Thread Data Sharing in Cache: Theory and Measurement

Hao Luo*

Google
Mountain View, CA

Pengcheng Li

University of Rochester
Rochester, NY

Chen Ding

University of Rochester
Rochester, NY

Abstract

On modern multi-core processors, independent workloads often interfere with each other by competing for shared cache space. However, for multi-threaded workloads, where a single copy of data can be accessed by multiple threads, the threads can cooperatively share cache. Because data sharing consolidates the collective working set of threads, the effective size of shared cache becomes larger than it would have been when data are not shared.

This paper presents a new theory of data sharing. It includes (1) a new metric called the *shared footprint* to mathematically compute the amount of data shared by any group of threads in any size cache, and (2) a linear-time algorithm to measure shared footprint by scanning the memory trace of a multi-threaded program. The paper presents the practical implementation and evaluates the new theory using 14 PARSEC and SPEC OMP benchmarks, including an example use of shared footprint in program optimization.

1. Introduction

Many programs spend most of the time accessing data, and most of these accesses happen in cache. In the case of a multi-threaded program, threads interact in cache. The interaction can produce opposite effects. The effect may be *collaborative*: if one thread loads a data item that is also needed by another thread, the latter thread can take the data from the cache directly. On the other hand, the effect may be *interfering*: if the fetched item is not used by other threads, the data consumes cache space and leaves less room for others to use. We may also call these two effects *constructive* and *destructive*. The distinction is whether the data is shared.

* The work was done when the author was a graduate student at University of Rochester.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP '17, Feb. 4–8, 2017, Austin, Texas, USA..

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-4493-7/17/02...\$15.00.

<http://dx.doi.org/10.1145/3018743.3018759>

Figure 1 exemplifies these effects. In this experiment, we test two programs from the PARSEC suite [9] and SPEC OMP [34]. We use a dual-socket machine, run each program first on both sockets and then on just one socket, and measure how the miss ratio and running time change. Changing from two sockets to one cuts the available space of last-level caches (LLCs) by half. The figure shows opposite effects on the two programs: *facesim* has little data sharing and loses performance from the reduction in available cache, but *ilbdc* has significant data sharing and actually benefits from using just one LLC.

Figure 1: Thread data sharing in cache causes opposite effects when the size of the last level cache (LLC) is reduced by half: the miss ratio and running time are increased in *facesim* but decreased in *ilbdc*.

This paper presents a new theory of data sharing. In particular, the paper defines a new metric called *shared footprint*, formulates its mathematical properties, solves the problems of efficient measurement, and gives several uses in the analysis and optimization of data sharing in cache.

The shared footprint refers to the amount of memory that have been accessed by more than one threads within a time window. It has two important properties. First, it is *composable*. The data sharing by any group of threads can be computed from each thread's shared footprint. Second, it is *machine independent*. It is defined using the data access trace and does not depend on the cache size. Moreover, the miss ratio of any size cache can be computed from the shared footprint. Because of these two properties, a programmer can compute the miss ratio curve of any group of threads using shared footprint and can therefore mathematically derive the best thread grouping without exhaustive testing or repeated trials.

Data sharing in cache has long been studied since the invention of shared-memory processors and distributed shared memory (DSM). Early studies focus on common patterns of data sharing seen at the system or hardware level [4, 23]. They do not account for all manners of data sharing or the aggregate effect. Other studies use simulation, and the results are cache specific [10, 48]. To study multicore processors, several machine-independent models have been developed. The concurrent reuse distance (CRD) can derive the miss ratio of threaded code for all cache sizes [32, 39, 49, 50]. Other models such as StatStack [24] and the high-order theory of locality (HOTL) [28, 53] compute the miss ratio for independent co-run programs (that do not share data). The concurrent reuse distance models data sharing but is not composable. StatStack and HOTL are composable, only if there is no data sharing. CounterStack [47], Shards [45] and AET [29] are recent techniques to measure the miss ratio curve efficiently through sampling. However, they do not solve the problem of composing miss ratio curves when there is data sharing between threads. The shared footprint is the first composable model to handle data sharing.

The new theory has several limitations. First, the shared footprint is specific to thread scheduling. This is both a weakness, i.e. the sharing may change in another run, and also a strength, i.e. the sharing in all thread groups is captured for the same execution. It is like taking one group photo and producing a photo for every sub-group as if they were all taken at the same instant. Second, the miss ratio calculation of shared footprint assumes fully associative LRU cache. The effect of cache associativity and non-LRU replacement can often be estimated using reuse distance, which we can compute using shared footprint (Section 2.4). In addition, the calculation assumes a single-level, shared cache and does not consider the effect of a cache hierarchy or coherence between multiple caches. Finally, even knowing the exact miss ratio does not mean knowing the performance. The latter depends on many other factors, e.g. prefetching. We will evaluate the effect on performance and demonstrate the benefits of machine independence (Section 5)

The main contributions of the paper are as follows:

- Formal definitions and properties of the shared footprint including the miss ratio prediction for all thread groups in cache of any size in Section 2
- A one-pass algorithm to measure the shared footprint and its efficient implementation in Section 3
- Evaluation of the efficiency and accuracy of the shared footprint using 8 PARSEC [9] and 6 SPEC OMP benchmarks [34] in Section 4
- Demonstration in optimizing thread placement and quantifying constructive and destructive thread interaction in Section 5

2. Shared Footprint Theory

We first formalize the new metric shared footprint and then discuss the conversion between shared footprint and miss ratio.

2.1 Data Sharing Spectrum

Two parameters can be used to describe data sharing in cache: the number of sharers and the cache size. Figure 2 shows a description of data sharing of a multi-threaded benchmark, *dedup* [8]. For each cache size, it shows what portion of the cache is shared by how many threads, averaged over the entire execution. Across cache sizes, it shows how the sharing changes as the cache size increases. If we consider the cache of an infinite size, the sharing would be measured for the whole execution. The *cache sharing spectrogram* of Figure 2 shows that in finite size cache (up to 32MB), most data is either not shared or shared by no more than two threads, even if the benchmark uses 8 threads to process data [9].

Figure 2: Cache data sharing of *dedup* by different numbers of threads for different cache sizes, shown by a spectrogram.

2.2 Quantify Data Sharing in Timescales

During multi-threaded program's execution, we assume the memory references of each thread can be timed and ordered by a general clock. The clock can be CPU cycles, or a counter of memory references. The referred memory addresses (at a granularity of cache line) ordered by this clock form a *memory trace*. We call a time range in the memory trace a *window*. The quantity of shared data depends on the window in consideration. Longer windows are likely to contain more shared data than shorter windows.

The metric, *shared footprint*, measures the number of distinct data accessed by more than one thread within a given window. For instance, in window W_1 of Figure 3, there is only one datum, a , accessed by more than one threads, therefore the shared footprint of this window is 1. Shared footprint can be further parameterized by data's sharing degree. The amount of data accessed by k or more threads ($k \geq 1$) is called k + sharers footprint.

Figure 3: An example multi-threaded execution trace

$sfp_{k+}(l)$	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$
$k=1$	0	1	2	2	2	2	2	2	2
$k=2$	0	0	0	0.83	1.60	1.75	2	2	2
$k=3$	0	0	0	0	0	0.75	1.33	1.50	2
$k=4$	0	0	0	0	0	0	0	0.50	1

Table 1: $sfp_{k+}(l)$ of the example trace in Figure 3. For any trace, $sfp_{k+}(l)$ is non-decreasing in window length l and non-increasing in thread count k .

Figure 3 shows an example memory trace. Letters are used to represent data blocks. In window W_1 , there is only one data block accessed by two threads, a, which contributes to the 2+ sharers footprint. In W_2 , two data blocks, a and b, are accessed but not shared, therefore its shared footprint is 0. Similarly in W_3 , the contribution is one data block to the 2+ sharers footprint, a, and one data block to the 3+ sharers footprint, b.

To represent the sharing throughout the entire execution, we use average shared footprint, which is the average of the shared footprints of all same-length windows. Throughout this paper, we focus on the average $k+$ sharers footprint, which is denoted as $sfp_{k+}(l)$, where l is window length. Table 1 summarizes all $sfp_{k+}(l)$ of Figure 3. It shows the data sharing in all time scales and of all sharing degrees. Take window length 3 as an example. There are 6 length-3 windows. In every window except W_2 , there is exactly one data block accessed by two threads. The 2+ sharers footprint is therefore $\frac{1+0+1+1+1+1}{6} = 0.83$. Successive columns are for window lengths from 1 to 8. As the window lengthens, the footprint is monotone and non-decreasing. Successive rows are for sharer counts k from 1 to 4. As k increases, the footprint is also monotone and non-increasing.

2.3 The Shared Footprint

The shared footprint has following variations:

- **$k+$ sharers footprint** $sfp_{k+}(l)$: the average amount of data accessed by k or more threads in length- l windows, as defined in Section 2.2.

- **k sharers footprint** $sfp_k(l)$: the average amount of data accessed by *exactly* k threads in length- l windows, computed by $sfp_k(l) = sfp_{k+}(l) - sfp_{(k+1)+}(l)$.
- **read-only shared footprint** $sfp_{ro,k/k+}(l)$: the average amount of read-only data accessed by k or more threads.
- **read-write shared footprint** $sfp_{rw,k/k+}(l)$: the average amount of shared data with at least one thread writing to it, computed using the read-only shared footprint by $sfp_{rw,k/k+}(l) = sfp_{k/k+}(l) - sfp_{ro,k/k+}(l)$.
- **thread-group footprint** $sfp_g(l)$: the amount of data accessed by and only by the thread group g .

The thread-group footprint is a finer partition of the k sharers footprint. Let Ω be the set of all threads and g be a subset, $sfp_k(l) = \sum_{g \subseteq \Omega \text{ s.t. } |g|=k} sfp_g(l)$, for all $1 \leq k \leq |\Omega|$.

The shared footprint has two important properties:

Machine independence All variations of the shared footprint are defined entirely by the data access trace of a program and not specific to cache parameters. In comparison, the actual data sharing in cache changes with the cache size. The shared footprint is defined for all window length $l \geq 0$. We will show that all-length l allows us to compute the miss ratio of cache of all sizes, and the shared footprint can be measured once and reused on any machine regardless of its cache size.

Composability The shared footprint is *composable*. With different parameters, the shared footprint decomposes the working set of each execution window into disjoint subsets based on the pattern of sharing. For example in any execution window, each k sharers footprint contains a disjoint set of data for different k . The disjointness enables simple additive composition: the footprint of two sets of data is the arithmetic sum of the footprint of each set. Furthermore, the addition between the shared footprint preserves the window length l and hence its machine independence.

2.4 From Shared Footprint to Miss Ratio

From Shared Footprint to Footprint Xiang et al. introduced a metric, footprint, $fp(l)$, to denote the average amount of distinct data accessed in all windows of length l [21, 51, 52]. For a group of threads G , its footprint can be computed from the shared footprint as follows:

$$fp(l) = \sum_{g \subseteq G} sfp_g(l)$$

From Footprint to Miss Ratio The metric footprint can be converted to miss ratio. The conversion is the higher-order theory of locality (HOTL) developed by Xiang et al. [53] HOTL computes the miss ratio curve $mr(c)$, for all cache sizes $c \geq 0$, from the footprint $fp(l)$ as follows:

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

where Δl is a time increment. We take the smallest increment, $\Delta l = 1$, and compute the miss ratio as:

$$mr(c) = mr(fp(l)) = fp(l+1) - fp(l)$$

In other words, *the miss ratio function is the discrete derivative of the footprint function*. The derivative can be interpreted as a fractional miss, incurred every time we increase the window size by 1. The essence of this conversion by HOTL is to equate the average increase of working-set size, i.e. the fractional miss, with the increase of average working-set size, i.e. the footprint.

Consider as an example the sequence `abcdeffedcba`, which has 12 accesses to 6 data blocks. Table 2 shows three rows. The first row give the window length l from 0 to 3 and then to 11 and beyond. The second row gives the footprint $fp(l)$ from 0 to 6. These values are also cache sizes. At each size c , the converted miss ratio $mr(c)$ is the difference between two consecutive footprints, shown in the third row.

access trace: `abcdeffedcba`

l	0	1	2	3	...	10	≥ 11
$c = fp(l)$	0	1	1.9	2.8	...	5.6	6
$mr(c)$ (%)	100	91	89	76	...	33	0

Table 2: Conversion from timescale locality to cache locality: l is the window length, $fp(l)$ the footprint, c the cache size, and $mr(c)$ the miss ratio, computed by the HOTL equation $mr(c) = mr(fp(l)) = fp(l+1) - fp(l)$.

For the example, the conversion is fairly accurate. When the cache size is 0, the miss ratio is 100%. When the cache size is 1 and 1.9, the miss ratio is 91% and 89%. When the cache size increases to 2.8, the miss ratio drops to 76%. If we compute the miss ratio accurately, we have $mr(0) = 100\%$, $mr(1) = 92\%$, $mr(3) = 75\%$.

Set Associative and Non-LRU Cache HOTL shows that after the footprint is used to compute the miss ratio, the miss ratio can be used to compute the reuse distance as follows [53]:

$$P(rd = c) = mr(c-1) - mr(c)$$

From the reuse distance, prior techniques can be used to estimate the effect of cache associativity, including the early model by Smith which assumes uniform access to cache sets [27, 41] and recent extensions of per-set reuse distance by Nugteren et al. [36] and Sen and Wood [40] which handles the general access, uniform or not. In addition, Sen and Wood show techniques to model PLRU, RANDOM and NMRU policies [40]. The shared footprint may be calculated per cache set, although the extension is outside the scope of this paper.

3. Measuring Shared Footprint

In this section, we formalize a set of concepts to establish an important theorem which leads to an algorithm to measure shared footprint and its variations in one pass over the memory trace.

3.1 Counting Data Sharing in All Windows

Key Observations It is computationally prohibitive to enumerate all windows and measure their data sharing. The efficient solution we will present next is based on two high-level ideas. First, compared to counting window by window, it is easier to take the perspective of data and count the *contribution* of each datum to the average. Its contribution is the number of windows within which the datum is accessed (present). Second, compared to measuring the presence of data, it is easier to first measure the number of windows when a datum is not accessed (absent) and then obtain the presence count by subtracting the absence count from the total count.

Throughout the rest of this section, we denote the trace length as N and the set of all accessed data as D . The window ending at time i and j in the trace (excluding both endpoints) is denoted as (i, j) . Let $W_d^k(l)$ be the number of length- l windows in which the datum d is accessed by at least k threads. The shared footprint can be computed by adding W_d^k from all data and dividing by the number of windows:

$$sfp_{k+}(l) = \frac{\sum_{d \in D} W_d^k(l)}{N - l + 1} \quad (1)$$

For datum, a window either has at least k sharers or has less than k sharers, therefore $sfp_{k+}(l)$ can equivalently be obtained by counting the windows with less than k sharers. To help count such windows, we select two types of special windows:

Definition For datum d , a window is a *sub- k interval* ($k > 0$) if (1) inside the window, d is accessed by $k-1$ threads, and (2) the window is maximal in that any window enclosing it must have k or more threads accessing d .

Excluding the windows ending at trace boundaries, for an interval to be sub- k , its both ends, i.e. at times i and j , must access d , and the accessing threads at i and j , t_i and t_j , must not access d inside the window.

Example 1. In Figure 3, W_1 is sub-1 interval to a and W_2 is sub-1 to b . W_3 a sub-2 interval.

Lemma 3.1. Let datum d be accessed by k or more threads in an execution. d has at most $k-1$ sharers in a window W if and only if W is nested in a sub- k interval of d .

Proof Assume d has at most $k-1$ sharers in W . We can expand W as much as possible. When it stops, we have a

sub- k interval covering W . On the other hand, by definition, any part of a sub- k interval has at most $k - 1$ sharers. ■

Another special type of windows is a sub-type of sub- k intervals called *switch* as follows:

Definition A sub- k interval (i, j) for datum d is called a **sub- k switch** ($k > 0$) if the accesses to d at times i, j are by two different threads, i.e. $t_i \neq t_j$.

Example 2. W_3 in Figure 3 is a sub-2 switch and W_1 is a sub-1 switch. But W_2 is not a switch.

Two sub- k intervals may overlap. In Figure 3, window $(1, 5)$ and window $(3, 7)$ are sub-2, to datum a , and they overlap at $(3, 5)$. The next lemma proves that $(3, 5)$, as the overlap of two sub- k intervals, is a switch.

Lemma 3.2. For datum d , a window is covered by n sub- k intervals if and only if it is covered by $n - 1$ sub- $(k - 1)$ switches.

Proof

⇒: Suppose the n sub- k intervals that cover the window are $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ and $b_1 < b_2 < \dots < b_n$. Because they are maximal, they can not nest each other. Therefore, $a_1 < a_2 < \dots < a_n$. Since they all contain the window, we have $a_n < b_1$. We next show that $(a_2, b_1), (a_3, b_2), \dots$, and (a_n, b_{n-1}) are $n - 1$ sub- $(k - 1)$ switches.

Without loss of generality, consider the sub- k interval (a_i, b_i) . Thread t_{b_i} accesses d at the trailing end but not inside the interval. Thread $t_{a_{i+1}}$ accesses d inside the interval, so $t_{a_{i+1}} \neq t_{b_i}$.

Let a' be the leftmost position in the interval (a_i, b_i) such that (a', b_i) has exactly $k - 2$ sharers for d . a' must exist and $a_i < a'$, since (a_i, b_i) already has $k - 1$ sharers. Thread $t_{a'}$ accesses d at a' and must differ from any sharers in (a', b_i) ; otherwise, a' would not be the leftmost position.

We now prove that a' is a_{i+1} by contradiction. If the two are not the same, there are two cases with regard to their relative position:

- If $a_{i+1} < a'$, then (a_{i+1}, b_i) contains (a', b_i) , and the two are not equal. (a', b_i) has $k - 2$ sharers and $t_{a'}$ adds a new sharer. Since (a_{i+1}, b_{i+1}) is sub- k interval, $t_{a_{i+1}}$ differs from those $k - 1$ sharers. In this case, the interval (a_i, b_i) would have k sharers, contradicting the assumption that it is a sub- k interval.
- If $a' < a_{i+1}$, then (a', b_i) contains (a_{i+1}, b_i) , and the two are not equal. Since (a', b_i) contains $k - 2$ sharers, we can expand it to the right to form a sub- k interval by including at least one position to include b_i . Since t_{b_i} differs from $t_{a'}$ and these $k - 2$ sharers, the sub- k interval can be formed. This sub- k interval is not any of the (a_k, b_k) , contradicting the fact that the window is covered by n sub- k intervals.

Therefore, a' is a_{i+1} . There are exactly $k - 2$ sharers between a_{i+1} and b_i . Since $t_{a_{i+1}}$ must differ from t_{b_i} , (a_{i+1}, b_i) is a sub- $(k - 1)$ switch. ■

⇐: Suppose that the $n - 1$ sub- $(k - 1)$ switches are $(a_1, b_1), \dots, (a_{n-1}, b_{n-1})$. Because t_{a_j} differs from t_{b_j} and from every thread accessing d in (a_j, b_j) , intervals (a_j, b_{j+1}) with $j = 1, \dots, n - 2$ are $n - 2$ sub- k intervals. In addition, stretching (a_1, b_1) to the left to include a_1 and (a_{n-1}, b_{n-1}) to the right to include b_{n-1} yield two more and a total of n sub- k intervals covering the window. ■

Figure 4: Illustration for the first part of the proof of Lemma 3.2, The red segment is the window of concern. Each (a_i, b_i) is a sub- k interval. The proof shows that (a_{i+1}, b_i) is a sub- $(k - 1)$ switch.

For datum d , a sub- k interval of length L will cover exactly $L - l + 1$ length- l windows, which have at most $k - 1$ sharers. In Figure 3, for length 4, the sub-3 interval of a , $(1, 7)$, has $5 - 4 + 1 = 2$ length-4 windows, $(2, 5)$ and $(3, 6)$, in which a has at most 2 sharers. By Lemma 3.1, any window with at most k sharers must be covered by a sub- k interval. The windows not covered by any sub- k interval then must have $k +$ sharers. Hence counting $k +$ sharers footprint can be converted to accumulating all windows nested in all sub- k intervals. Due to the overlap of sub- k intervals, the same window may be overcounted. But by Lemma 3.2, the effect of overcounting can be canceled using sub- k switches.

Let $SI_k^d(l)$ and $SS_k^d(l)$ be the count of datum d 's length- l sub- k intervals and sub- k switches respectively. Then we have the following result:

Theorem 3.3. The total count of length- l windows with fewer than k sharers of d is

$$\sum_{i=l}^N [SI_k^d(i) - SS_{k-1}^d(i)] \times (i - l + 1)$$

Proof $\sum_{i=l}^N [SI_k^d(i) \times (i - l + 1)]$ is the number of length- l windows with fewer than k sharers, but with overcounting. The overcounting is removed by subtracting $\sum_{i=l}^N [SS_{k-1}^d(i) \times (i - l + 1)]$. The result counts the windows covered by a sub- k interval exactly once. By Lemma 3.1, it is the number of the length- l windows with less than k sharers. ■

Let D_{k+} be the set of the data with $k +$ sharers and $|D_{k+}|$ be its cardinality. We denote $\sum_{d \in D_{k+}} SI_k^d(i)$, the cumulative sub- k intervals for all data in D_{k+} , as $SI_k(i)$. Similarly

$\sum_{d \in D_{k+}} SS_k^d(i)$ is denoted as $SS_k(i)$. By Theorem 3.3 and Equation (1), the $k+$ sharers footprint is:

$$\begin{aligned} sfp_{k+}(l) &= |D_{k+}| - \frac{\sum_{d \in D_{k+}} \sum_{i=l}^N [SI_k^d(i) - SS_{k-1}^d(i)](i-l+1)}{N-l+1} \\ &= |D_{k+}| - \frac{\sum_{i=l}^N [SI_k(i) - SS_{k-1}(i)](i-l+1)}{N-l+1} \end{aligned} \quad (2)$$

Algorithm 1 shows that all $|D_{k+}|$, $SI_k(i)$ and $SS_k(i)$ can be obtained by scanning through the trace once.

3.2 The Measurement Algorithm

Algorithm 1 profiles the memory trace once and computes the shared footprint using Equation 2 (line 27).

Algorithm 1 Algorithm of $k+$ sharers footprint

Require: A memory trace of length N

Ensure: None

```

1: procedure MEMORY_REFERENCE_CALLBACK( $d, tid$ )
  ▷ This routine is called upon reading in a new element
  in the trace,  $d$  is the datum,  $tid$  is its accessing thread
2:    $t_{now} \leftarrow$  current time
3:    $k \leftarrow 1$ 
4:   for  $e$  in  $d$ 's list do
5:      $SI_k[t_{now} - e.time] \leftarrow SI_k[t_{now} - e.time] + 1$ 
6:     if  $tid$  equals to  $e.tid$  then
7:       break
8:     end if
9:      $SS_k[t_{now} - e.time] \leftarrow SS_k[t_{now} - e.time] + 1$ 
10:     $k \leftarrow k + 1$ 
11:  end for
12:  if  $tid$  is not found in  $d$ 's list then
13:     $SI_k[t_{now}] \leftarrow SI_k[t_{now}] + 1$ 
14:  end if
15:  promote  $tid$  and  $t_{now}$  to the list head
16: end procedure
17: procedure COMPLETION ▷ This routine is called after
  reading all elements of the trace
18:  for every datum  $d$  having appeared in trace do
19:     $k \leftarrow 1$ 
20:    for every element  $e$  in  $d$ 's list do
21:       $SI_k[N - t_{now}] \leftarrow SI_k[N - t_{now}] + 1$ 
22:       $|D_{k+}| \leftarrow |D_{k+}| + 1$ 
23:       $k \leftarrow k + 1$ 
24:    end for
25:  end for
26: end procedure
27:  $sfp_{k+}(l) = |D_{k+}| - \frac{\sum_{i=l}^N [SI_k(i) - SS_{k-1}(i)] \times (i-l+1)}{N-l+1}$ 

```

The algorithm maintains a sorted list for each datum d , which stores each thread's last access time to d . At every

memory access, this list will be traversed (line 4-11). For each element e in the list, the difference between the current time t_{now} and the recorded time of the element $e.time$ is accumulated in the histograms SS and SI accordingly (line 5 and line 9). The routine *Completion* accumulates the maximal intervals ending at the trace boundary. (line 21).

Complexity The time complexity is $\mathcal{O}(NT + MT)$, where M is the total footprint, N is the trace length, and T is thread count. The space complexity is $\mathcal{O}(2NT + MT)$ if we compute the shared footprint for every window length. However, it is sufficient to maintain logarithmically many window lengths in practice by grouping the N window lengths in $\mathcal{O}(\log N)$ ranges. $SI_k(i)$ and $SS_k(i)$ now count the windows in the i th range. $i \times SI_k(i)$ and $i \times SS_k(i)$ are also profiled in $\mathcal{O}(\log N)$ ranges independent of $SI_k(i)$ and $SS_k(i)$. The space overhead is reduced to $\mathcal{O}(2T \log N + MT)$.

Measuring Thread-group Footprint Thread-group footprint is measured similarly as $k+$ sharers footprint. Instead of profiling SS_k and SI_k based on the sharing degree, we can refine them into specific thread groups. By doing so, the space overhead becomes $\mathcal{O}(2^T N)$, which is too expensive. To balance precision and cost, we choose to maintain per-group SS_k and SI_k for only a few window lengths, for example, all lengths of 2^i ($i > 12$). This choice compresses the profile and requires only $\mathcal{O}(2^T)$ space. To recover the full profile for all window lengths, we use interpolation to estimate between the selected lengths. Interpolation has a stable accuracy because shared footprint averages over all sliding windows and therefore produces smooth curve.

Measuring Read-only and Read-write Shared Footprint

To measure the $k+$ sharers read-only footprint, we can profile the *read-only* sub- k intervals and switches, which exclude all the write accesses. These two types of windows have the same properties as given in the two lemmas in Section 3. By combining with Theorem 3.3, we can count the windows with fewer than k read-only sharers. Using Equation (2), we have the read-only $k+$ sharers footprint. By taking the difference between sfp_{k+} and its read-only sub-part, we have the read-write $k+$ sharers footprint.

3.3 Implementation Details

Clock Algorithm 1 requires reading the time at each memory reference (line 2). We could have used a globally synchronized software counter to denote time, but its cost would increase linearly with the number of threads. As an optimization, we use a loosely synchronized, per-core hardware time stamp counter (TSC) [30], which are available on most modern processors and sufficient for our analysis.

Sampling Bursty sampling has been effective in reducing the cost of profiling [3, 11, 17], including recent uses in measuring the footprint [28, 29, 53]. We use bursty sampling and see two benefits. The first is greater efficiency. In our experiments, some OpenMP benchmarks take several hours

in a native parallel run, and the profiling on full trace would take too long to finish. Their parallel behavior is repetitive and could be captured by a few samples. The second is the more accurate miss ratio prediction for programs with phase behavior. In comparison, full-trace analysis measures the average of the entire trace and does not capture phase behavior. Sampling is effective in addressing this problem. We will evaluate sampling analysis in the next section.

4. Evaluation

We implemented Algorithm 1 using Intel’s binary rewriting tool Pin [33]. The computed shared footprint is used to guide the optimization as discussed in Section 2.3. We evaluate shared footprint analysis on (1) its time/space overhead, (2) the sensitivity of its profiled results on different machines and (3) the accuracy of its miss ratio prediction.

4.1 Experimental Setup

To evaluate shared footprint analysis itself, we profile 8 pthread benchmarks from PARSEC [9]. To evaluate miss ratio prediction, another 6 OpenMP benchmarks from SPEC OMP 2012 [34] are included. The selected benchmarks represent a wide range of parallel programming patterns such as data-parallel (*blackscholes*), pipeline-style (*dedup*) and task-parallel (*facesim*). The OpenMP benchmarks are widely used for evaluation of OpenMP applications. We exclude some benchmarks because they 1) are not memory-intensive (*swaptions*); or 2) require too much memory that does not fit in our test machine (*mgrid331* and *ilbdc* on ref size); 3) or launch too many threads that our analysis can not handle for now (*x264*); or 4) fail to compile on our testbed (*bt331*).

Two different platforms were set up in the experiment. One was used for shared footprint collection and the other for performance testing. The machine for collection has 12 cores (Intel Xeon E5649 “Westmere”) with each 6 sharing a 12MB LLC and the performance-test machine is featured with 8 cores (Intel Xeon E5520 “Nehalem”), with each 4 sharing an 8MB LLC.

4.2 Profiling Cost

Table 3 summarizes the time for measuring $k+$ sharers footprint and thread-group footprint. The baseline is the execution time on 8 threads without profiling shared footprint.

The slowdown of performing $k+$ sharers analysis is between 142x and 503x for 7 benchmarks and 747x for *stream-cluster*. Thread-group analysis may be twice as long. The slowdown factors are larger than other tools in locality profiling. For example, the cost of reuse distance analysis is between 52x and 426x (153x average) for SPEC INT 2006 as reported in [53]. However, the previous work profiles sequential programs. In our experiment, the baseline is 8-threaded parallel execution time. The relative overhead would be much closer if we compared with the sequential running time.

Figure 5: The difference between sharing spectrums due to different thread interleaving (on two different machines). The three bars for each benchmark represent three cache sizes (from left): 32KB, 256KB, and 8MB.

The overhead can be dramatically reduced by sampling. We periodically sample contiguous memory accesses at run-time. Each sampling phase terminates when 32MB memory has been profiled. Between two sampling phases is a fixed time interval. Its specific length depends on the desired sampling rate and its value ranges from 10^8 to 10^{10} memory accesses, as shown in Table 4. The overhead of sampling analysis can be contained in 250x except for *bodytrack*. Five benchmarks are within 100x. *Bodytrack* shows one limitation of our sampling approach: if the memory footprint of the application is small, it would take too long for one single sampling phase to terminate and the sampling will degenerate into full profiling.

4.3 Effect of Interleaving

A parallel execution has many sources of non-determinism due to hardware and OS scheduling. Pin’s instrumentation and our analysis also perturb the parallel execution, and the effect has been shown to reduce the speed difference between threads in relative terms since all threads spend most time in the instrumentation code [32].

Our analysis forcefully serializes memory accesses from different threads whenever it is necessary. Such interleaving is one source of the non-determinism that our analysis introduces among others. To quantify how the interleaving affects the validity of results obtained from the analysis, we draw and compare the *cache sharing spectrums* on the two testbeds. An example cache sharing spectrum of the benchmark *dedup* has been given in Figure 2, which shows the portion of sharing of different degrees in all cache sizes.

Figure 5 shows the absolute difference between the spectrums on the two machines. Each bar is the Manhattan distance of the two spectrums. One previous conjecture was that the non-deterministic effect was more visible in short time periods than in long time periods, and more in small cache than in large cache. The spectrum results show that

benchmarks	input	native exec.	k + sharers		thread group	
		time (sec.)	time (sec.)	slow-down	time (sec.)	slow-down
blackscholes	simlarge	0.460	146	317x	189	411x
bodytrack	simlarge	0.860	241	280x	650	755x
canneal	simlarge	0.404	57	142x	83	207x
dedup	simlarge	6.649	1730	260x	2372	356x
facesim	simlarge	1.974	828	419x	1263	639x
ferret	simlarge	2.100	750	357x	1067	508x
fluidanimate	simlarge	0.679	342	503x	595	876x
streamcluster	simlarge	1.627	1216	747x	2336	1435x

Table 3: Cost of measuring k -sharers and thread-group footprints using full-trace profiling

benchmarks	interval between two sampling phases (No. mem. accesses)	slow-down
blackscholes	10^{10}	19x
bodytrack	N/A	778x
canneal	2×10^8	14x
dedup	2×10^9	245x
facesim	2×10^9	114x
ferret	2×10^9	47x
fluidanimate	10^{10}	57x
streamcluster	10^{10}	10x

Table 4: Cost of measuring thread-group footprints using sampling

this intuitive view is not true. While the largest cache sees the smallest variation (under 0.3%) in 6 programs, the variation is largest (4% to 10%) in the other 2 programs.

Prior work collects sharing results using cache simulation. Bienia et al. found that because of different thread interleaving enforced by the simulator, the number of memory accesses varies by $\pm 4.7\%$, and the amount of sharing changes by $\pm 15.2\%$ in 4-way set associative 4MB cache [9]. Simulation is cache specific. Shared footprint, however, captures the effect of different interleaving on all cache sizes in particular in the exact impact on the miss ratio.

As we see in these results, data sharing in cache changes from program to program, cache size to cache size, execution to execution, and input to input (which we don’t evaluate in this paper). Through the derived metric of sharing spectrum, shared footprint enables quantitative analysis of these variations.

4.4 Thread Group Composition

To evaluate the composition of shared footprint, we compare the thread-group footprint composed from the shared footprint (Section 2.4) against the directly measured footprint by running Xiang et al.’s algorithm [52]. The comparisons were done in two different executions, so the difference in the results may be partly due to the difference in trace interleaving. We examined all 4-thread groups in each of the benchmarks on 8-threaded runs. For the sake of clarity in plotting, we only plot two special 4-thread groups, the maximal one and the minimal one, in Figure 6. Between these two curves lies the other 68 groups’ results.

The composition results are close to direct measurement. We found similar accuracy for other thread groups. The benefit of composition is significant: instead of measuring different 4-thread groups 70 times (once for each group), shared footprint measures the execution once and derives all 70 footprints. In fact, it derives the footprint for all thread groups. The benefit increases exponentially as we analyze programs with more threads.

Figure 6: Footprint composition for all 4-thread groups. The x -axis is the window length in logarithmic scale and y -axis is the footprint. Only the two groups with the smallest and largest footprints are shown. The other 68 groups fall between the two extremes. *Canneal* and *blackscholes*, not shown here, are similar to *streamcluster*.

The benchmarks in Figure 6 are grouped horizontally for 3 types of parallelism: pipeline-parallel (*ferret* and *dedup*), task-parallel (*facesim* and *bodytrack*) and data-parallel (*fluidanimate* and *streamcluster*). We discovered that these three categories correspond to different degrees of thread symmetry. The threads’ memory behavior is most asymmetrical among pipeline workloads but most symmetrical among data-parallel programs. Shared footprint is an effective way to check for thread symmetry or asymmetry.

Figure 7: Comparison of the predicted and measured miss ratio for 14 programs using 1 and 2 LLCs. For each program, the first 3 bars (blue) are miss ratios for one LLC, while the next 3 bars (red) are miss ratios on two LLCs. Each LLC is 8MB. The full-trace analysis is used for the first 8 programs (up to *ferret*).

4.5 Miss Ratio Prediction

Shared footprint can be converted to the miss ratio curve as discussed in Section 2.4. We also compared the miss ratios predicted using shared footprint against the results read from hardware performance counters [30]. The following performance events were measured:

- MEM_INST_RETIRED.LOADS,
- MEM_INST_RETIRED.STORES
- OFFCORE_RESPONSE_0.DATA_IN.L3_MISS

Figure 7 shows the 8-thread executions of 14 benchmarks from PARSEC and SPEC OMP. For applications with symmetric threads, we show the miss ratio of a single thread. For applications with asymmetric threads, e.g. *ferret* and *dedup*, only the thread with heaviest workload is shown. We run the SPEC OMP benchmarks on the *ref* input size (except *ilbdc*, which we run *train* size) only with sampling.

The shared footprint predicts the miss ratio accurately except in 4 tests, *bots-spar* with 1 LLC, *md* with 2 LLCs and *fluidanimate* with 1 LLC and 2 LLCs. The full-trace shared footprint also has a large error in *facesim* at 2 LLCs, but the error is corrected by sampling.¹ The errors are unlikely due to footprint composition, which we have shown in Section 4.4 to be accurate. The likely cause is program phase behavior, as it is the case for *facesim*.

In predicting the miss ratio of sequential (SPEC 2006) programs, previous work has shown improvements over full-trace footprint analysis by different sampling techniques. While fixed-interval sampling did not show a significant improvement on a single cache size [53], adaptive bursty sampling (ABF) showed clear improvement for one pro-

gram (*perlbench*) for a fixed range of cache sizes (up to 8MB) [28], and average eviction time (AET) sampling for four programs for all cache sizes [29]. Among the 16 tests (of the 8 parallel programs) where the full-trace analysis is used, sampling improves significantly only *facesim* at 2 LLCs. We leave it a future study whether the phase effect differs in parallel code than in sequential code and whether more advanced sampling can further improve the prediction accuracy.

5. Applications

This section shows two uses of the shared footprint: optimizing thread placement and quantifying constructive and destructive thread interaction.

5.1 Optimizing Thread-Core Mapping

It is beneficial to let the threads that share data to share cache. For example, a technique called faithful scheduling separately schedules threads from different applications [38]. Within the same application, it is still a difficult task to find the best way to group threads since the number of possible placements grows exponentially with the number of threads. On the other hand, optimization may significantly improve performance, especially for programs with asymmetric threads like *dedup*. *Dedup* is a 5-stage data-processing pipeline, where three stages are parallel and every stage shows dramatically different locality.

Figure 8 summarizes *dedup*'s throughput under different thread placements on a two-hyperthreading quad-core processor. The performances vary as high as 60%. The difference is largely due to the use of per-core L2 cache. Shared footprint analysis can quickly evaluate *dedup*'s cache performances for all placements. More specifically, we label each placement with its predicted four L2 cache miss ratios through shared footprint composition and then select

¹ Note that *facesim* in Figure 1 is different because it was tested on 12MB LLCs, although the miss ratio on 8MB LLCs is similar, so is the performance.

Figure 8: Performances of thread-core mappings, sorted by their speedup over the worst mapping. Shared footprint analysis identifies Pareto mappings without testing.

the *Pareto* placements for further evaluation [54]. In our scenario, a placement is Pareto if *no other placement can reduce the miss ratio in one cache without increasing it in the other cache*. We use shared footprint to find Pareto placements and show them as black solid dots in Figure 8. Only 1/3 placements are Pareto. We then tested them to find the optimal one. Compared to testing all mappings, this method is 37% faster.

5.2 Effective Cache Occupancy

In the shared cache, the thread interaction may be constructive and destructive. Under the assumption of single-level fully associative LRU cache, the shared footprint can precisely quantify these two types of interaction.

Let $\Omega = \{T_i\}$ be the set of threads sharing a fully associative LRU cache of size c . Let the size of program data be larger than c and the window length l_0 exist such that the total footprint is the cache size, $sfp_{1+}(l_0) = c$. For each thread T_i , we compute its *effective cache occupancy* $eco(T_i)$ by the amount of data T_i accesses in the total footprint at l_0 , $sfp_{1+}(l_0)$. We may compute it in different ways based on the thread-group footprints and their composibility (Section 4.3). One way is to start with the amount of data only T_i accesses and then add in the amount of data T_i shares with every peer group. The equation is as follows:

$$eco(T_i, c) = sfp_{\{T_i\}}(l_0) + \sum_{g \subseteq (\Omega - T_i)} sfp_{g+T_i}(l_0)$$

where $sfp_{1+}(l_0) = c$, g is every peer group of threads of T_i , and $sfp_{g+T_i}(l_0)$ the average amount of data T_i shares with g in a window of length l_0 . We may compute the total occupancy from the individual occupancy. Alternatively, it is simpler to use the k sharers footprints by taking each k sharers footprint and multiplying it with k :

$$eco(\Omega, c) = \sum_{k=1}^{|\Omega|} k sfp_k(l_0)$$

It is easy to prove $eco(\Omega, c) \geq c$, because $eco(\Omega, c) \geq \sum_{k=1}^{|\Omega|} sfp_k(l_0) = sfp_{1+}(l_0) = c$, i.e. every data in the total footprint $sfp_{1+}(l_0)$ is accessed by at least one thread, and any data shared by k threads is counted exactly k times. Earlier in Section 2.1, we use $sfp_k(l)$ as the sharing spectrum. In the spectrum, a vertical line at the cache size c shows the portions of data accessed by different number of threads. The portion used by k threads is counted k times in the total occupancy, which makes the occupancy greater than c .

West et al. modeled the cache occupancy in random replacement cache [46]. For LRU cache, Brock et al. showed through HOTL (Section 2.4) that for a set of independent programs that share the cache of size c (but do not share data), there exists of a cache partition for each program P_i , called the *natural cache partition (NCP)* $ncp(P_i, c)$, such that the miss ratio of P_i in the shared cache equals to the miss ratio of P_i in a private cache of size $ncp(P_i, c)$, and the total size of natural partitions is c , i.e. $\sum_i ncp(P_i, c) = c$ [12].

If threads do not share data, the effective cache occupancy is the same as the natural cache partition, i.e. $eco(T_i, c) = ncp(T_i, c)$. In the absence of data sharing, the total occupancy is $eco(\Omega, c) = c$.

The constructive thread interaction is quantified by the total occupancy, $eco(\Omega, c)$. The more data is shared, the greater this total is, and the better the threads uses the cache as a group.

The destructive thread interaction is shown by the individual occupancy, $eco(T_i, c)$. If a thread has a large working set but a much smaller individual occupancy, the thread has trouble keeping its data in cache because of the interference from other threads.

6. Related Work

We focus the discussion on the theory and techniques that model the performance of general rather than specific configurations of shared cache.

Reuse Distance without Data Sharing The earlier models were given by Suh et al. [42] for time-sharing systems and Chandra et al. [15] for multicore processors. Although different terminology was used, the common design is to compose the reuse distance of a program with the footprint of its peers, as explained by Xiang et al., who also showed that just mere reuse distance and (by inference) miss ratio are not composable [51]. Reuse distance is expensive to measure precisely [2, 37], although the cost can be reduced by approximation algorithms [47, 56], sampling [7, 24, 39, 55], static analysis [6, 14, 16], OS and hardware support [5, 13, 43, 57] and parallelization [18, 26, 35, 39]. Recent theories use reuse time. CAPS and StatStack are first models to analyze shared cache entirely from reuse time and have a linear-time complexity. In CAPS, the composition is based on distinct data blocks per cycle (DPC) [31]. In StatStack,

the composition is done by first composing the private reuse time and then converting it to reuse distance [24].²

When used to analyze shared cache, these techniques must assume independent programs without data sharing. The reason is fundamental — the composition, whether between reuse distance and footprint or between reuse times, is invalid when there is data sharing.

Footprint without Data Sharing The study of memory sharing was pioneered by Denning and others through the development of the working set theory [19]. Thiebaut and Stone defined *footprint* as a program’s data in cache [44]. Falsafi and Wood redefined it to mean data blocks accessed by a program, so the data in cache is its “projection” [25]. Early studies used footprint to model interference in time-shared cache [1, 25, 44]. The footprint was measured for a single window length [25, 44] or estimated for all lengths [15, 42], including the working set theory (property P2 in [20]) in 1972 and recently DPC in CAPS [31]. Xiang et al. gave a linear-time algorithm to precisely measure the footprint for all-length windows [52]. Their higher-order theory (HOTL) converts between the footprint and the miss ratio and reuse distance [53], so the footprint models are now as efficient as CAPS and StatStack.

Shared footprint solves the more difficult problem than Xiang et al. [52], because it measures not just the footprint but also the number of sharers. It subsumes the footprint of Xiang et al. as a sub-case, i.e. $fp(l) = sfp_{1+}(l)$. Furthermore, shared footprint gives the read-only and read-write footprint for sequential applications as it does for parallel code.

Concurrent Reuse Distance Early studies focus on common patterns of data sharing seen at the system/hardware level [4, 23]. They did not account for all manners of data sharing, nor for the aggregate effect. Other studies use simulation, and the results are cache specific [10, 48].

Recent solutions developed the concept of concurrent reuse distance [22, 32, 49, 50], also called multicore reuse distance [39]. Concurrent reuse distance gives the shared cache performance for all cache sizes. The miss-ratio prediction is accurate and not affected by phase behavior (unlike shared footprint). Data sharing is modeled by first measuring the amount of shared data between threads in the entire run and then inferring its effect in smaller windows through probabilistic models [22, 32]. For loop-based code, Wu and Yeung developed scaling models to predict the the concurrent and private reuse distances (CRD/PRD) profiles for different thread counts and data input sizes [49]. They used the model to study the scalability of multicore cache hierarchies, to separate the shared cache locality into interference-based and sharing-based components, and to construct a new profile type to model the effect of cluster caches [50].

² The naming in StatStack is different. Its reuse distance is our reuse time, and its (LRU) stack distance is our reuse distance.

Reuse distance considers the accesses from multiple threads together. It focuses data reuse rather than data sharing. It does not measure the number of sharers.

Shared Footprint Falsafi and Wood gave a simple model of sharing where all processes share the same footprint, measured for a single window-length [25]. Shared footprint in this paper measures it for all window-lengths. More importantly, it counts the number of threads accessing the same data, adding a new type of locality — the thread-count locality. Thread-count locality is necessary for thread-group composition. In addition, it is necessary to derive metrics of cache occupancy and cache sharing spectrum (Section 5.2). The former is important for understanding performance scaling on multicore, while the latter is important for cache analysis since it shows precisely how interleaving, program input, and other factors affect data sharing in cache.

Thread-count locality in this work extends both window- and access-based locality, for the first time making both types of metrics composable for multi-threaded code. We have shown this in composing thread-group footprint (Section 4.4) and thread-group miss ratio (Section 4.5), both are new and impossible with previous techniques.

7. Summary

We have defined shared footprint, a collection of metrics parameterized by the number of sharers, thread groups, access types, and by derivation, cache of all sizes. We have developed a linear-time algorithm to measure all these metrics in a single pass. The efficiency is further improved by parallelization and sampling. We have measured and analyzed data sharing in 14 multi-threaded applications from PARSEC and SPEC OMP, including sharing spectrum, effect of interleaving, and optimization of thread-core mapping to improve performance by up to 60%. Shared footprint adds the missing piece in the locality theory and augments previous theories of reuse distance and footprint to use thread count, thread composition and access-type analysis.

Acknowledgment

Michael Scott suggested the use of the hardware time stamp counter (Section 3.3), and a PPOPP reviewer measuring the destructive effect of thread cache sharing (Section 5.2). The authors also thank Jacob Brock, Dong Chen, Sandhya Dwarkardas, Michael Gage, Rahman Lavaee, Chencheng Ye, Peng Zhao and other PPOPP reviewers for their comments which have helped the development and the presentation of the paper. The research is supported in part by the National Science Foundation (Contract No. CCF-1629376, CNS-1319617, CCF-1116104), IBM CAS Faculty Fellowship, and a grant from Huawei. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

- [1] Anant Agarwal, Mark Horowitz, and John L. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.
- [2] George Almasi, Calin Cascaval, and David A. Padua. Calculating stack distances efficiently. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*, pages 37–43, Berlin, Germany, June 2002.
- [3] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of PLDI*, pages 168–179, Snowbird, Utah, June 2001.
- [4] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of ISCA*, pages 125–134, 1990.
- [5] Erik Berg and Erik Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of ISPASS*, pages 20–27, 2004.
- [6] Kristof Beyls and Erik H. D’Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.
- [7] Kristof Beyls and Erik H. D’Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of High Performance Computing and Communications. Springer. Lecture Notes in Computer Science*, volume 4208, pages 220–229, 2006.
- [8] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of PACT*, pages 72–81, 2008.
- [10] Christian Bienia and Kai Li. Fidelity and scaling of the PARSEC benchmark inputs. In *Proceedings of the 2010 International Symposium on Workload Characterization*, December 2010.
- [11] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: proportional detection of data races. In *Proceedings of PLDI*, pages 255–268, 2010.
- [12] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *Proceedings of ICPP*, 2015.
- [13] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of PACT*, pages 339–349, 2005.
- [14] Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of ICS*, pages 150–159, 2003.
- [15] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, pages 340–351, 2005.
- [16] Arun Chauhan and Chun-Yu Shei. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of ICS*, pages 295–304, 2010.
- [17] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of PLDI*, pages 199–209, 2002.
- [18] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. A highly parallel reuse distance analysis algorithm on GPUs. In *Proceedings of IPDPS*, 2012.
- [19] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.
- [20] Peter J. Denning and Stuart C. Schwartz. Properties of the working set model. *Communications of the ACM*, 15(3):191–198, 1972.
- [21] Chen Ding and Trishul Chilimbi. All-window profiling of concurrent executions. In *Proceedings of PPOPP*, 2008. *Poster paper*.
- [22] Chen Ding and Trishul Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical Report MSR-TR-2009-107, Microsoft Research, August 2009.
- [23] Susan J. Eggers and Randy H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of ISCA*, pages 373–382, 1988.
- [24] David Eklov, David Black-Schaffer, and Erik Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of HiPEAC*, pages 147–157, 2011. *Best paper*.
- [25] Babak Falsafi and David A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1):104–130, 1997.
- [26] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. In *Proceedings of IPDPS*, 2012.
- [27] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.
- [28] Xiameng Hu, Xiaolin Wang, Yechen Li, Yingwei Luo, Chen Ding, and Zhenlin Wang. Optimal program symbiosis in shared cache. In *Proceedings of CCGrid*, June 2015.
- [29] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *Proceedings of USENIX ATC*, pages 351–364, 2016.
- [30] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Number 325462-051US. June 2014.
- [31] Yunlian Jiang, Kai Tian, and Xipeng Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of HiPEAC*, pages 201–215, 2010.
- [32] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of CC*, pages 264–282, 2010.
- [33] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.
- [34] Matthias S. Müller, John Baron, William C. Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, Pavel Shelepugin, Matthijs van Waveren, Brian Whitney, and Kalyan Kumaran. SPEC OMP2012 – an application benchmark suite for parallel systems using OpenMP. In *Proceedings of the International Workshop on OpenMP*, pages 223–236, Berlin, Heidelberg, 2012. Springer-Verlag.
- [35] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of IPDPS*, 2012.
- [36] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri E. Bal. A detailed GPU cache model based on reuse distance theory. In *Proceedings of HPCA*, 2014.
- [37] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.
- [38] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. No more backstabbing... a faithful scheduling policy for multithreaded programs. In *Proceedings of PACT*, pages 12–21, 2011.
- [39] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.
- [40] Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *Proceedings of SIGMETRICS*, pages 279–292, 2013.
- [41] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of ICSE*, 1976.

- [42] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of ICS*, pages 1–12, 2001.
- [43] David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of ASPLOS*, pages 121–132, 2009.
- [44] Dominique Thiébaud and Harold S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.
- [45] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.
- [46] Richard West, Puneet Zaro, Carl A. Waldspurger, and Xiao Zhang. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 44(4):19–29, 2010.
- [47] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. Characterizing storage workloads with counter stacks. In *Proceedings of OSDI*, pages 335–349. USENIX Association, 2014.
- [48] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *The 22nd annual international symposium on Computer architecture (ISCA '95)*, pages 24–36, 1995.
- [49] Meng-Ju Wu and Donald Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of PACT*, pages 264–275, 2011.
- [50] Meng-Ju Wu, Minshu Zhao, and Donald Yeung. Studying multicore processor scaling via reuse distance analysis. In *Proceedings of ISCA*, pages 499–510, 2013.
- [51] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPoPP*, pages 91–102, 2011.
- [52] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.
- [53] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Proceedings of ASPLOS*, pages 343–356, 2013.
- [54] Seyed Majid Zahedi and Benjamin C. Lee. REF: resource elasticity fairness with sharing incentives for multiprocessors. In *Proceedings of ASPLOS*, pages 145–160, 2014.
- [55] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of ISMM*, pages 91–100, 2008.
- [56] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM TOPLAS*, 31(6):1–39, August 2009.
- [57] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of ASPLOS*, pages 177–188, 2004.