) shows that with block size larger than 4 bytes, *bzip2* features a high amount of locality within small near future windows. For larger windows, the locality improvement is pretty much saturated. This implies that the L1 access stream of *bzip2* is



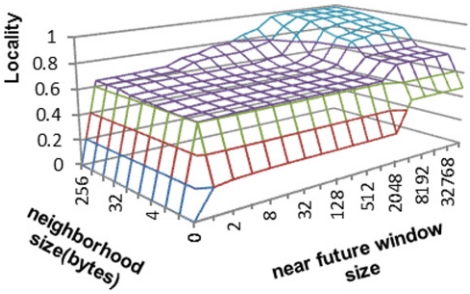
(a) *mcf*—no prefetching.



(b) *mcf*—prefetching.



(c) *soplex*—no prefetching.



(d) *soplex*—prefetching.

Fig. 13. The locality improvement of *mcf* and *soplex* at the L2 cache level (a) the locality of the L2 demand accesses of *mcf* without the prefetcher, (b) the sub-trace locality of the L2 demand trace of *mcf* with the prefetcher, (c) the locality of the L2 demand accesses of *soplex* without the prefetcher, (d) the sub-trace locality of the L2 demand trace of *soplex* with the prefetcher.

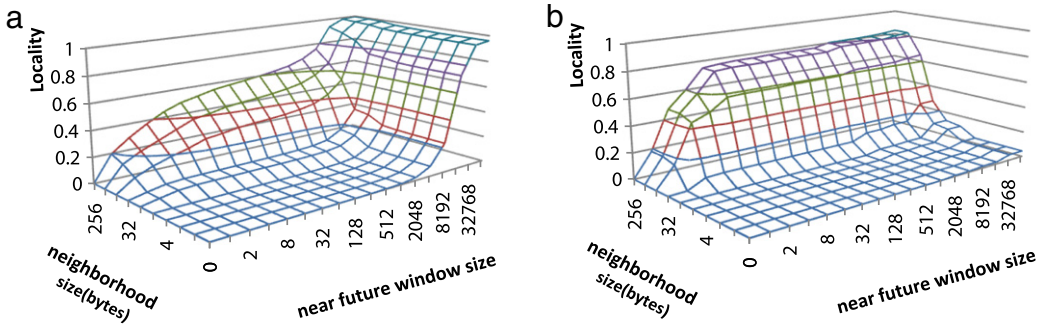


Fig. 14. Locality of the LLC access stream for benchmarks (a) *xalanbmk*, and (b) *milc*.

LRU friendly as most of the data or their neighbors will be used in a 'short/small' near future window. In comparison, if we look at the locality of the L1 accesses for *mcf (SPEC2006)*, shown in Fig. 2(a), the locality is very gradual when the block size is less than 4 bytes and there is no "knee" behavior along the X axis (the near future window size) as has been observed in *bzip2*. This shows that in *mcf*, many data reuses require long reuse distance, which is not LRU friendly as they are likely to become least recently used before they are re-accessed. Interestingly, if we look at the other dimension of the 3D mesh for *mcf*, we can see that it becomes more LRU friendly for as neighborhood size goes up (i.e., when the cache block sizes ≥ 4 bytes).

Next, we examine the L2 access streams. As discussed in Section 4.2, the L1 cache effectively exploits the locality in small near future windows and the L2 cache has to explore a larger near future window to capture the remaining locality. From Fig. 7(f), we can see that for *bzip2*, with the neighborhood size being 64 bytes ($K = 32$), once the window becomes reasonably large (e.g., 16 384, corresponding to a 1 MB $=$ 16 384 $*$ 64 B cache), there exists significant locality. Therefore, it is still LRU friendly. In contrast, the locality is not present for small near future windows for benchmarks *art* or *mcf (SPEC2000)* (Fig. 7(b) and (d)). This suggests that for *art* or *mcf*, there are many reuses at the L2 level requiring very long reuse distances. For an L2 cache with limited capacity (less than 2 MB), the LRU replacement policy does not work well since the cache is not able to sustain such large reuse distances and exhibits thrashing behavior. For such access patterns, thrash resistant replacement policies such as DIP [22] or DRRIP [17] can be a better choice. The locality plot also reveals that it gets more reuse (or becomes more LRU friendly) if the cache block size is larger than 64 bytes.

4.6. Spatial locality vs. temporal locality in last level caches

Given the importance of last-level caches (LLCs) in multi-core architectures, we have observed a resurgence of cache management research work toward LLCs, such as different intelligent cache replacement policies. In this section, we present a detailed locality analysis for LLCs, revisit these newly developed cache replacement policies, and make a case for LLCs with large block sizes but relatively simple replacement policies.

We adopt a different methodology in this section to simulate different cache replacement policies under a common framework provided for *Cache Replacement Championship (CRC)* from JWAC-1 [7]. The processor model has an 8-stage, 4-wide out-of-order pipeline with a 128-instruction scheduling window. The memory hierarchy contains 3 cache levels: a per-core private 32 kB L1 data cache (8-way set assoc. 64-byte cache line, 1-cycle hit latency) and a per-core private 32 kB L1 instruction cache (32 kB, 4-way set assoc. 64-byte cache line, 1-cycle hit latency), a per-core private 256 kB L2 cache (8-way set assoc. 64-byte cache line, 10-cycle hit latency), and a 1 MB LLC (16-way set assoc. 64-byte cache line, 30-cycle hit latency, 200-cycle miss latency). When studying a 4-core system, the LLC capacity is increased to 4 MB. Note that in this framework as well as in practice, the cache block sizes are kept the same across various cache levels because it greatly simplifies the coherence protocol among them. This, however, causes the LLC to have a limited opportunity to exploit spatial locality, as discussed in Section 4.2. As a result, the temporal locality captured by LLCs is determined mostly by their capacity since set-associative caches already reduce conflict misses by a large amount. Therefore, LLCs needs a significantly higher capacity to capture long reuse distances for them to be effective. The locality plots shown in Fig. 14 confirm this observation.

From Fig. 14(a), we can see that for the benchmark, *xalanbmk*, with the neighborhood size being 64 bytes ($K = 32$), after the window becomes reasonably large (e.g., 16 384, corresponding to a 1 MB $=$ 16 384$*$64 B cache), there exists significant locality. This implies that a high fraction of the accesses observe a reuse within the window that can be captured by the LLC and therefore, it is not thrashing in nature for near future window sizes bigger than 16 384. In contrast, for the benchmark, *milc* (Fig. 14(b)), the locality for near future windows of 16 384 and beyond remain very low when the neighborhood size is kept as 64 bytes ($K = 32$). This suggests that for LLC, there are many reuses requiring very long reuse distances, which translate into a requirement of capacity much higher than 1 MB ($=$16 384$*$64 B). As a result, the LRU replacement policy does not work well for an LLC with limited capacity. On the other hand, it is very interesting to see from the locality plot that *milc* gets more reuses (or becomes more LRU friendly) if the cache block size is larger than 64 bytes. In other words, the same access pattern can be LRU-friendly or LRU-unfriendly/thrashing depending upon the size of neighborhood exploited in the cache design.

In the next experiment, we compare the performance gains from exploiting temporal locality using intelligent replacement policies with performance gains from exploiting spatial locality using bigger cache block sizes. Fig. 15 presents the IPC speedup of the different benchmarks when they are simulated with different LLC configurations. We vary the cache block size as well as alter the cache replacement policy to see the impact of different replacement policies with different cache block sizes. On average, the more intelligent replacement policies including 3P–4P [20] and DRRIP [17] can improve the performance by 6% and 11% on average w.r.t. LRU when 64-byte cache blocks are used. In comparison, increasing the cache block size to 128-byte and 256-byte can provide IPC speedup of 10% and 37% on average, respectively using the LRU replacement policy. It is interesting to see that performance difference between intelligent replacement policies and LRU replacement policy is less than 2% when 128-byte/256-byte block sizes are used. The only exception among all the benchmarks is *xalanbmk*, for which the locality score $SL(16\,384, 32) = 0.76$ while $SL(8192, 64) = 0.54$ and $SL(4196, 128) = 0.57$, meaning that for

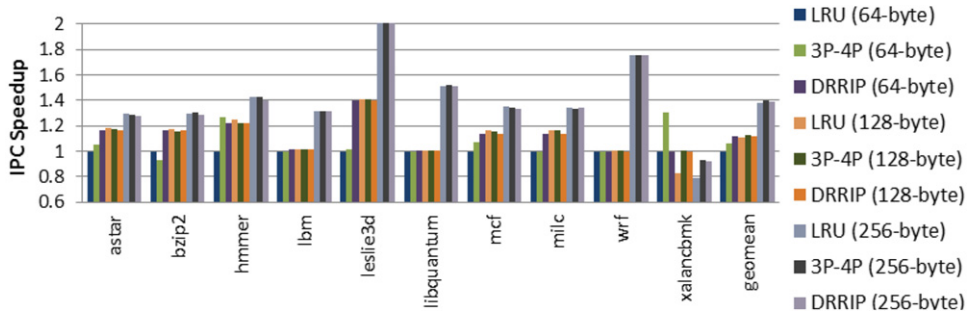


Fig. 15. IPC speedup of LRU, 3P–4P and DRRIP with different cache block sizes (1 MB LLC) over the baseline 1 MB LLC with a 64-byte block size and the LRU replacement policy.

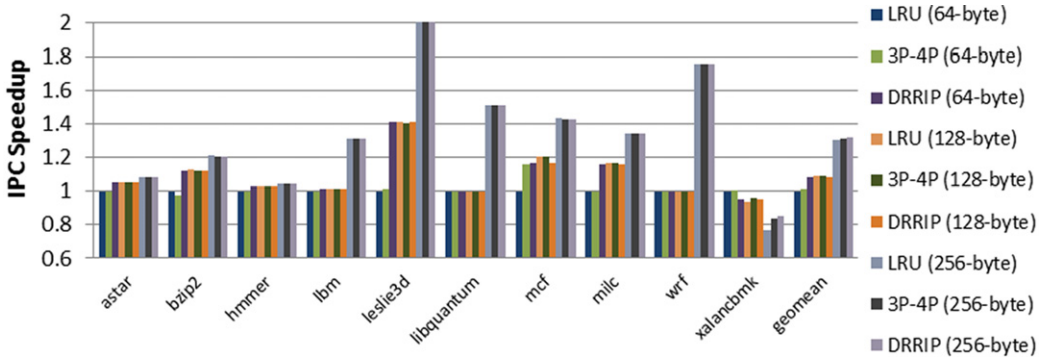


Fig. 16. IPC speedup of LRU, 3P–4P and DRRIP with different cache block sizes (2 MB LLC) over the baseline 2 MB LLC with a 64-byte block size and the LRU replacement policy.

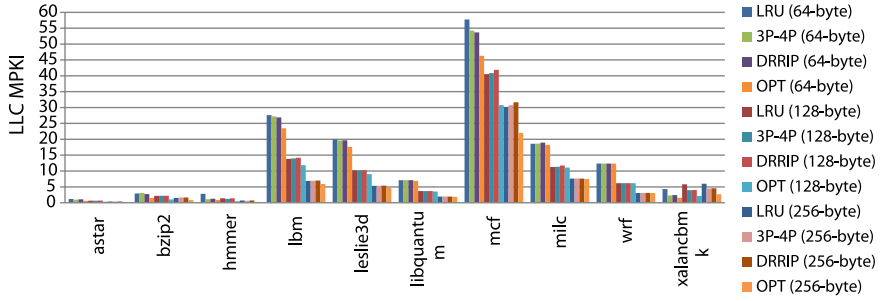


Fig. 17. LLC MPKI with LRU, 3P–4P, DRRIP and OPT replacement policies for different cache block sizes (1 MB LLC capacity).

this benchmark, increasing the block size while keeping the same overall cache capacity (1 MB) results in less locality.

Overall, our observations from locality plots are confirmed as follows: (1) locality plots suggest that the benchmarks become more LRU ffriendly when cache block sizes are increased, and (2) as the locality curve is not LRU friendly at smaller block sizes, intelligent replacement policies are effective when 64-byte cache blocks are used, and (3) with bigger cache block sizes they become LRU friendly and intelligent replacement policies render to be less effective compared to LRU. We also repeat the same experiment on a 2 MB LLC. The results are shown in Fig. 16, which shows a very similar trend to Fig. 15.

Further, it is interesting to evaluate the impact of optimal replacement algorithm (OPT) [2] on the LLC performance with different block sizes as it answers an important question on whether there is still significant room for better cache replacement algorithms in the context of bigger cache block sizes. In Fig. 17, we present the LLC miss rates, measured using misses per kilo instructions (MPKI), of different cache replacement algorithms, LRU, DRRIP, 3P–4P, and OPT, for various block sizes. From the figure, we can see that, with 128-byte and 256 byte block sizes, many benchmarks do not show any significant reduction in LLC MPKI when OPT is used compared to LRU. Only *lbm, mcf* and *xalancbmk* show significant reduction in MPKI. On average, the harmonic mean of MPKI reduction (i.e. $\text{MPKI}_{\text{LRU}} - \text{MPKI}_{\text{OPT}}$) is 0.055 for 64-byte block size, 0.043 for 128-byte block size, and 0.035 for 256-byte block size.

To get a better understanding as to why increasing the cache block size is highly effective in capturing locality for LLCs, it is important to highlight that the higher locality is not only a result from reduced compulsory misses, but also from reduced capacity misses [16]. This can be illustrated with an example access pattern. Consider the access pattern, A, A + 64, A + 128, A + 192, A, A + 64, A + 128, A + 192. For an LRU managed, fully associative cache with 64-byte blocks and capacity of 128 bytes, there are 4 compulsory misses and 4 capacity misses. The last 4 accesses are misses due to limited capacity. When the block size is 128-bytes, there are 2 compulsory misses and 2 capacity misses. In other words, both compulsory and capacity misses are reduced as a result of larger block sizes.

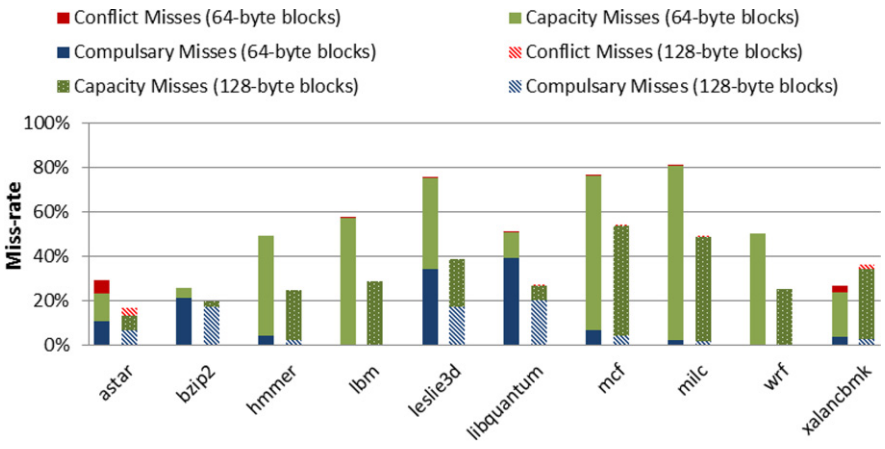


Fig. 18. Different types of cache misses for LLCs with the block size of 64 bytes vs. 128 bytes.

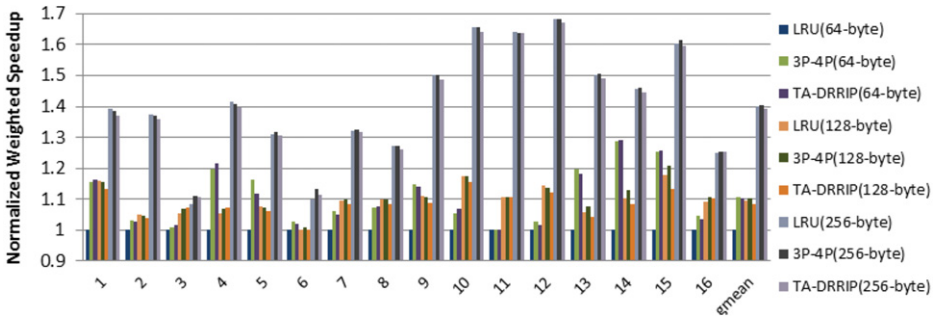


Fig. 19. Weighted speedups of LRU, 3P–4P and DRRIP with different cache block sizes (4 MB LLC) over the baseline 4 MB LLC with a 64-byte block size and the LRU replacement policy.

We also quantitatively analyze the reduction in miss-rate for each type of misses in the benchmarks. Fig. 18 presents the contribution of each type of misses to the overall miss-rate as a stacked column plot. Most benchmarks encounter a small fraction of conflict misses due to high set-associativity of LLC (16). Among the benchmarks we studied, only *bzip2* and *libquantum* have more compulsory misses as compared to capacity misses. All other benchmarks have high amount of capacity misses. When the cache block size is increased from 64-byte to 128-byte we see a reduction in compulsory misses as well as capacity misses for all the benchmarks except *xalancbmk*. For *xalancbmk*, although the compulsory misses are reduced, its capacity misses are increased due to cache pollution.

As the results shown in Figs. 15 and 16 are based on a single workload utilizing the LLC, in the next experiment, we evaluate the performance gains of exploiting temporal locality using intelligent replacement policies with performance gains of exploiting spatial locality using bigger cache block sizes when LLCs are shared among multiple cores. Here, for our multi-core evaluations, we create 16 multi-programmed workloads by randomly picking 4 benchmarks to combine in mixes. These 4-way multi-programmed workloads are executed on a 4-core configuration with a shared L3 cache of 4 MB capacity. We use instructions per cycle (IPC) for comparing performance in single-core scenario and weighted IPC speedup, i.e. $\sum_i(\text{IPC}_i^{\text{shared}}/\text{IPC}_i^{\text{alone}})$, for comparing the performance in multi-core scenario. This methodology of evaluation is adopted from the 1st JILP-CRC. The results are shown in Fig. 19.

From Fig. 19, we observe that for LLCs shared among multi-programmed workloads, 3P–4P and TA-DRRIP can provide good speedups when the LLC block size is 64-byte. When the cache block size is 128 or 256 bytes, in contrast, they have similar performance to the LRU replacement policy on average. Therefore, these results are consistent with those shown in Figs. 15 and 16 and show that intelligent replacement policies are only effective when LLC block size is same as other cache level and they do not provide any significant improvement over LRU when bigger cache block sizes are used to exploit spatial locality.

5. A GPU-based parallel algorithm for locality computation

From the discussion in Sections 3 and 4, we show that our probability-based locality provides a useful way to analyze reference patterns and drive memory hierarchy designs. However it requires a significant amount of computation time, particularly for a large trace as we need to move a near future window throughout the trace. As derived in Section 2.4, our temporal locality essentially represents the same information as reuse distance histograms (cumulative distribution function vs. probability distribution function). Although we can leverage the previous works on fast computation of reuse distances [1], we aim to reduce the computation time by an order of magnitude to make it practical for compiler or runtime profile analysis. Based on the inherent data-level parallelism of our locality computation, we resort to parallel computation on graphics processing units (GPUs).

Our parallel algorithm for locality computation of an address trace $A[0 : S - 1]$ is outlined as follows.

1. Each thread will be responsible for the locality information of $A[\textit{thread id}]$. More specifically, it calculates the joint probability $P(X_n$ is in the neighborhood of $A[\textit{thread id}],\ \exists n < N \cap X_0 =$

A[*thread id*]) for a specific neighborhood size K. It does so by comparing *A*[*thread id*] with *A*[*thread id* + 1], *A*[*thread id* + 2], *A*[*thread id*+3], etc., until it exceeds the maximal near future window size or finds an address meeting the requirement of neighborhood. We keep W counters in each thread for W different near future window sizes. When an address in its neighborhood is found, the distance to *A*[*thread id*] is determined. If the distance is smaller than a near future window size, the corresponding counter is incremented.
2. After we obtain the locality information of each address, we need to combine them to generate the locality information of the trace. Since we have W counters in each thread, we will perform W reductions across all threads to accumulate them. After reduction, those accumulated counters are divided by S to generate the data points of the locality function, i.e., $SL(1, K), SL(2, K), SL(4, K), \ldots, SL(2^W, K)$.
3. Repeat steps 1 and 2 for a different neighborhood size K.

We implement this algorithm using CUDA [21]. In our implementation, both steps 1 and 2 are optimized by utilizing GPU's on-chip shared memory. We run our GPU algorithm on a NVIDIA GTX480 GPU and compare with our sequential CPU implementation on an Intel Core i7 920 processor. We compute locality with the near future window size ranging from 1 to 2^{16} addresses (exponential scale) and we vary the neighborhood size from 0 to 512 bytes (exponential scale). For address traces with the length of 1 million to 16 million addresses, our GPU implementation takes less than 4 s and achieves 30× to 33× speedups over our CPU implementation. In our experiments, we run 100 million instructions from each benchmark to generate the trace.

6. Related work

Given its importance, the principle of locality has been extensively studied. However, most previous works on quantifying locality, spatial locality in particular, are based on intuitive notions or heuristics and there is a lack for formal quantitative definition of locality.

For temporal locality, reuse distances or LRU stack distances [19] are used to determine how far in the future a temporal reuse is going to happen. The histogram of reuse distances is used as a signature to quantify the temporal locality of a trace [28]. StatCache [3] provides an efficient way to estimate reuse distance histograms through sampling. Section 2.4 derives the relationship between our probability-based measures and reuse histograms and provides a theoretical justification for reuse histograms. Based on this relationship, previous works on reuse distances such as component-based locality analysis [11,27] can also be adapted to our proposed measure. In [12], mass-count disparity is used to show that most reuses are from accessing a small number of addresses.

For spatial locality, Bunt et al. [5] assume that the references can be fit to a Brad–Zipf distribution and the parameters of such a distribution are used to generate a spatial locality score. Weinberg et al. [26] propose to compute a histogram of different strides within a window of previously accessed addresses and then combine the histogram with strides into a single score for spatial locality. Berg et al. [4] and Gu et al. [14] propose to quantify the spatial locality effect using the changes in the miss rates or the reuse histograms when the block size is increased or doubled.

Using a 3D surface to visualize locality is proposed in [13] and it is recognized that the temporal locality is a special case of spatial locality. The locality measure in is based on an averaged probability that the next access is within a stride of the current location within a near future window, which is fundamentally different from our proposed conditional probability.

Compared to previous attempts on quantifying temporal or spatial locality, the advantages of our proposed approach include: (1) it provides a numeric measure with a clear mathematical meaning (i.e., conditional probability); (2) it offers a unified quantification for both temporal and spatial locality; and (3) it does not only present a single locality score for a specific cache configuration but also shows the trend of how locality vary when we change the near future window and neighborhood size so that we can better understand the nature of the program locality.

7. Conclusions

The locality principle describes the phenomenon that after a reference to a datum, the same datum or its neighbors are likely to be referenced in near future. This paper addresses the question 'how likely'. The key contribution of the paper includes (a) a formal definition of locality as a conditional probability and derivation of how it can be computed using joint probabilities, which can in turn be computed from address traces, (b) the derivation of difference functions of the locality function and the justification for reuse histograms as a quantitative measure of temporal locality, (c) the formal definition of sub-trace locality to focus on certain reuses of interest, and (d) various locality analysis using our proposed measure, revealing the following insights (1) how locality can be exploited at different levels of memory hierarchy and how the locality curves drive various memory optimizations; (2) the information of the working set: instead of the common way of using the knee of the miss rate curve over cache sizes, the working set is represented as a contour in a 3D surface showing the same conditional probability can be exploited using different neighborhood sizes and near future scopes; (3) an evaluation of memory optimizations such as data prefetcher over a wide range of cache configurations; and (4) a detailed locality analysis for LLCs to make a case for LLCs with large block sizes and relatively simple replacement policies.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments to improve our paper. This research is supported by an NSF grant CNS-1004945, an NSF CAREER award CCF-0968667 and a research fund from Intel Corporation.

References

[1] G. Almasi, C. Cascaval, D. Padua, Calculating stack distances efficiently, in: Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance, Berlin, Germany, June 2002.
[2] L.A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal 5 (2) (1966) 78–101.
[3] E. Berg, E. Hagersten, Statcache: a probabilistic approach to efficient and accurate data locality analysis, in: IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2004, pp. 20–27.
[4] E. Berg, E. Hagersten, Fast data-locality profiling of native execution, in: Intl. Conf. on Measurement and Modeling of Computer Systems, SIGMETRICS, 2005, pp. 169–180.
[5] R. Bunt, J. Murphy, The measurement of locality and the behavior of programs, The Computer Journal 27 (3) (1984) 238–253.
[6] D. Burger, T.M. Austin, The simplescalar tool set version 2.0, Technical Report, Computer Science Department, University of Wisconsin–Madison, 1997.
[7] Cache Replacement Championship 1st JILP Workshop on Computer Architecture Competitions, JWAC-1, 2010. http://www.jilp.org/jwac-1/.
[8] C. Caşcaval, D. Padua, Estimating cache misses and locality using stack distances, in: Proceeding of the 17th Annual International Conference on Supercomputing, ICS, 2003, pp. 150–159.
[9] P. Denning, The locality principle, Communications of the ACM 48 (7) (2005) 19–24.
[10] M. Dimitrov, Huiyang Zhou, Combining local and global history for high performance data prefetching, in: The JILP Data Prefetching Championship, 2009.

[11] C. Ding, Y. Zhong, Predicting whole-program locality with reuse distance analysis, in: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI, pp. 245—257.
[12] Y. Etsion, D.G. Feitelson, L1 cache filtering through random selection of memory references, in: Proceeding of International Conference on Parallel Architecture and Compilation Techniques, PACT, 2007, pp. 235–244.
[13] K. Grimsrud, J. Archibald, Locality as a visualization tool, IEEE Transactions on Computers 45 (11) (1996) 1319–1326.
[14] X. Gu, I. Christopher, T. Bai, C. Zhang, C. Ding, A component model of spatial locality, in: Proceedings of the 2009 International Symposium on Memory Management, ISMM, 2009, pp. 99–108.
[15] G. Hamerly, E. Perelman, J. Lau, B. Calder, SimPoint 3.0: faster and more flexible program analysis, in: Proceedings of the Workshop on Modeling, Benchmarking and Simulation, MoBS, 2005.
[16] M. Hill, A.J. Smith, Evaluating associativity in CPU caches, IEEE Transactions on Computers 38 (12) (1989) 1612–1630.
[17] Aamer Jaleel, Kevin Theobald, Simon C. Steely Jr., Joel Emer, High performance cache replacement using re-reference interval prediction, RRIP, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA, 2010, pp. 60–71.
[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: Proceedings of PLDI 2005, Chicago, Illinois, USA, June 2005, pp. 191–200.
[19] R. Mattson, J. Gecsei, D. Slutz, I. Traiger, Evaluation techniques and storage hierarchies, IBM Systems Journal 9 (1970) 78–117.
[20] P. Michaud, The 3P and 4P Cache replacement policies, in; Proceedings of 1st JILP Workshop on Computer Architecture Competitions, JWAC-1, 2010.
[21] NVIDIA CUDA Programming Guide, Version 2.2, 2009.
[22] M. Qureshi, Yale N. Patt, Adaptive insertion policies for high performance caching, in: Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA, 2007, pp. 381–391.
[23] P. Rosenfeld, E. Cooper-Balis, B. Jacob, Dramsim2: a cycle accurate memory system simulator, Computer Architecture Letters 10 (1) (2011) 16–19.
[24] J. Smith, A comparative study of set associative memory mapping algorithms and their use for cache and main memory, IEEE Transactions on Software Engineering SE-4 (2) (1978) 121–130.
[25] The 1st JILP Data Prefetching Championship, DPC-1, 2009. http://www.jilp.org/dpc.
[26] J. Weinberg, M.O. McCracken, E. Strohmaier, A. Snavely, Quantifying locality in the memory access patterns of HPC applications, in: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, SC, 2005, p. 50.
[27] Y. Zhong, S.G. Dropsho, X. Shen, A. Studer, C. Ding, Miss rate prediction across program inputs and cache configurations, IEEE Transactions on Computers 56 (3) (2007) 328–343.
[28] Y. Zhong, X. Shen, C. Ding, Program locality analysis using reuse distance, ACM Transactions on Programming Languages and Systems (TOPLAS) 31 (6) (2009) Article 20.

Saurabh Gupta is a Ph.D. candidate in the department of Electrical and Computer Engineering at North Carolina State University. He got his Masters and Bachelor's degrees (dual-degree program) in Electrical Engineering from the Indian Institute of Technology Kanpur in 2009. His research interests include processor microarchitecture, cache architecture and memory hierarchy optimizations.

Ping Xiang is a Ph.D. student in the department of Electrical and Computer Engineering at North Carolina State University. He got his M.S. from the University of Central Florida in 2010. His research interests include architectural support for high performance microarchitecture, many-core architecture, GPGPU, and reliability.

Yi Yang is a Ph.D. candidate in the department of Electrical and Computer Engineering at North Carolina State University. He got his M.S. from Institute of Computing Technology, Chinese Academy of Sciences and B.S. from the University of Science and Technology of China. His research interests include high performance computing, architectural support and compiler optimization for many-core architecture. His current emphasis is on designing software interfaces to utilize heterogeneous architectures.

Huiyang Zhou received his bachelor's degree in electrical engineering from Xian Jiaotong University, China, in 1992 and his Ph.D. in computer engineering from North Carolina State University in 2003. He is currently an associate professor in the Department of Electrical and Computer Engineering at North Carolina State University. Between 2003 and 2009, he was an assistant professor at the School of Electrical Engineering and Computer Science, University of Central Florida. His research focuses on high performance microarchitecture, low-power design, GPU Computing (General Purpose computing on Graphics Processing Units or GPGPU), architecture support for system dependability, and backend compiler optimization. He is a recipient of the NSF CAREER award and a senior member of the ACM and IEEE.