A Composable Model for Analyzing Locality of Multi-threaded Programs *

Chen Ding‡ and Trishul Chilimbi†

†Microsoft Research, Microsoft Corporation
‡Computer Science Department, University of Rochester

ABSTRACT

In a multi-threaded execution, threads may negatively interfere when their private data contends for shared cache or positively interact when the data brought in by one thread is used by other threads. This paper presents a model of such cache behavior to predict locality without exhaustive simulation and provide insight into trends. The new model extends prior work that assumes no data sharing and uniform thread interleaving. Based on a single pass over an interleaved execution trace, we compute a set of per-thread statistics that includes the effect of thread interleaving and data sharing. The per-thread statistics is then composed to predict performance for all cache sizes, either for sub-clusters of threads or for futuristic environments with a larger number of similar threads.

We evaluate and validate our model against exhaustive simulation using a server application running on a quad-core machine and productivity, multimedia and gaming applications running on a dual-core machine. The results indicate that our model is accurate and relies on incorporating both irregular thread interleaving and data sharing to achieve this accuracy. In addition, it identifies and separates individual factors affecting locality and scalability and hence opens new possibilities in performance tuning, program scheduling, and hardware cache design for concurrent applications.

1. INTRODUCTION

The hardware industry has embraced multi-core processors and future machines will likely include an increasing number of processor cores. This will result in multi-threaded applications becoming more commonplace and increase the performance impact of locality as the memory system and shared on-chip caches serve data to a large number of processor cores. Recent use of in-order cores by processors such as Larabee [33] makes cache simulation a viable strategy for modeling program locality and performance. An accurate locality model for multi-threaded applications that quantifies how concurrent threads interact with the memory hierarchy and how their data usage affects the efficiency and scalability of the system would be very useful in evaluating software and hardware design decisions and improving scheduling at the application, operating system, virtual machine, and hardware levels.

Locality is defined by active data usage. For a memory access to data block a, the locality of the access depends on the amount of data that has been accessed since the last reference to a. This metric is known as reuse distance or LRU stack distance. In a concurrent environment, the locality of one thread is affected by the data access in other threads. In the absence of data sharing, the effect is purely negative. The reuse distance of an access by one thread is inflated by data access from concurrent threads. This effect is first modeled by Suh et al. [35] for time-shared execution and Chandra et al. [8] for interleaved execution. The two studies considered multiple instances of independent applications running concurrently rather than a single multi-threaded application.

In this paper, we present a locality model for multi-threaded code. We use a similar model as Suh et al. [35] and Chandra et al. [8] but extend it with two components necessary for multi-threaded applications: thread interleaving and data sharing. A multi-threaded application may have internal dependences that cause non-uniform interleaving among threads. It is possible that not all threads are active at all times. A model must consider the precise interleaving in order to precisely model thread interaction.

The second issue is shared data. It has two effects on locality. First, the amount of data access by multiple threads is not purely additive if some of the data accessed is the same. Second, there is the benefit of an *intercept*, in which a thread can avert a cache miss if the data block it is accessing has just been brought in by another thread. An intercept is a phenomenon unique for a multi-threaded execution. Because of it, parallel execution can have a positive effect on locality in that a reuse distance in a thread may be *shortened* when executing with other threads.

Models of data sharing have been devised for scientific

*The primary pieces of this work were developed at Microsoft Research, Redmond, where the first author worked as a visiting researcher.

applications [23, 28], divide-and-conquer algorithms [9], and cache-coherence machines [4, 15, 17]. These models analyze the identity, the size, and the frequency of shared data but not the locality. Our model focuses on the locality and its allied factors: the locality of individual threads, the amount of data sharing, and the degree of interleaving. The individual locality and mutual interleaving affect the amount of data sharing, which in turn affects the probability of intercepts and the aggregate locality.

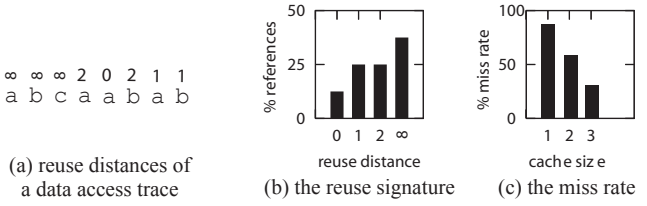
Our model is trace based. It computes a set of per-thread metrics in a single pass over a concurrent execution. The per-thread model contains statistics of locality, data sharing, and interleaving. For a system of p threads, the model can compute all $O(2^p)$ sharing relations by composing per-thread statistics without having to traverse the trace again. The composibility of our model allows us to easily analyze concurrent executions that only involve a subset of application threads or executions that include a larger number of similar application threads. In addition, to computing miss rates for a shared cache, we briefly describe an extension that permits modeling of coherence misses in the interleaved execution on a partitioned cache.

The rest of the paper is organized in accordance with our main contributions.

- A composable per-thread data sharing model that is scalable and allows investigating concurrent executions with a smaller or larger number of similar threads (Sections 5.1 and 5.2)
- A model for irregular thread interleaving, which is integrated with the data sharing model (Section 4.3).
- Evaluating and validating the two new models using four real-world multi-threaded applications and showing how the model can be used to answer interesting questions about application locality and scalability (Section 6).

2. BACKGROUND

For each memory access, the *LRU stack distance* or *reuse distance* is the number of distinct data elements accessed between this and the previous access to the same datum. The locality of an execution is given by the distribution of all finite reuse distances, which we call the *locality signature*. Locality signature gives the capacity miss rate in caches of all sizes [30]. It can also be used to estimate the effect of cache associativity [21, 29, 34]. As an illustration, Figure 1 shows an example trace and its reuse distances in (a), the locality signature in (b), and the capacity miss rate of all cache sizes in (c).



(a) reuse distances of a data access trace (b) the reuse signature (c) the miss rate

Figure 1: Data access, reuse distance, locality signature, and miss rate

The locality signature is represented by a histogram $< R, P >=< r_1 r_2 \ldots r_n, p_1 p_2 \ldots p_n >$, where r_i is a series of lengths and p_i is the portion of reuse distances whose length is r_i. For a compact representation, we let r_i be ranges of base-2 logarithmic scale. A locality signature corresponds a probability function $p(r)$ in statistics. For each range r, $p(r)$ is the portion of reuse distances whose length fall in r. If we pick a memory access at random, with probability p_i its reuse distance is in interval r_i.

The basic problem of locality modeling is to identify relevant factors to predict the locality signature, from which we can predict the miss rate for cache of all sizes. In a multi-threaded execution, we need to distinguish two types of single-thread locality. The first is given by the reuse distances of its memory references when they are not interleaved with the actions of other threads. We call this case *private reuse distance* and *private locality (signature)* since they correspond to running the thread using a private cache (without invalidation). The second is given by the reuse distances of its memory references when they are interleaved in a concurrent execution. We call this case *shared reuse distance* and *shared locality (signature)*. The problem we address is to predict shared locality for each thread for some combination of threads. The overall locality is simply a sum of the shared locality from all participants.

3. OVERVIEW OF THE NEW MODEL

The new model collects per-thread statistics that includes five components. The first is the private locality signature. The second is the reuse time map, which converts a reuse distance to the reuse time window. The third is footprint, which is the size of data access over a time window. These first three are collected from only the memory references of a single thread. The next two, thread interleaving and data sharing, are estimated from the interleaved trace. These components are composed to produce an estimate of the shared locality for each thread.

Figure 2 shows more quantitatively how a private reuse distance d in thread A is converted to its shared counterpart d' when A is run with thread B. Roughly

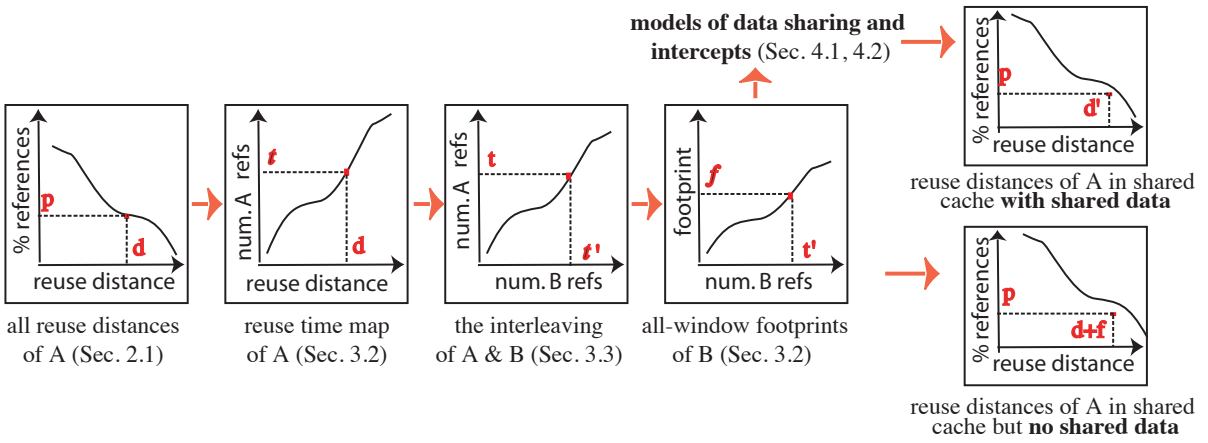


Figure 2: The steps of predicting the change in reuse distance d of thread A due to parallel execution with thread B: (1) finds the reuse time window t of d using the reuse time map of A, (2) finding the coinciding time window t' of thread B, (3) finding the size of data f accessed in t'. If A, B share no data, d' is $d+f$; otherwise, d' is computed using the data-sharing model.

the process is as follows. First it finds the reuse time window t of the reuse distance d using the reuse time map of A. Second, it finds the coinciding time window t' of thread B, using the interleaving model. Third, it finds the size of data f accessed in t', using the footprint of B. If the two threads share no data, the shared reuse distance is $d + f$. When they share data, the shared reuse distance d' is computed using the data-sharing model.

Figure 2 also indicates in parentheses the sections in which the referenced terms and metrics are defined and described. As suggested in the figure, these metrics are probability functions related to each other by composition.

Of the five components, four (other than the locality signature) are added to model a parallel environment. Two of the four, the interleaving and the data sharing components, are unique in this work for modeling multi-threaded applications. The reuse-time map and footprint map have been used by previous studies [8,35]. We will use a more precise representation for the former and a different measurement method for the latter.

Limitations. The model is based on traces and designed for off-line program characterization and performance evaluation rather than on-line performance optimization. For performance valuation, the model predicts the miss rate. The effect of a miss can be precisely calculated on in-order processors such as Larabee [33], although the effect is less direct on out-of-order issue processors. For program characterization, a single trace contains limited information. It does not show how the parallel behavior changes with another program input. Neither does it show behavior variations due to scheduling and other non-deterministic factors in a parallel environment. However, we believe that this model is a useful first step. By extracting the characteristics of data sharing and thread interleaving into an abstract form, we can compare different executions by their model parameters rather than their trace sequences. An accurate modeling technique for a single trace can enable further analysis of the effect of a program input or a system environment. Similarly, it can support phase analysis and show the behavior variation over time. In all these cases, we have to identify the minimal set of statistics necessary to characterize a multi-threaded execution.

4. THREADS WITH NO SHARED DATA

In this section, we address the case where threads access a shared cache but do not share data. We extend prior work, which assumes uniform thread interleaving, to account for general types of interleaving.

4.1 The Model Used in Previous Work

We use an example to introduce the previous model [8, 35] and its basic components. The left portion of Figure 3 shows the data access trace for two threads separately. The right portion shows two types of thread interleaving. One is uniform and the other is not. The private reuse distance of the second access of b in Thread 1 changes from $R_1 = 4$ to $R'_1 = 9$ when the two threads have uniform interleaving. We have a different result $R'_1 = 8$ with the non-uniform interleaving. Next we will describe first how the uniform interleaving is modeled in the previous work and then how the general interleaving is modeled in this work.

The previous models used two metrics to compute the shared reuse distance R'_1. The first is the time win-

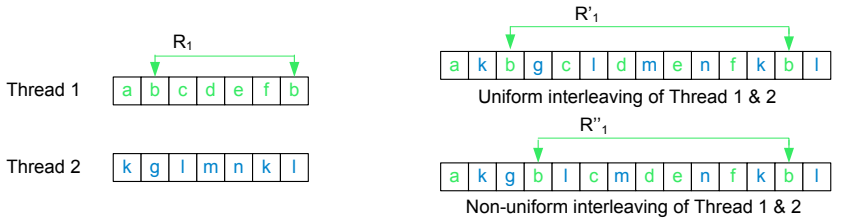


Figure 3: Shared reuse distance depends on thread interleaving

dow $T(R_1)$ in Thread 1, which we call the *reuse time* of R_1. Under the assumption of uniform interleaving, the length of the coinciding time window in Thread 2 is computed as $|T_2| = \frac{N_2}{N_1}|T_1|$, where N_1 and N_2 are the length of the trace of the two threads. Thread 2's accesses in T_2 affect the shared reuse distance R'_1. The second metric is the number of distinct data accessed in T_2, which we call the *footprint* of T_2 and denote as $F(T_2)$. The new reuse distance is $|R'_1| = |R_1| + F(\frac{N_2}{N_1}T(|R_1|))$.

When there are k threads, the new reuse distance R'_1 of Thread i is the original R_1 plus the footprint from all other threads in the overlapping window, as shown by the following equation.

$$|R'| = |R| + \sum_{p=1...k, p\neq i} F_p(\frac{N_p}{N_i}T_i(|R|))$$

where T_i is the reuse time map of Thread i, and F_p is the footprint map of Thread p ($p \neq i$). T_i is used to compute the overlapping window of R in other threads, and F_p show the footprint in a time window. Because of uniform interleaving, the equation uses a constant ratio to convert the length of the R time window in Thread i to the length of time window in another thread. The shared locality of Thread i is obtained by converting all private reuse distances of i and then combining the results.

While uniform interleaving is expected in situations where independent applications run continuously in parallel, the assumption does not always hold for a threaded execution, especially for one with asymmetrical threads where some threads run occasionally but not constantly. Next we explain the metrics we use in the basic conversion and then describe the extension for general thread interleaving.

4.2 Distributions, Maps, and Sampling

For efficiency reasons, we cannot individually represent reuse distances, time windows, footprints, and their relations. Instead, we represent them collectively as statistical distributions and their mappings. The locality signature $< R, P >$ is a distribution, where R is a series of intervals, $r_i = [d_i, d_{i+1})$, and P is a series of probabilities p_i. It says that p_i portion of reuses have distances between d_i and d_{i+1}. A randomly selected reuse distance has probability p_i to be between d_i and d_{i+1}. From this we can compute estimates such as the average reuse distance or the most probable reuse distance.

We represent a distribution with a w-wide histogram. It uses logarithmic ranges for its bins, where each consecutive power-of-two range is divided into w bins of equal size. For example, if M is the number of data blocks used by an execution, the locality signature has a total of $w \log M$ entries. Hence the histogram is logarithmic in size yet the precision can be tuned by controlling w, which is 8 in our implementation. More discussion about the w-wide histogram can be found in the Appendix.

The reuse time is represented by a *distribution map*, which maps from an input distribution to an output distribution. The reuse-time map T_i converts from reuse distance to the length of its time window. It is a matrix whose first dimension (rows) consists of bins of reuse distances and second dimension (columns) bins of time-window sizes. Each cell p_{ij} is the probability that a reuse distance in the ith distance bin has a time window of size in the jth window-size bin. The matrix can be read by rows, which shows the distribution of time window sizes for a reuse distance, or by columns (with additional bookkeeping), which shows the distribution of reuse distances for a time-window size.

The footprint is also represented by a distribution map that maps from a time window size to a footprint. It is a matrix with bins of time-window sizes in the first dimension and bins of footprints in the second dimension. Each row shows the distribution of footprints for a time-window size, and each column shows the distribution of time-window sizes for a footprint.

A reader may detect certain redundancy in the two maps. Since both reuse distance and footprint measure the number of distinct data accessed in a time window, why isn't the footprint map simply the inverse of the reuse-time map? This is due to the difference between a reuse window and a time window. A reuse window must access the same data at both ends, but a time window can access different data. For a trace of length n, numerically there are $O(n)$ reuse windows but $O(n^2)$ time windows. If one randomly selects a time window in an execution, it is most likely not a reuse window.

Therefore we cannot use the inverse of the reuse time map as the footprint map.

Throughout the paper we assume an application model where a concurrent execution of multiple threads is recorded in an interleaved trace. The private reuse distance and the size of its time window are measured by extracting from the interleaved trace just the accesses by one thread. The private footprint is measured by random sampling of time windows and recording the relation between the window size and the volume of data accessed in the window.

4.3 General Thread Interleaving

The executions of different threads may not interleave uniformly. This is often the case for client applications, when threads perform different tasks. For server workloads, the relative rate of execution of parallel threads may change from one phase to another. In both cases, it is possible for a thread to execute only periodically. To be general, the interleaving model needs to answer two questions: when are threads are executed together, and what are their relative rates of execution?

The interleaving between from thread i to j is a distribution map from the size of reuse time window in i to the size of reuse time window in j. The key observation is that we care about the interleaving in all reuse windows rather than all windows. We can use the same record keeping as in reuse-distance measurement does. During the analysis, as we traverse the interleaved execution trace, we maintain a record of the private execution counts, that is, the number of executed instructions by one thread, at the points of the last access of each datum. At each data access, we compute the instruction count since the last access of the same datum. Since we store the interleaving relation between each pair of threads, the number of interleaving maps is quadratic to the number of threads.

Let B be the number of bins in a time histogram. For each thread t, the interleaving with other $k-1$ threads is represented by two $B \times (k-1)$ matrices: the *probability matrix* and the *ratio matrix*. In both matrices, each row is a reuse distance bin. The element (b_i, t_j) is the probability of a reuse distance of length b_i being concurrent with the execution of Thread t_j. When they are concurrent, the element (b_i, t_j) in the ratio matrix gives the rate of execution of Thread t_j relative to t in that window. We denote the two matrices as $Interleave_t^{prob}$ and $Interleave_t^{rate}$.

With general thread interleaving, the extended model is shown in Figure 4. For each bin of private reuse distance, the algorithm computes all non-empty companion subsets. For example, if Thread t is run with u, v, this step separates the times when t runs with just u, with just v, and with u, v. For each companion set, it invokes the data model (described in the next section) as a subroutine to compute the new reuse distance. It then computes the probability *prob* of the companion subset and use it to weigh the new reuse distance. The subroutine $ComputeConcurrencyProb$ uses the standard inclusion/exclusion method and is omitted. The subset enumeration includes the empty set, in which case Thread t runs by itself.

Uniform interleaving is a special case of general interleaving. In this case, there is one companion subset including all other threads. Every number in the probability matrix is 1, and every column of the ratio matrix has the same value. Other types of matrices represent the general case. The model accounts for the full range of possible interleaving.

One may question whether the two numbers, interleaving probability and ratio, can be combined so the relation is represented by one matrix rather than two. For example for a reuse window size, if Thread j executes with i 50% of the time and has an interleaving ratio of one-to-one, then Thread j would execute the same number of instructions in these windows if j executes with i 100% of the time at a ratio of half-to-one. However, the effect on reuse distance is actually very different. 50% of i's reuse distances are unchanged in the first case but all reuse distances are changed in the second case. Indeed, we have observed poor prediction results when we tested a version of the model using this idea.

The space cost of the interleaving model is quadratic to the number of threads. We have considered two alternative solutions. The first avoids the quadratic space cost by using a global virtual time and computing the relative rate of execution in each virtual time window. However, it needs an additional map from the reference count of each thread to its virtual time range. In addition, it needs to measure all windows rather than all reuse windows. The second method measures the interleaving probability for each thread subset at a cost exponential to the number of threads. In comparison, our interleaving model has the lowest analysis cost, and the space cost in absolute terms is not large.

When threads do not share data, the concurrent cache use effectively reduces the cache size available to each thread. The model in this section computes this negative interference. The model shows that a reuse distance is never shortened, which means that shared locality is never better than private locality. When threads share data, however, they may improve the cache performance of one another. We consider the shared-data case next.

5. THREADS WITH SHARED DATA

In this section, we consider the case where multiple threads access shared data in a shared cache. First, we describe a model that estimates the amount of data sharing between threads. Next, we show how data shar-

Inputs: (Thread t running with $t_1, t_2, ..., t_{k-1}$)
the same first four inputs as in data sharing model (Figure 6)
$Interleave_t^{prob}$ and $Interleave_t^{rate}$: the interleaving probability and rate
Output
the shared locality signature of thread t, $ReuseSig_{t'}$
Algorithm InterleaveShare ($ReuseSig_t, ReuseTimeMap_t, FootprintMap_{t,t_1,...,t_{k-1}}, U,$
$Interleave_t^{prob}, Interleave_t^{rate}$)
foreach reuse distance bin $r_i = [d_1, d_2]$ that has p_i portion total references
foreach subset S of $(t_1, ..., t_{k-1})$
dist = GetSharedReuseDis($S \cup \{t\}, r_i, ReuseTimeMap_t, FootprintMap, U, Interleave_t^{ratio}$) /* Section 5.2 */
prob = GetConcurrencyProb($S, Interleave_t^{prob}$)
add ($dist' p_i prob$) to $ReuseSig_{t'}$
end foreach subset
end foreach distance bin
End algorithm

Figure 4: The locality model of threads with general interleaving

ing affects the miss rate in a shared cache (of any size). Finally we briefly discuss how to model coherence misses to extend this model to partitioned caches.

5.1 Composable Data Sharing Model

We divide the data accessed by a thread into three components or types. The first is *always shared data*, which is accessed by all threads. We denote its size by x. This represents shared data that is always accessed in the given scenario. Examples of this type are usually global constants in an application such as the root node of a B-tree or the front node of a shared list.

The second type is *potentially shared data*, which is a pool of data that any threads can access. The size of the shared pool is the same for all threads, which we denote w. The probability of thread i accessing a datum in w is m_i ($0 \leq m_i \leq 1$). This category represents shared data that is not necessarily accessed by all threads, such as the leaf nodes of shared tree structures. The last type is the *private data* accessed by each thread. We denote its size as p_i for Thread i.

The complete data accessed by Thread i is u_i, which is the sum of the three components, $u_i = x + m_i w + p_i$. We call our model the *WXP* model for its three components. The combined sets of always shared, potentially shared, and private data are called three universes. As a basic property in our model, different data universes have no overlap. Figure 5 illustrates the *WXP* model with a two thread example. Note that while the example shows *WXP* as sets for illustrative purposes, our model only computes the size of these sets not their contents.

This model is a deliberate approximation since for efficiency it is not scalable to record the sharing in all thread subsets. For a system of k threads, the model approximates these $O(2^k)$ relations with $2k+2$ quantities: two numbers per thread, p_i and m_i, and two numbers for the system, x and w. In addition, it makes it possible for one to investigate the impact of additional threads by modeling their data sharing with these parameters, p_i and m_i.

When analyzing multi-threaded applications, we discovered that the data footprint and data sharing differed greatly between different threads. The *WXP* model is designed to approximate asymmetrical threads and their data sharing. The model does not constrain the size of private data in a thread. Each can have any size p_i. If a group of threads share more data than another group, the model will assign higher m_is to the first group.

Data sharing in an application may involve many data structures accessed in different ways, and the sharing may change at different time scales. *WXP* attempts to capture the first order effects of variable data size and data sharing by taking some average statistics over the entire trace. There is naturally a limit to how much an automatic technique can analyze a program. Our strategy is to develop a minimal and intuitive model and then test its fit with a range of applications.

The complication, even in this simple model, is that *none of the* $2p+2$ *numbers can be directly measured.* For example, if a datum is accessed by all threads, it could be part of x or always shared, but it could also be part of w or potentially shared when it happens to be accessed by all threads. Similarly, if a datum is accessed by only one thread, it may be either w potentially shared or p_i private.

Instead of direct measurement, we derive the model parameters through one-pass profiling of the interleaved execution. We need enough observations to constrain the calculation. For p threads, there are $2^p - (p+1)$ sharing relations (subsets with at least two members). Minimally we need $p = 4$ so the number of observations, $2^p - (p+1) = 9$, is greater than the number of parameters $2p + 2 = 8$. We next describe the derivation for $p \geq 4$.

First, we record the set of data blocks accessed by each thread and compute the amount of data sharing between all subsets of threads. Let n be the total data size and p the number of threads, the computation cost

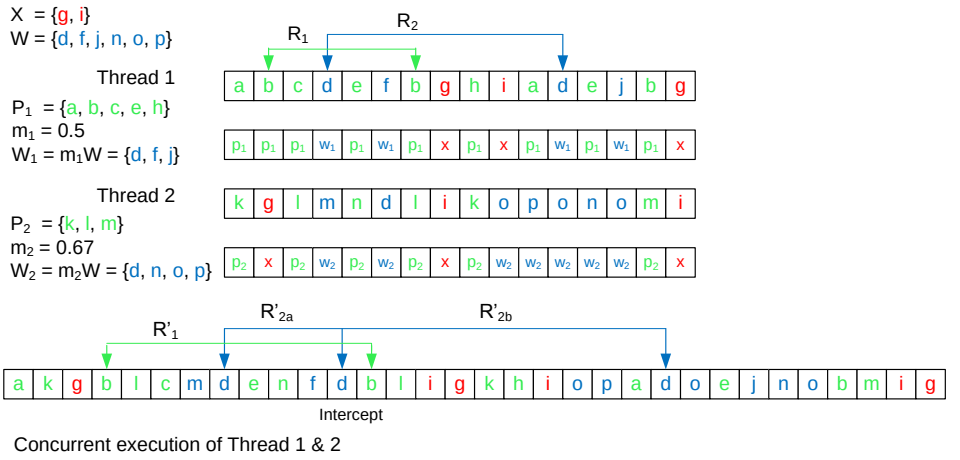


Figure 5: Illustration of the data sharing model and the effect of intercepts

is $O(n2^p)$ and the space cost is $O(n)$ for this step.

Assuming $p = 4$, we next compute the average size of shared universe between 2, 3, and 4 threads. Let the three numbers be $\bar{u^2}, \bar{u^3}$, and $\bar{u^4}$. We then approximate the average $\bar{m} = mean(m_i)$. Observe that from

$$\bar{u^2} \approx x + \bar{m}^2 w$$
$$\bar{u^3} \approx x + \bar{m}^3 w \quad \text{we compute} \quad \bar{m} = \frac{\bar{u^3} - \bar{u^4}}{\bar{u^2} - \bar{u^3}}$$
$$\bar{u^4} \approx x + \bar{m}^4 w$$

With $\bar{m}$ we solve for x and w using any of the above two *now linear* equations for $\bar{u^2}, \bar{u^3}$, and $\bar{u^4}$. The choice does not matter since a solution to any two is the solution to all three. A higher p can be considered at this step by taking the best approximation when the number of equations is greater than the number of unknowns.

The last step is to compute the approximation of m_i and p_i. Let $u_{i,j}$ denote the size of shared universe by threads i and j. Then we have $\frac{m_i}{m_j} = \frac{u_{i,q}}{u_{j,q}}$ for $q \neq i, j$. To improve precision, we can approximate the ratio by the average from all q. Once we have an estimate for $\frac{m_j}{m_1}$ for all $j \neq 1$, we compute $m_1 = \frac{\bar{m}p}{1+\sum_j \frac{m_j}{m_1}}$ and then $m_j = \frac{m_j}{m_1} m_1$. Finally, we have $p_i = u_i - m_i w - x$, where u_i is the size of the data accessed by thread i.

The *WXP* model requires four or more threads in an interleaved trace. With enough observations, it solves a non-linear equation to derive the parameters that give the best approximation to the observation. The model can be applied to the whole trace or partial pieces. The *WXP* is composable. Each thread is represented by a set of numbers, and the behavior of a thread subset is composed from per-thread numbers, without re-analyzing the trace. One benefit of a composable model is that we can "clone" a thread by giving a new thread the same WXP parameters. In this way we can study the behavior of hypothetical thread groups however large they are.

5.2 Computing Shared Cache Miss Rate

We now describe how to consider data sharing when predicting the miss rate of shared cache of all sizes. Data sharing has two effects on a reuse distance. First, the size of data accessed by multiple threads is not simply additive. Second, there is the chance of an *intercept*, when a reuse distance in one thread is split into two segments by an intervening access from another thread. The two-thread example in Figure 5 shows both effects. Datum i is accessed by both threads during the course of reuse distance R_2 and should not be double counted in R'_2. Thread 2 accesses d in the middle of the time window of R_2, forming an intercept and dividing the reuse distance into R'_{2a} and R'_{2b}.

The algorithm for the k-thread case is given in Figure 6, where Thread t is running with Threads $t_1, ..., t_{k-1}$. The inputs to the model include the reuse-distance bin r, the reuse-time map of Thread t, the footprint map, the data sharing model, i.e. the *WXP* parameters, the data size for all threads, and the interleaving ratio (Section 4.3) between other threads and Thread t. Recall that the thread interleaving probability is used in the parent subroutine `InterleaveShare` shown in Figure 4.

The algorithm can be thought of having three stages. The first stage computes the footprint for the $(k-1)$ peer threads. It has three steps: finding the length of the time window of r in Thread 1, finding the coinciding time windows in peer threads, and finding the footprint in these windows. Stage 2 computes the reuse distance assuming no intercept. It has four statements: decomposing r based on the *WXP* model, decomposing footprints f_{t_i} based on the *WXP* model, computing the size of shared data $data_{shared}$, and adding it to the size of private data to obtain the result, $dis_{no\ intercept}$.

The key step in Stage 2 is estimating the overlap in shared-data access. From the *WXP* model, we have separated the shared-data components in the reuse dis-

Inputs: (Thread t running with $t_1, t_2, ..., t_{k-1}$)
r: a private reuse distance in Thread t
$ReuseTimeMap_t$: the reuse time map of Thread t
$FootprintMap_{t_1,...,t_{k-1}}$: the footprint map of Threads $t_1, t_2, ..., t_{k-1}$
$U_{t,t_1,...,t_{k-1}}$: the WXP sharing model
$M_{t,t_1,...,t_{k-1}}$: total data size of Threads $t, t_1, t_2, ..., t_{k-1}$
$Interleave_t^{rate}$: the interleaving ratio of Threads t with $t_1, t_2, ..., t_{k-1}$
Output
r': the modified reuse distance considering parallel execution and data sharing
Algorithm GetShareReuseDistance ($r, ReuseTimeMap_t, FootprintMap_{t_1,...,t_{k-1}}, U_{t,t_1,...,t_{k-1}}, M_{t,t_1,...,t_{k-1}}, Interleave_t^{rate}$)
the time window of r in Thread t is $time_t = ReuseTimeMap_t(r)$
the coinciding time window in Thread t_i is $time_{t_i} = time_t Interleave_{t \to t_i}^{rate}$, $i = 1, ..., (k-1)$
the footprint of t_i is $f_{t_i} = FootprintMap_{t_i}(time_{t_i})$, $i = 1, ..., (k-1)$

decompose r into $< r^x, r^w, r^p >$ such that $r^x + r^w + r^p = r$ and $r^x : r^w : r^p = U^x : U_t^w : U_t^p$
decompose each f_{t_i} into $< f_{t_i}^x, f_{t_i}^w, f_{t_i}^p >$ similarly based on $U^x, U_{t_i}^w$, and $U_{t_i}^p$
$data_{shared}$ = GetUnionSize($r^x, f_{t_1}^x, ..., f_{t_k}^x, r^w, f_{t_1}^w, ..., f_{t_k}^w$)
$dis_{no\ intercept} = r^p + \sum f_{t_i}^p + data_{shared}$

$prob_{intercept} = \frac{r^x}{M_t} + \frac{r^w}{M_t}$
$dis_{intercept} = \frac{dis_{no\ intercept}}{k}$
return $dis_{no\ intercept}(1 - prob_{intercept}) + dis_{intercept} prob_{intercept}$
End algorithm

Figure 6: Computing the effect of data and cache sharing on private reuse distance r in Thread t, when t is run in parallel with $k-1$ other threads.

tance r and each footprint f_{t_i}. Consider the X components: r^x from the reuse distance and $f_{t_i}^x$ from each footprint. We need to estimate the size of the union of these k data sets. Let the union size be s_{union}. We have

$$max(r^x, f_{t_1}^x, ..., f_{t_{k-1}}^x) \le s_{union} \le sum(r^x, f_{t_1}^x, ..., f_{t_{k-1}}^x)$$

The inequality shows that s_{union} is bounded below by the size of the largest component and bounded above the total size of all components. We call the lower bound s_{max} and the upper bound s_{sum}.

We estimate s_{union} by taking the joint probability. We take $\frac{r^x}{U^x}$ and $\frac{f_{t_i}}{U^x}$ as the probabilities. We compute s_{union} as the union of their probabilities using the inclusion-exclusion principle. Let $p_1, p_2, ..., p_k$ be the probabilities. The union p_{union} is

$$p_{union} = \sum_{i=1}^{n} (-1)^{i-1} \sum_{all\ subset\ S\ of\ size\ i} \prod_{i \in S} p_i$$

Then we have $s_{union} = p_{union} U^x$, where U^x is the size of the X data in the WXP model.

As mentioned before U^x is cumulative sharing and not necessarily the active sharing, especially in small execution windows. We have experimented with different methods of estimating the working set size. Since different threads may execute a very different number of instructions during a time window, it is unclear what a common working set means. We therefore use the sum of all components, $s_{sum} = sum(r^x, f_{t_1}^x, ..., f_{t_{k-1}}^x)$, for smaller ranges of r. Instead of setting an arbitrary threshold, we let the model dynamically pick from s_{sum} and s_{union} the one that is closest to $mean(s_{max}, s_{sum})$. The shared portion of W data is computed in the same way. The total shared data, $data_{shared}$, is the sum of the two. The reuse distance without an intercept, $dis_{no\ intercept}$, is the sum of private data accessed by all threads plus $data_{shared}$, as computed in the last statement in Stage 2.

When predicting for a large number of threads, we approximate the exponential-cost inclusion/exclusion computation by first calculating the average probability and then computing as if all threads have this probability. The approximation may overestimate the amount of data sharing (which can be shown using Jensen's inequality), but the time cost is linear to the number of threads rather than exponential.

Stage 3 of the algorithm computes the effect of intercepts. The probability and the frequency of intercepts depend on the overlap among shared-data components. By experimenting with the inclusion-exclusion principle, we found that two primitive heuristics perform well: the average probability of the intercept is the relative size of shared data in Thread t, and the number of the intercepts is the number of thread peers. The algorithm uses the denominator k when computing $dis_{intercept}$ because $k-1$ threads mean $k-1$ intercepts, which divide a time window into k sections.

5.3 Extension to Partitioned Caches

We describe a simple extension for modeling coherence misses that allows us to extend this model to com-

pre-processing	time	model size
locality signature (§ 4.2)	$O(t \log\log n)$	$k \log n$
reuse time (§ 4.2)	$O(t \log\log n)$	$k \log t \log n$
footprint (§ 4.2)	$O(t \log\log n)$	$k \log t \log n$
data sharing (§ 5.1)	$(n2^k + t)$	$2k + 2$
interleaving (§ 4.3)	$O(t)$	$k^2 \log n$

prediction	time	space
no shared data (§ 4.3)	$O(p^2 \log n)$	$O(k \log t \log n)$
with shared data (§ 5.2)	$O(p2^{2p} \log n)$	$O(k \log t (k + \log n))$
simulation	$O(t \log n)$	$O(t)$

Table 1: The time complexity in data collection and locality prediction and the size of model data. The model stores statistics for $\log t$ window sizes, $\log n$ cache sizes, and 2^p thread combinations.

pute locality for partitioned caches. If we assume that each thread accesses a separate partitioned cache then the only way for another thread to impact its locality is by writing to a shared data item that results in a cache invalidation (we ignore the impact of write to read downgrades). This situation only impacts the cache miss rate computation when the second thread *intercepts* the reuse distance of the first thread (see Figure 5), the intercept is a write access, and the reuse distance was less than the cache size (otherwise the access would have anyway counted as a capacity miss). The number of coherence misses can be modeled by multiplying the intercept probability with the probability that the corresponding access was a write. This write probability is estimated from the interleaved trace. When multiple threads share a partitioned cache, we first use the shared cache, shared data model to estimate the combined reuse distances for each partitioned cache and then model coherence misses across these caches.

5.4 Time and Space Complexity

Table 1 summarizes the asymptotic cost of collecting and storing the five model components and of using them in miss-rate prediction. The symbols are t, the length of the interleaved trace; n, the size of accessed data; k, the number of threads in the training run; p, and the number of threads for prediction. The table shows the profiling cost, prediction cost, and for comparison purpose the cost of exhaustive simulation. The cost of the reuse distance and footprint measurement is based on a relative-precision algorithm [13].

6. EVALUATION

We first measure the accuracy of our model statistics in reproducing the cache effect of a multi-threaded execution and then use it to compose the cache effect for different thread mixes without re-executing a program.

6.1 Methodology

For a trace of t data accesses to n data blocks in p threads, our goal is to predict the miss rate for $O(\log n)$ cache sizes and 2^p thread combinations. Our model consists of a number of components interacting with each other, which makes it difficult to attribute and explain individual effects and therefore raises the question that a result may due more to chance or minor details than to the reasoning we have used in constructing the model.

Our design strategy is to build the model entirely from observed data in each trace with *no thresholds or constants specific for an application, a cache size, or a thread combination*. Each component of the model exerts a systematic effect on the prediction for all programs, cache sizes, and thread combinations. The model cannot be tuned for any single prediction scenario. To evaluate, our strategy is to test the accuracy of as many different predictions as possible. We include results for many cache sizes not because these are practical in cache design but because they provide additional tests of the model. Accurate prediction in all scenarios would provide assurance that the statistics in the model captures key locality characteristics of multi-threaded applications, at least the ones in our test set.

6.2 Benchmarks

We collected traces from four multi-threaded commercial applications using an instruction-level tracing tool [6]. The first trace is from a server application running on a quad-processor system. The other three are a productivity, a multimedia, and a game application, run on a dual-processor system. We call these traces *Server*, *Productivity*, *Multimedia*, and *Game*. The difference in the machine platform will become significant later when we see that the interleaving model is critical for dual-processor runs. The characteristics of the traces are given in Table 2. Most traces have 50 million memory accesses except for the server trace, which is 30 times larger and has 1.7 billion memory accesses. The miss rate results that follow are reuse miss rates in that the cold-start or compulsory misses, which is equal to the number of blocks used by a trace, are not counted. A simple calculation from the data in the table shows that the cold-start miss rates are below 0.2% for all except *Game*, which is 0.4%.

The four traces have 4 to 22 threads. The last two columns show the size of the largest single thread and of the largest four threads. *Productivity* and *Multimedia* have little parallelism—a single thread accounts for over 90% and 97% of the actions. The extreme asymmetry provides a good test for the interleaving model. *Server* has the

traces	number of			size of	
	accesses	blocks	threads	top 1	top 4
Server	1.70B	989K	22	29.7%	99.5%
Productivity	40.0M	36.6K	4	90.2%	100%
Multimedia	50.0M	106K	7	97.0%	99.8%
Game	50.0M	190K	6	71.4%	99.8%

Table 2: Characteristics of the four multi-threaded execution traces

most balanced parallelism—the largest of the four accounts for less than 30% of the total. We model the top four threads, which account for at least 99.3% of a trace. The unit of data is 64-byte blocks. We show results for data locality, which has the highest miss rate. The same model can predict for instruction locality and mixed instruction and data locality.

6.3 Model Validation

Figure 7 compares the measured and predicted locality of our test applications running four threads on 8 cache sizes from 32KB to 4MB. In each of the four graphs, the curve "sim share" shows the miss rates measured through simulation of the interleaved trace. The curve "sharing+interleaving models" shows the miss rates predicted by our technique. The curve "no sharing, uniform interleaving" shows the prediction assuming no data sharing and uniform interleaving. Finally as a reference, the curve "avg serial" shows the (weighted) average sequential locality of the four threads.

The first two applications, *Server* and *Game*, displayed in the upper two graphs in Figure 7, show significant effect of data sharing: concurrent execution ("sim share") increases the miss rate of individual threads ("avg serial"), and the increase would be far more if the threads do not share data and run independently ("no sharing, uniform interleaving"). We have tested the effect of data sharing and thread interleaving separately. It turns out that the interleaving is mostly uniform, and the prediction is based mostly on the effect of data sharing. The fact that the prediction closely matches the measurement in all cache sizes indicates the accuracy of our data sharing model, including the derivation of the *WXP* parameters and the consideration for intercepts.

The next two applications, *Productivity* and *Multimedia*, displayed in the lower two graphs in Figure 7, have asymmetrical interleaving. The largest thread accounts for 90% and 97% of all data accesses. The interleaving model shows that the three smaller threads only run occasionally rather than at all times, so the locality of the concurrent execution is close to the average serial locality. The assumption of uniform interleaving, however, leads to a prediction of a much higher miss rate, as shown by the curve "no sharing, uniform interleaving." We have also observed similar mis-prediction if we do not separately consider interleaving probability and interleaving ratio (discussed in Section 4.3). This effect of the interleaving model diminishes over large execution windows. All methods predict the miss rate correctly for cache sizes 128KB and up in *Multimedia* and 512KB and up in *Productivity*. In other tests, we found that data sharing has little visible effect on the miss rate, and the prediction is based entirely on the effect of interleaving.

The gap in *Multimedia* shows a property of distance-based locality prediction. The model predicts for each reuse-distance bin, while the miss rate is cumulated from multiple bins. An error happened for reuse-distance bin at 1MB, and the error is carried left through the smaller cache sizes.

In the four test applications, *Server* has the longest length and the largest amount of data and data sharing. Figure 8 shows the same type of plots as in Figure 7 but for two-thread and three-thread executions. There are a total of 11 different concurrent combinations, 6 2-thread sets, 4 3-thread sets, and 1 4-thread set. Since it is impractical to compare 22 curves in a single figure, we show a 2-thread set and a 3-thread set by taking the two and the three threads that have the lowest thread identifiers.

By processing the 1.7 billion memory access trace once, the model is able to predict the miss rate of 11 different executions for 8 different cache sizes, a total of 88 different miss rates with models constructed over a single pass of the access trace. In the 3-thread execution shown in Figure 8, the miss rate drops from 2.1% in the 32KB cache to 0.016% in the 4MB cache. The absolute error in prediction drops from 0.077% at 32KB and 0.098% at 64KB to 0.0063% at 2MB. The relative error increases from 3.4% at 32KB and 7.3% at 64KB to 13% at 2MB. We observe similar results for other concurrent executions of the *Server* threads.

The *WXP* model. The amount of data sharing is estimated by the *WXP* model. Table 3 shows the size of always shared, potentially shared, and private data for the *Server* trace, which has the highest degree of sharing among the four benchmarks. Over 9 thousand blocks are estimated to be always shared, and each thread has between 16% to 18% probability accessing a pool of 177 thousand blocks. The private data range between

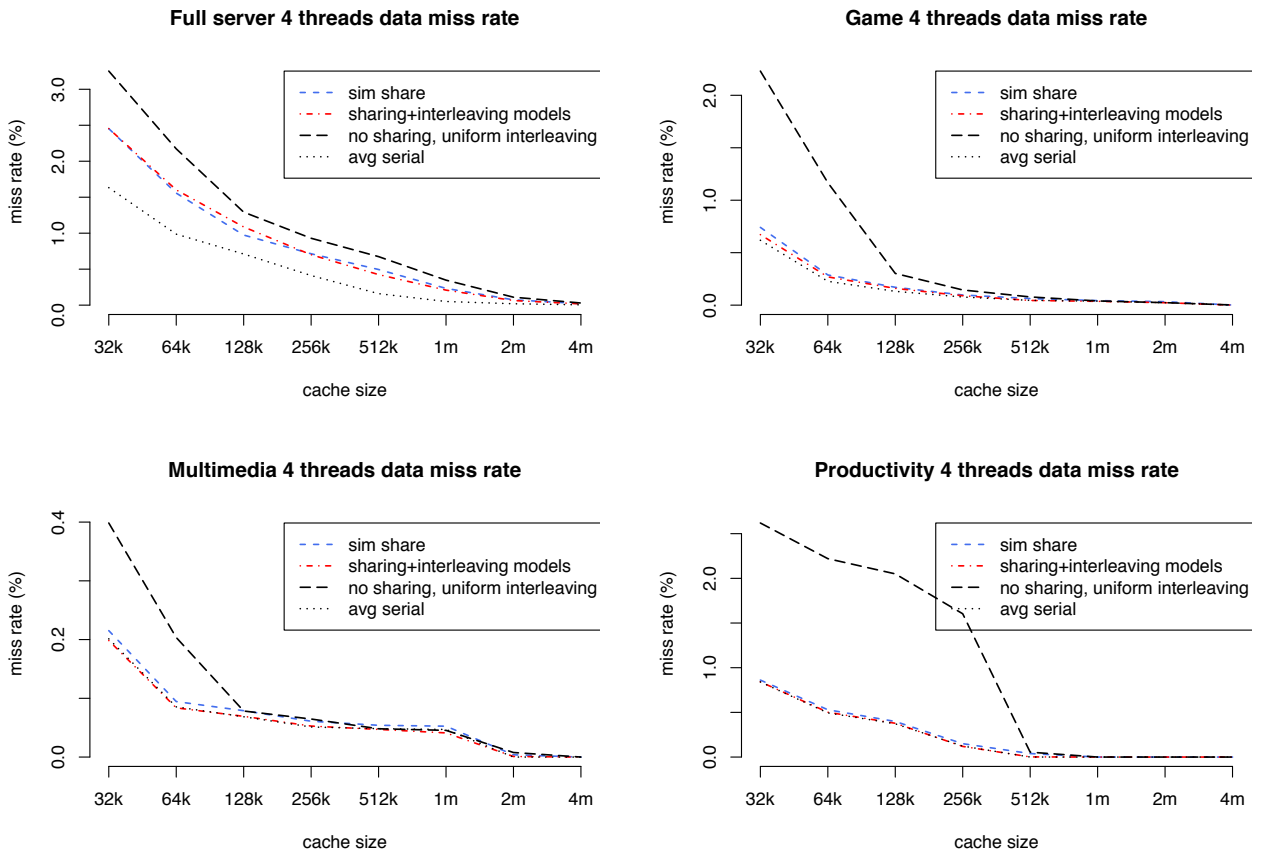


Figure 7: Measured and predicted locality for the largest four threads for a mix of server, desktop, and game applications. The prediction shows the effect of data sharing in *Server* and *Game* in the top two graphs and thread interleaving in *Multimedia* and *Productivity* in the bottom two graphs.

X	W	thread id	prob. acc. m_i	private p_i	tot. data	tot. acc.
9K	177K	8	18.7%	56K	99K	494M
		40	18.6%	57K	99K	486M
		88	17.3%	67K	107K	207M
		72	16.3%	62K	100K	505M

Table 3: The *WXP* model for *Server* threads

98 thousand to 106 thousand blocks. The effect of data sharing is greater at larger thread counts, as shown in Figure 8 by the growing gap between the simulated miss rate and the prediction assuming no data sharing.

The cost of modeling. For the largest trace we analyzed, a histogram contains less than 300 numbers and a distribution map contains less than 80000 numbers. The pre-processing takes less than six hours for the longest trace and less than an hour for smaller traces and only needs to be done once per trace. The prediction time using a 2.2GHz Intel Core 2 Duo takes less than 20 seconds for each thread combination up to four threads and less than 10 minutes for 32 threads.

6.4 Predicting Scalability

With the composable model we can examine the scalability of a concurrent program by adding threads with similar characteristics to those we have already modeled. In this section, we evaluate the concurrent execution of different thread counts from 8 to 32 by replicating the set of *Server* threads by factors from 2 to 8. The predicted behavior of large concurrent runs can reveal or verify trends in the effect of data sharing that are not as visible in small scale executions. It can also predict the resource demands for cache and memory bandwidth for future systems.

We show two predictions: the positive effect of data sharing in a shared cache and the negative effect of data sharing in a partitioned cache.

Figure 9 shows the increase of the miss rate with the number of threads for caches of size 512KB to 64MB. The two graphs are miss rates with data sharing (on

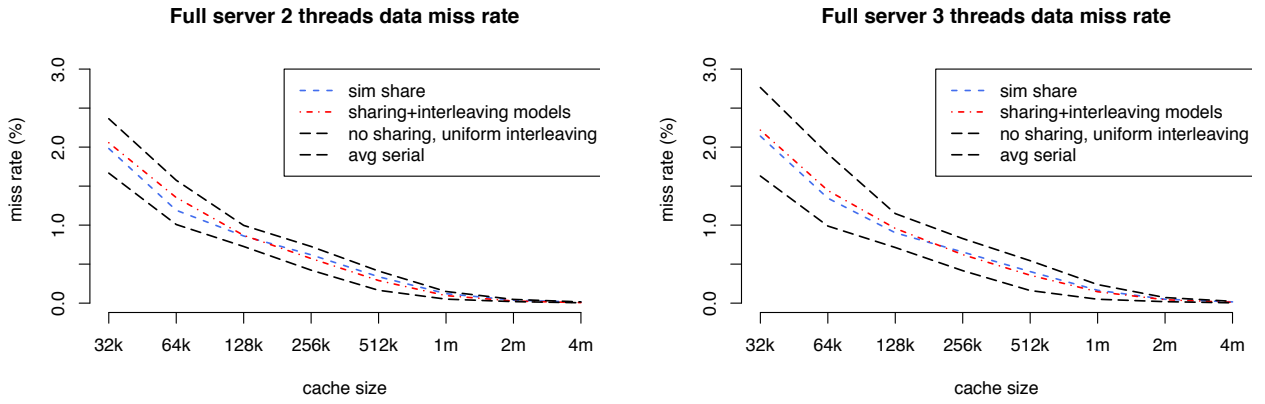


Figure 8: Measured and predicted locality for 2 and 3 threads of the *Server* application. The effect of data sharing is greater in a larger number of threads. The 4-thread *Server* is shown in the upper-left graph in Figure 7.

the left) and without data sharing (on the right), and the benefit of data sharing is shown by the lower miss rates on the left. Note that we simulated no sharing by effectively tagging every access with a thread ID, so no two threads share data. The average reduction in the overall miss rate is 31%, 29%, 44%, 57%, and 50% for 2, 4, 8, 16, and 32 threads. The reduction is higher for more threads and also for larger caches. The average reduction is 28%, 32%, 40%, 49%, 70%, 74% and 82% for the seven cache sizes from 512KB to 32MB.

These estimates may be optimistic because we compute them using the average sharing rather than inclusion-exclusion formula, as discussed in Section 5.2. However, the error is small if the ratios are similar, which is the case for the *Server* threads. We plan to study more accurate approximation schemes but we believe that the general trends shown in these results are correct.

The partitioned cache interacts with other caches through coherence events. For a running thread, the most disruptive effect of data sharing are invalidations. We call the thread using the cache the *resident thread* and other threads *neighbor threads*. If a neighbor thread modifies a data block that is present in the resident's cache, the block will be removed and the next access by the resident thread will be a miss, regardless of the reuse distance.

In the next experiment we use the extension to the data sharing model (Section 6.3) to estimate the likelihood of data blocks that are modified by at least one neighbor thread. This model has not been carefully refined, and its prediction has not been exhaustively verified. Nevertheless, the preliminary results already demonstrate an interesting property of the coherence misses.

Figure 10 shows the overall miss rate for one of the

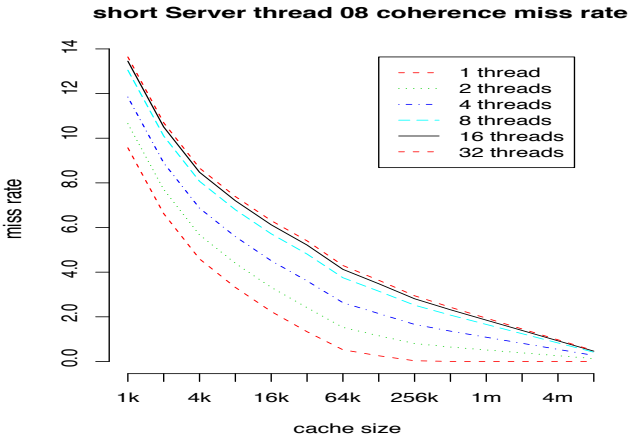


Figure 10: Projected miss rates, including the coherence misses, of a short *Server* thread in a partitioned cache

short *Server* threads, including the number of coherence misses. The cache size ranges from 1KB to 8MB. The curves are labeled by the total number of threads. The "one thread" case is when there is only the solitary resident. As we add 1, 3, and 7 neighbor threads, the miss rate increases significantly. After that point, however, the increase nearly stops. An additional 8 and 24 neighbor threads have little effect on the miss rate of the resident thread.

The cause can be explained in terms of data sharing. The neighbor threads can invalidate shared data in the resident's cache, and the more neighbors threads there are, the more likely a shared block is invalidated before it is reused. At some point, all shared accesses of the resident thread become cache misses because of heavy invalidation, and the miss rate no longer increases.

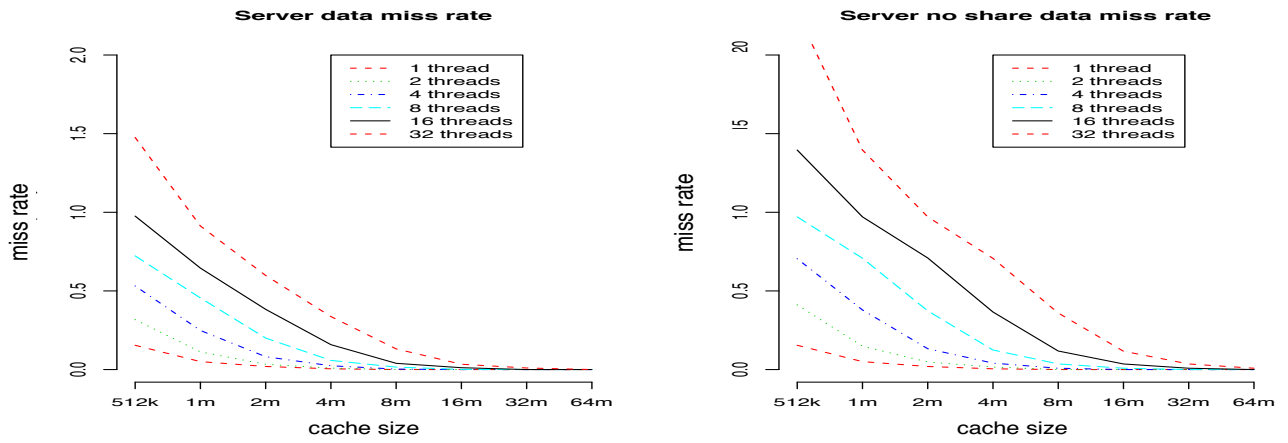


Figure 9: Projected locality of up to 32 *Server* threads

7. RELATED WORK

Distance-based models. For time-shared environments, Suh et al. showed how the reuse distance of one process is lengthened by the footprints of others [35]. Chandra et al. were the first to demonstrate this type of calculation for parallel processes running on multiprocessor systems [8], which is later extended to accurately model the effect of cache conflicts [10]. In this paper, we augment the models in [8, 35] (which is described in more detail in Section 4.1) to consider data sharing and thread interleaving, which are necessary to predict locality of multi-threaded applications.

Performance of shared cache can be measured directly either through hardware counter sampling [5] or by augmenting the notion of reuse distance [32]. The measurement includes the effect of data sharing, intercepts, and cache invalidation for a particular concurrent execution. Instead of direct measurement, our approach tries to model the effect from per-thread statistics. Because it is composable, our technique can be used to infer how data sharing changes in different thread combinations without re-analyzing different executions.

Static analysis. For scientific applications, regular array section analysis (by many researchers, see [2]) has been used to characterize shared data access to reduce false sharing on multiprocessors [23] and to improve data prefetching and message aggregation on DSM [28]. More general forms of dependence analysis are used by parallelizing compilers [7,20,22,26]. Static analysis does not directly estimate the locality of active data usage, which depends on factors outside the scope of data dependence, such as the degree of interleaving, the frequency of data access, and the effect of read-read sharing. In addition, traditional dependence checking cannot accurately model pointer-based programs such as the applications considered in this paper.

Window-based models. Footprint models have been developed for time-shared systems where processes are switched at regular intervals or execution windows. They are used to estimate the cache interference by the probability of one process evicting the cached data of another process [1, 17, 36]. In these studies, the statistics such as reuse time and footprint is for a single window size rather than all window sizes as in distance-based methods. A single window size is not always sufficient. For example, the footprint of a large window may not simply be the sum of the footprints of its sub-windows. Previous models do not consider data sharing except by Falsafi and Wood, who assumed that code was shared but data was not [17].

Analytical models. The independent reference model assumes a program with n pages, and each has an independent access probability p that adds to 1. King defined the model and showed that steady miss rate exists for fully associative caches managed by LFU, LRU, and FIFO replacement policies [25]. Later studies gave approximation methods for LRU and FIFO that are computationally tractable [11, 16]. These methods, especially [11], can be viewed as composable models because they show the combined miss rate from interleaved access streams to different data. Recent techniques analyze the effect of cache replacement policies by probabilistic prediction [19] and competitive analysis [31].

Task scheduling and cache partitioning. One way to manage shared cache is to partition it among concurrent applications for fairness or performance. Cache partitioning can be done by OS level support including task scheduling [18] and page coloring [27]. The problem of optimal co-scheduling can be approximated using machine learning techniques [24]. An earlier study addressed the resource partitioning problem at the main memory level [37]. These techniques use highly efficient

on-line locality and performance models to dynamically adjust the memory system when a program is executing. However, they target independent sequential applications rather than multi-threaded applications. As a result, they do not consider the effect of data sharing or irregular thread interleaving. A recent study focused on constructive caching in three kernel benchmarks and developed a profiling step that used a two-dimensional distance histogram to estimate the (sequential) miss rate of task sub-sequences to aid on-line task scheduling [9].

Other simulation-based analysis. In the context of DSM, Bennett et al. classified data sharing as write-once, write-many, producer-consumer, migratory, and general read-write [4]. For cache-coherent systems, Eggers and Katz defined *write run*, a sequence of write references to a shared data block by a single thread, uninterrupted by accesses from other threads [15]. These models focused on the amount and the type of sharing but not the timing of shared-data access and therefore cannot be used to evaluate cache capacity miss rate, which depends on data locality rather than on data access frequency or data size. Neither model considered cache size as a parameter. In comparison, we model the aggregate locality using the reuse distance and the probability of interleaving and intercepts.

In a system where shared data were explicitly declared, Darema et al. found that among other aspects, the portion of shared data was large, the fraction of accesses to shared data was small, and shared and private data benefited from caching by a similar fashion [12]. Different sharing granularity affects performance differently depending how and what data are shared. Based on this observation, Dubnicki and LeBlanc designed a dynamic cache block splitting and merging scheme [14] and Amza et al. designed dynamic page grouping scheme [3]. In these studies, the effect of data sharing was measured for a given thread combination rather than predicted for all thread combinations.

8. SUMMARY

In this paper we have shown that the cache behavior of a multi-threaded application can be modeled using a set of statistics collected over a one-pass analysis of the interleaved execution trace. The chief components include the interleaving probability, interleaving ratio, *WXP* data sharing, and the model for intercepts. We have tested the prediction for real multi-threaded applications and found that both data sharing and thread interleaving are necessary in capturing the memory behavior of these applications.

Acknowledgment

The authors wish to thank Ashit Gosalia, Sadashivan Krishnamurthy, Jim Larus, Ben Livshits, Madanlal Musuvathi, Rich West, Ben Zorn, and the rest of the RAD group for the discussions of ideas of the paper and its presentation.

9. REFERENCES

[1] A. Agarwal, M. Horowitz, and J. L. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.

[3] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Trade-offs between false sharing and aggregation in software distributed shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 90–99, 1997.

[4] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 125–134, 1990.

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

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the International Conference on Virtual Execution Environments*, 2006.

[7] W. Blume et al. Parallel programming with polaris. *IEEE Computer*, 29(12):77–81, 1996.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[9] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, 2007.

[10] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 329–340, 2009.

[11] A. Dan and D. F. Towsley. An approximate

analysis of the lru and fifo buffer replacement schemes. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 143–152, 1990.

[12] F. Darema, G. F. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1987.

[13] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[14] C. Dubnicki and T. J. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the International Symposium on Computer Architecture*, pages 170–180, 1992.

[15] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the International Symposium on Computer Architecture*, pages 373–382, 1988.

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

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

[18] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, September 2007.

[19] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 228–239, 2006.

[20] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in SUIF. *ACM Transactions on Programming Languages and Systems*, 27(4):662–731, 2005.

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

[22] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[23] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.

[24] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 220–229, 2008.

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

[26] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, Jan. 1990.

[27] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 367–378, 2008.

[28] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[29] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.

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

[31] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *Proceedings of the ACM SIGPLAN-SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, pages 51–60, 2008.

[32] D. L. Schuff, B. S. Parsons, and V. S. Pai. Multicore-aware reuse distance analysis. Technical Report TR-09-07, Purdue University School of Electrical and Computer Engineering, August 2009.

[33] Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), 2008.

[34] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main

memory management. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[35] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *International Conference on Supercomputing*, pages 1–12, 2001.

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

[37] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 177–188, 2004.

Appendix

A. RANGE-BASED CONVOLUTION

Given two threads and a time window of their execution, the combined footprint is a joint distribution from the two distributions of individual footprints. The computation is known as convolution in the discrete form. Convolution has many uses in statistics and digital processing. Common examples include computing the product of two polynomials and its special case the product of two large numbers.

Since we represent not the complete histogram but the average over some intervals, the footprint is a piecewise, continuous linear function. We use the integral form of the convolution. Let X and Y be two footprints and $p_X(n)$ and $p_Y(n)$, $(n > 0)$, be the probability of X and Y have a footprint of size n. The combined footprint $X + Y$ is defined by

$$d_{X+Y}(n) = \int_0^n d_X(x) d_Y(n-x) dx$$

The ranges, which are the bases, are in a logarithmic scale, like scalar multiplication and unlike function convolution (which is in linear scale). The bases are added, like function convolution and unlike scalar multiplication (where the bases are multiplied). In fact in the exponent domain, scalar multiplication is equivalent to function convolution, where logarithmic ranges become linear logarithm ranges and multiplication becomes addition. Previous work we are aware of does not address the problem of adding logarithmic ranges.

The essence of the problem lies in the combination of two ranges. In particular, the footprint X has p_X probability in range R_X, and the footprint Y has p_Y probability in range R_Y. The combined footprint can be represented as a third random variable Z, where Z has at least the probability $p_X p_Y$ in $R_X + R_Y$.

A.1 Combining Two Footprint Ranges

The sum of two footprints: the first has a probability of p within the range $[f_1, f_2]$, and the second has a probability of q within the range $[g_1, g_2]$. Using elementary statistics, we define two random variables, X and Y, to represent the two footprints. Assuming they are uniformly distributed, then the density function of X is

$$d_X(x) = \begin{cases} \frac{p}{f_2 - f_1} & x \in [f_1, f_2] \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The desity function for Y is similar. The joint distribution is $Z = X+Y$. It is easier to consider ranges from 0 and then add the initial offsets back. Let $\Delta f = f_2 - f_1$ and $\Delta g = g_2 - g_1$. We compute the joint distribution of p in $[0, \Delta f]$ and q in $[0, \Delta g]$ and re-state the results in terms of the original ranges. Without loss of generality we assume $\Delta g \geq \Delta f$.

$$d_{X+Y}(s) = \int_{-\infty}^{\infty} d_X(x) d_Y(s-x) dx \quad (2)$$

which is

$$\begin{cases} \int_0^s d_X(x) d_Y(s-x) dx = \frac{pqs}{\Delta f \Delta g} & 0 \leq s \leq \Delta f \\ \int_0^{\Delta f} d_X(x) d_Y(s-x) dx = \frac{pq}{\Delta g} & \Delta f \leq s \leq \Delta g \\ \int_s^{\Delta f + \Delta g} d_X(x - \Delta g) d_Y(s - x + \Delta g) & \\ = \frac{pq(\Delta g + \Delta f - s)}{\Delta f \Delta g} & \Delta g \leq s \leq \Delta f + \Delta g \end{cases} \quad (3)$$

The joint distribution for the original ranges is

$$d_{X+Y}(s) = \begin{cases} \frac{pq(s - f_1 - g_1)}{\Delta f \Delta g} & f_1 + g_1 \leq s \leq f_2 + g_1 \\ \frac{pq}{\Delta g} & f_2 + g_1 \leq s \leq f_1 + g_2 \\ \frac{pq(f_2 + g_2 - f_1 - g_1 - s)}{\Delta f \Delta g} & f_1 + g_2 \leq s \leq f_2 + g_2 \end{cases} \quad (4)$$

We make a few observations. The density function integrates to the probability pq. It becomes symmetrical if $g_2 - g_1 = f_2 - f_1$, and the length of the middle subrange becomes 0. When the lengths from two equal-size ranges are added, the sum has a 0.5 probability in either half of the combined range (although not a uniform distribution). When the lengths from two different size ranges are added, the sum approaches a uniform distribution as the difference between the two ranges becomes large.

B. K-WIDE HISTOGRAM

A histogram is an approximate representation of the distribution of a set of values. If we view the exact distribution as a curve, a k-bin histogram is an approximation of the curve using k line segments. A V-optimal k-bin histogram is one that has the minimal difference with the original curve. It is straightforward to use

dynamic programming to find the V-optimal k-bin histogram for an n-element curve (or set) in $O(n^2k)$ time and $O(n)$ space. As an approximate representation, histograms can used for curve data compression and simplification. In fact, each level of a Haar wavelet is a histograms with equal-size bins.

For the purpose of locality analysis, we have several distinct requirements not considered in other uses.

First, when modeling the effect of parallel threads, we have one set of values from each thread, and we need to add them and compute the new histogram. The number of values in a set is proportional to the length of the execution. Since it is too costly to record the set exactly, we need to use a histogram. The problem is how to record the information in the histogram so that we can estimate from these histograms the result of the addition of the sets of values.

Second, when the values represent the data volume, we care mostly about the comparison with power of two sizes, which are the sizes of cache on modern machines. For a data reuse, what mostly matters is whether the reuse distance is less than the cache size and not how much it is lower and higher than the cache size. This is in sharp contrast with the requirements of V-optimal histograms. Here the number of bins and their ranges are fixed a priori and the distribution inside each bin has almost no effect on the accuracy of the histogram, at least when a single thread is concerned [1].

K-wide histogram is a refined version of a logarithmic histogram. Each bin in a logarithmic histogram is divided into k sub-bins of equal length. For example, the bin $[8, 15]$ includes four sub-bins $[[8, 9], [10, 11], [12, 13], [14, 15]]$ in a 4-wide histogram. For our purpose, we use only integer boundaries for the range of a bin and do not subdivide when the size of a range is 1. If we view the contiguous bin ranges as points on the axis of positive numbers, the ranges of a logarithmic histogram can be represented by a 1-bit floating-point number. The ranges of a k-wide histogram are the points of a k-bit floating-point number. We discuss the properties of k-wide histograms in terms of the two requirements mentioned before.

A k-wide histogram has the property that a k_1-wide histogram includes a k_2-wide histogram when $k_1 \geq k_2$. Hence all k-wide histograms contain the basic logarithmic histogram, which is 1-wide histogram, required by the second condition.

If we have n_1 values in range $[a_1, b_1]$ and n_2 values in range $[a_2, b_2]$, their pairwise sum produces different sets of values depending how the values in the two ranges are paired. However, we are concerned with the range of all possible sums, which is $[a_1+a_2, b_1+b_2]$. The tricky problem is finding out the distribution of these values in logarithmic-size bins that we care about. If the two input bins are the same in two logarithmic histograms, that is, $a_1 = a_2 = 2^t$ and $b_1 = b_2 = 2^t - 1$, the resulting range is $[2^{t+1}, 2^{t+2} - 2]$, which is in $[2^{t+1}, 2^{t+2} - 1]$, and we have an accurate distribution. However, the accuracy become problematic when the two ranges differ. We call it the problem of asymmetrical ranges.

As an example of the problem of asymmetrical ranges, consider the two ranges $[4, 7]$ and $[1024, 2047]$. The sum lies in $[1028, 2058]$ but it is unclear how many values are still in range $[1028, 2047]$ and how many in $[2048, 4097]$. K-wide histogram alleviates the problem. If we use 2-wide histograms, the input will be in four ranges $[4, 5], [6, 7], [1024, 1535]$, $[1536, 2047]$. The value of the sum between the first two ranges and the third range definitely lies between $[1024, 2047]$. Although the distribution of the rest of the sum is still uncertain, we can guarantee accuracy for a subset of the values. The accuracy increases as we use k-wide histograms with more sub-bins. We now quantify the accuracy as a function of k.

To derive a closed-form result, we take the simple case where the input distribution is uniform across different bins and uniform within the same bin. It is different from a uniform distribution, so we call it bin-wise uniform for logarithmic bins. The following proposition shows the basic result.

PROPOSITION 1. *The accuracy of adding two k-wide histograms is either $1 - \frac{k-1}{k}$, $\frac{2k-1}{2k}$, or 100% for each pair of bins, assuming the input distribution is bin-wise uniform.*

Proof Sketch Let a and b be two logarithmic size bins and $a < b$ (the accuracy is 100% if $a = b$). The range $a + b$ spans across the upper boundary of b. Only one of the sub-bins of b, b_i, actually span the boundary. The portion of a that may cause b_i to produce an uncertain value is half of a if a is the immediate left neighbor of b or the entire a otherwise. The accuracy is $\frac{2k-1}{2k}$ in the first case and $\frac{k-1}{k}$ in the second. *end of sketch*

In most cases in our uses, the addition is between same size bins or neighboring bins, so the accuracy is closer to 100% and $\frac{2k-1}{2k}$ than to $\frac{k-1}{k}$.

Much of the literature in the use of histograms comes from the area of databases. Query optimization in particular makes extensive use of histograms especially for selection and join operations. Histograms are used by the query optimizer to estimate the number of disk operations of a select operator and the size of the output of a join operator. A good estimate would enable the database to select the operators and their order of oper-

[1] For set-associative cache, if the reuse distance is close to the cache size, the chance of it being a hit or miss depends on cache conflicts. However, the impact is often negligible since a high concentration at a particular distance is statistically unlikely.

ation so that the cost is minimized. Histograms are also used to provide fast, approximate answers to queries of large data sets. Most of these techniques use sampling possibly through multiple passes, and some guarantee a level of accuracy.