Abstract

Cache sharing on a multicore processor is usually competitive. In multi-threaded code, however, different threads may access the same data and have a cooperative effect in cache. This report describes a new metric called shared footprint and a new locality theory to measure and analyze parallel data sharing in cache. Shared footprint is machine independent, i.e. data sharing in all cache sizes, not just one cache size and compositional, i.e. data sharing in all thread sets, not just one set. The report gives a single-pass, parallel algorithm for measurement and evaluates the new metric using 14 PARSEC and SPEC OMP benchmarks, including a use in improving the performance of multi-threaded code.
1 Introduction

Modern parallel programs are run on multicore machines with shared cache. Many programs are memory bound, spending most of the time accessing data. Most of the accesses happen in cache, which has multiple layers. The largest is the last level cache (LLC), which is shared by all cores. In the case of simultaneous multi-threading (SMT), threads can share all levels of cache.

As a program executes, its threads interact in the shared cache in two ways. They can cooperate: if a data block is shared, i.e. needed by multiple threads, one thread loads it into the shared cache for everyone to use. They can also compete: if a data block is private, i.e. needed by just one thread, it occupies cache space, leaving less room for other threads.

To demonstrate this, we run two programs first on 1 processor and then on 2 processors. Figure 1 shows how they change after given twice as much cache: `ilbdc` incurs more LLC misses and runs slower, while `facesim` incurs fewer misses and runs faster. The difference is data sharing in cache: with little or no data sharing, using 2 LLCs is beneficial; otherwise, it is more efficient to use 1 LLC.

![Performance On 2 LLCs](image)

<table>
<thead>
<tr>
<th></th>
<th>access locality</th>
<th>window locality</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared cache without data sharing</td>
<td>reuse distance (i.e. LRU stack distance), reuse time</td>
<td>footprint (i.e. working set in cache)</td>
</tr>
<tr>
<td>shared cache with data sharing</td>
<td>concurrent reuse distance</td>
<td>shared footprint (this work)</td>
</tr>
</tbody>
</table>

Figure 1: (Left) Two programs seen opposite effects when given twice as much (but separated) cache. (Right) A new theory needed to understand and derive the results on the left. See Section 8 for a review.

In this report, we define a concept called *shared footprint* to measure data sharing in cache. A data block is *cache shared* if it is used by multiple threads during a continuous period when it is in cache.

Shared footprint has two theoretical properties. First, it measures the data sharing in all execution windows, so it can predict performance in all cache sizes. Second, shared footprint is compositional in that it requires only one single-pass analysis over an interleaved execution trace and then is able to compose the data sharing in all thread groups.

There have been many techniques to analyze data sharing. In many studies of cache coherence in shared-memory processors and distributed shared memory (DSM), data sharing was simulated precisely but for specific cache. Recent work has developed concurrent reuse distance, which can derive the miss ratio for all cache sizes [19, 24, 31, 32]. Concurrent reuse...
distance, however, is measured not composed. Without composition, the type of effects seen in Figure 1 can only be measured, but not derived.

Shared footprint is specific to interleaving. This is both a weakness, i.e. data sharing may change in another run, and also a strength, i.e. the behavior of 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 it were taken specifically for the sub-group in the same instant. In comparison, testing cannot reproduce all thread-group results.

We do not solve the problem of how to optimize data sharing, neither do we consider the effect on coherence misses. Instead we solve its prerequisite — how to quantify data sharing. We build a theoretical foundation to support measurements and models that can be used to minimize the number of cache misses. The study makes five contributions.

- One-pass, linear-time algorithms to measure data sharing in all cache sizes, all thread groups, different number of sharers and different access types (read-only and read-write). (Section 3)

- Speed improvement using parallelization and sampling. (Section 4)

- Derived data sharing metrics including miss ratio, concurrent reuse distance, and two new metrics: cache sharing spectrogram and effective cache size. (Section 5)

- Validation and uses on a multicore processor for PARSEC and SPEC OMP benchmarks, including an example of optimizing thread-core mapping. (Section 6 and 7)

- A short synthesis to show how shared footprint extends existing locality theory for shared cache. (Section 8)

2 Data Sharing in Cache

We explain the notion of data sharing by an example.

A Multi-threaded Program  Figure 2 shows the skeleton code for a threaded application. When executed, the code constructs a 4-stage pipeline. The first three stages are parallel, and each has two threads. The last stage is sequential and has just one thread.

Each input data is processed in four stages by four threads. Most data sharing happens in 8 four-thread groups. The 8 groups and the amount of sharing, in (64-byte) blocks, are shown in the table in Figure 2. The groups do not contain thread pairs (1,2), (4,5) or (6,7) in the same group, because the three pairs operate three parallel stages and do not share data. Thread 3 operates the last stage and is a sharer in every group. We call these results whole-execution data sharing. Data sharing in cache is more complex because it depends on the cache size and management. Only in fully-associative cache of an infinite size is the data sharing in cache the same as whole-execution data sharing.
```cpp
// threads 1,2
for i from 0 to 1
create_thread;

// threads 4,5
for i from 0 to 1
create_thread;

// threads 6,7
for i from 0 to 1
create_thread;

// thread 3
create_thread;
```

Figure 2: (Left) The PARSEC program `dedup` uses 7 threads to implement a 4-stage pipeline. (Middle) Whole-execution data sharing in different groups of 4 threads, equivalent to cache data sharing for infinite size cache. (Right) Data sharing in cache by different number of threads for different size (fully associative LRU) cache, shown by a spectrogram. In the 32MB cache, most data have less than 3 sharers (during their lifetime in cache).

**Data Sharing in Cache** Later in Section 5.2 we will define a cache sharing spectrogram. For each cache size, it shows what portion of cache is shared by how many threads, averaged over the entire execution. A sharing spectrogram shows how the sharing changes as the cache size increases. The full spectrogram includes whole-execution data sharing at the right end of the spectrogram as the sharing in the infinite-size cache.

The spectrogram for `dedup` is shown in Figure 2. It shows data sharing for cache size up to 32MB, most data in cache is either not shared or shared by no more than three threads. It is strikingly different from whole-execution data sharing, where most data are shared by four threads.

## 3 Shared Footprint Analysis

Shared footprint is entirely defined by window statistics. We will introduce size and other factors of cache when presenting derived metrics in Section 5.

### 3.1 Types of Shared Footprint

In one time window, the $k+$ threads sharing is the amount of distinct data accessed by $k$ or more threads ($k \geq 1$). In all windows of length $l$, the $k+$ sharers footprint, $\text{sfp}_{k+}(l)$, is the average $k+$ threads sharing in all windows of length $l$.

$$\text{sfp}_{k+}(l) = \frac{\text{sum of } k+ \text{ threads sharing in length-}l \text{ windows}}{\text{number of length-}l \text{ windows}}$$
Next we list the types of shared footprints:

- **k+ sharers footprint** $sfp_{k+}(l)$: the average amount of data accessed by at least $k$ threads in all length-$l$ windows.

- **$k$ sharers footprint** $sfp_k(l)$: the average amount of data accessed by exactly $k$ threads, computed from $k+$ sharers footprints by $sfp_k(l) = sfp_{k+}(l) - sfp_{(k+1)+}(l)$.

- **Read-only footprint** $sfp_{ro,k/k+}(l)$: the average amount of read-shared data by $k$ or $k+$ threads.

- **Read-write footprint** $sfp_{rw,k/k+}(l)$: the average amount of shared data with at least one thread writing to it.

The $1+$ sharers footprint includes all the data accessed in windows. It gives the total footprint. The $2+$ sharers footprint includes all shared data. The difference between $1+$ and $2+$ footprints is the data not shared, i.e. accessed by only 1 thread. We may call it the unshared footprint. $K$ sharers footprint is partitioned by read-only and read-write footprints, i.e. $sfp_k(l) = sfp_{ro,k}(l) + sfp_{rw,k}(l)$.

Figure 3 shows an interleaved trace. Each letter represents a data block, and the number overhead is the thread id. First consider individual windows. In $W_1$, the unshared footprint is 2. In $W_2$, the 2 sharers footprint is 1. In $W_3$, the 2 sharers footprint is 1, 3 sharers footprint is 1, and unshared footprint is 0.

![Figure 3: An example multi-threaded execution trace](image)

Let’s now consider the average footprint for window length 3. There are 6 length-3 windows. In every window except $W_1$, there is exactly one datum accessed by two or more threads. The average $2+$ sharers footprint is therefore $\frac{1+0+1+1+1+1}{6} = 0.83$. The average footprints for all $k$ and $l$ values are given in Table 1. When the window length is 1, only one thread can access. The total footprint is 1. The shared footprint is 0. Successive rows are for window lengths from 1 to 8. Across window lengths, the footprint is monotone and non-decreasing as $l$ increases. Successive columns are for sharer counts from 1 to 4. For the same window length, the footprint is monotone and non-increasing as $k$ increases.

### 3.2 Computing the $k+$ Sharers Footprint

In this section, we mathematically simplify the problem of footprint measurement. We denote the trace length as $N$ and the set of all data as $D$. Instead of enumerating all
windows and taking their average, we count each datum’s “contribution” to the average. While measuring the footprint of a time window is hard, we will show that counting the “contribution” of a datum is simpler and can be done efficiently.

First, we convert the measurement problem into a counting problem. Suppose datum $d$ is accessed by $t$ threads in the whole execution. Let $W^k_d(l)$ be the number of length-$l$ windows in which $d$ is accessed by at least $k$ threads. The shared footprint can be computed by adding the contribution from all data $d$ and dividing it by the number of windows:

$$sf_{k+}(l) = \frac{\sum_{d \in D} W^k_d(l)}{N - l + 1}$$ (1)

It can be shown that counting the windows with fewer than $k$ sharers is easier than counting those with at least $k$ sharers. We define a type of window:

**Definition** For datum $d$, a window $(i, j)$ is a **maximal sub-$k$ sharing interval** ($k > 0$) if (1) inside the window (between $[i + 1, j - 1]$), $d$ 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 $d$.

For brevity, we call the maximal sub-$k$ sharing interval simply the **sub-$k$ interval**. Excluding boundary windows, i.e. $i > 0$ or $j < N$, for an interval to be sub-$k$, both ends of the interval, accesses at times $i, j$, must be accesses to $d$, and the accessing threads, $t_i$ and $t_j$, must not access $d$ inside the interval. For example in Figure 3, for datum $b$, $W_1$ is a sub-1 interval, and $W_3$ is a sub-2 interval.

**Lemma 3.1** Let datum $d$ be accessed by $k$ or more threads in an execution. $d$ does not have $k$ 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. Since $d$ has at least $k$ sharers in the execution, the expansion cannot carry on forever. When it stops, we have a sub-$k$ interval covering $W$. On the other hand, it is trivial to see that any part of a sub-$k$ interval of $d$ has at most $k - 1$ sharers. \[\square\]
We pick out 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. $t_i \neq t_j$.

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

Two sub-$k$ intervals may overlap. Consider the expansion of a sub-$(k-1)$ switch into a sub-$k$ interval. The expansion can occur on either side but not on both sides at the same time. By expanding on each side, we obtain two overlapping sub-$k$ intervals from a sub-$(k-1)$ switch.

A numerical relation between intervals and switches is given in Lemma 3.2.

**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**

$\Rightarrow$: Suppose the $n$ sub-$k$ intervals that cover the window are $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ and $b_1 < b_2 < \ldots < b_n$. Because they are maximal, they can not contain each other. Therefore, $a_1 < a_2 < \ldots < a_n$. Since they all contain the window, we have $a_n < b_1$. We next show that $(a_2, b_1), (a_3, b_2), \ldots$, 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 $(a_1, b_1), ..., (a_{n-1}, b_{n-1})$. 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 $j = 1, ..., n - 2$ 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.](image)

Let $l$ be the length of the windows we wish to count for those that have $k$ sharers for datum $d$. If there is a sub-$k$ interval of length $L$, it will cover $(L - l + 1)^+$ length-$l$ windows, where the function $(x)^+$ is $x$ if $x > 0$ and 0 otherwise. In these windows, the number of sharers is $k - 1$ or less.

From Lemma 3.1, any window with fewer than $k$ sharers is covered by a sub-$k$ interval. Therefore, the windows not covered by any sub-$k$ interval have at least $k$ sharers. Hence the original problem of $k+$ sharers footprint is converted to one that counts the number of windows that have sub-$k$ sharers.

Sub-$k$ intervals may overlap, so the same window may be overcounted. From Lemma 3.2, the effect of overcounting can be canceled using sub-$k$ switches.

Let $SI^d_k(l)$ and $SS^d_k(l)$ be the count of length-$l$ sub-$k$ intervals and sub-$k$ switches respectively. Then we have following result:

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

$$
\sum_{i=1}^{N} [SI^d_k(i) - SS^d_{k-1}(i)](i - l + 1)^+
$$

**Proof** As stated above, the quantity $\sum_{i=1}^{N} [SI^d_k(i)(i - l + 1)^+]$ is the number of length-$l$ windows with fewer than $k$ sharers, but with overcounting. The overcounting is removed by subtracting $\sum_{i=1}^{N} [SS^d_k(i)(i - l + 1)^+]$. The result counts the windows covered by a sub-$k$ interval exactly once. From Lemma 3.1, this value is the number of the length-$l$ windows with less than $k$ sharers. □
From Theorem 3.3 and Equation (1), we can derive the formula for the $k+$ sharers footprint:

$$sfp_{k+}(l) = |D_{k+}| - \frac{\sum_{d \in D_{k+}} \sum_{i=1}^{N}[SI_k^d(i) - SS_k^d(i)](i-l+1)}{N-l+1}$$

$$= |D_{k+}| - \frac{\sum_{i=l}^{N} \sum_{d \in D_{k+}} [i \cdot SI_k^d(i) - i \cdot SS_k^d(i)]}{N-l+1}$$

$$+ \frac{\sum_{i=1}^{N} \sum_{d \in D_{k+}} [SI_k^d(i) - SS_k^d(i)](l-1)}{N-l+1}$$

$$= |D_{k+}| - \frac{\sum_{i=l}^{N} I_k(i) - i \cdot SS_k(i)}{N-l+1}$$

$$+ \frac{\sum_{i=1}^{N} I_k(i) - SS_k(i)(l-1)}{N-l+1}$$  (2)

The formula depends on three terms. $D_{k+}$ is the set of data shared by at least $k$ threads in the execution, and $|D_{k+}|$ is its cardinality. $SI_k(i)$ is $\sum_{d \in D_{k+}} SI_k^d(i)$, the cumulative sub-$k$ intervals for all data in $D_{k+}$. Similarly $\sum_{d \in D_{k+}} SS_k^d(i)$ is denoted as $SS_k(i)$. All three terms can be obtained by scanning the whole trace in one pass, as will be shown in the next section.

3.3 One-Pass Linear-Time Measurement

Algorithm 1 profiles the memory trace in one pass and computes the shared footprint using Equation (2) (line 28).

The algorithm maintains a list for each data block to store the last access times by each thread ordered by increasing time. At every memory access, the list of the accessed block will be traversed once (line 5-12). For each element $e$ in the list, the time interval between the current time $t_{now}$ and the recorded time of the element $e.time$ will be accumulated in the histograms $SS$ and $SI$ accordingly (line 6 and line 10). At program finish, the maximal intervals at the boundary are also collected (line 22).

The time complexity is $O(NT + MT)$, where $M$ is the total footprint, $N$ is the trace length, and $T$ is thread count.

The space complexity is $O(2NT + MT)$ if we compute the shared footprint for every window length. However, it is sufficient only to maintain logarithmically many window lengths in practice by grouping the $N$ window lengths in $O(\log N)$ ranges. $SI_k(i)$ and $SS_k(i)$ is now counting the windows in the $i$th range. $i \cdot SI_k(i)$ and $i \cdot SS_k(i)$ are also profiled in $O(\log N)$ ranges independent of $SI_k(i)$ and $SS_k(i)$. The space overhead is reduced to $O(2T \log N + MT)$.

3.4 Thread Group Sharing

A thread-group footprint or a group footprint in short is the amount of data accessed by a particular group of threads. Group footprints are finer partitions of a $k$ sharers footprint,
Algorithm 1 Algorithm of \(k+\) sharers footprint

Require: A trace of execution

Ensure: None

1: procedure MEMORY_REFERENCE_CALLBACK(cache_line, thread_id) \(\triangleright \) This routine is called upon reading in a new element in the trace, cache_line is the data, thread_id is its accessing thread

2: \(t_{now} \leftarrow N\)

3: \(N \leftarrow N + 1\)

4: \(k \leftarrow 1\)

5: for e in cache_line’s list do

6: \(SI_k[t_{now} - e.time] \leftarrow SI_k[t_{now} - e.time] + 1\)

7: if thread_id equals to e.tid then

8: break

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 thread_id is not found in cache_line’s list then

13: \(SI_k[t_{now}] \leftarrow SI_k[t_{now}] + 1\)

14: end if

15: promote thread_id and \(t_{now}\) to the list head

16: end procedure

17: procedure POST_PROCESSING \(\triangleright \) This routine is called after reading all elements of the trace

18: for every cache line c having appeared in trace do

19: \(k \leftarrow 1\)

20: for every element e in c’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: \(sfp_{k+}(l) = D_{k+} - \sum_{i=0}^{N-l} \frac{SI_k(i) - SS_{k-1}(i)(i-l+1)^{+}}{N-l+1}\)

showing which \(k\) threads are sharing which portion. For example in an execution of 3 threads: \(T_1, T_2, T_3\), the 2 sharers footprint is made of three parts: \(\{T_1, T_2\}\) group footprint, \(\{T_2, T_3\}\) group footprint, and \(\{T_1, T_3\}\) group footprint. The 2 sharers footprint is completely partitioned by the three non-overlapping group footprints.

Group footprints are measured similarly as \(k+\) sharers footprints. While counting the contribution of each datum to \(k+\) sharers footprints, we can be more precise and count the contribution to each thread group and each window length. The main overhead is space. If we count for all window lengths, we will need additional \(O(2^TN)\) space (one value per
thread group per window length). To balance precision and cost, we choose the window length \(2^i\) \((i > 12)\), which requires almost only \(O(2^T)\) space. Then we can get the precise thread group footprints at the above lengths and use interpolation to estimate between these lengths. Interpolation has a stable accuracy because of a mathematical property—the footprint is a concave function over window lengths (when trace length \(n \gg w\) the window length) [35].

With thread group sharing, we can predict data sharing in all thread groups for all window lengths. One example has been shown in Figure 2 as the data sharing in 4-thread groups for one window length (the entire execution).

3.5 Read-only and Read-write Footprint

To measure the \(k+\) sharers read-only footprint, we need a few changes in the algorithm. For datum \(d\), we profile its maximal read-only sub-\(k\) sharing intervals and switches for \(k > 0\). These two concepts have the same properties as given in the two lemmas in Section 3.2. By combining these using Theorem 3.3, we can count the windows with fewer than \(k\) read-only sharers. Using the read-only footprint in Equation (2), we have the \(k+\) sharers read-only footprint. By taking the difference between the \(k+\) sharers footprint and its read-only subpart, we have the read-write \(k+\) sharers footprint. Further taking the differences, we have the \(k\) sharers read-only and read-write footprint.

4 Parallel Measurement and Sampling

We start with the most serialized solution based on a global lock and then remove as much serialization as possible using two techniques. In addition, we combine parallel measurement with sampling.

**Global Lock** In our design, threads are instrumented so each analyzes its own accesses. At each memory access, a thread increments the global time counter and updates the list of last access times for the accessed datum. In the initial solution, a thread uses a global lock to serialize the time increment and list update in order to avoid data race.

**Atomic Time Counter and Privatization** We implement global time as an atomic variable, so the increment is done much faster using a hardware atomic instruction. In addition, we privatize the access record. Privatization is possible since the counting of reuse intervals \((SS\) and \(SI)\) and total footprints \((D_k)\) is commutative. When a thread exits, it adds its private records atomically to the global records. With privatization, updates on reuse histogram are completely parallel.

**Distributed Time Stamp Counter (TSC)** Instead of a global time counter, which still requires synchronization to access, we use the per-core hardware time stamp counter (TSC), available on most modern processors. At every memory access, the time is obtained
by reading the host-core’s TSC register. Meanwhile, the thread also counts the number of its memory accesses. When computing the footprint, we measure the window length in CPU cycles but also compute the average CPU cycles per memory access. Combining the two, we obtain the shared footprint in window lengths in terms of memory accesses.

**Sampling** We periodically take a subtrace and measure the shared footprint. This has two benefits. First, it reduces the cost. For example, some OpenMP benchmarks are so long running that there are several hours in a native run, for which the full profiling failed to finish. With sampling, we found that the parallel behavior is repetitive and could be quickly captured by a few samples. Second, sampling improves the accuracy of miss ratio prediction. For programs with different phases, sampling can generate different shared footprint profiles for each phase. We found that combining each phase’s miss ratio leads to more accurate prediction.

### 5 Derived Metrics for Shared Cache

#### 5.1 Miss Ratio and Reuse Distance

To convert between the footprint and the miss ratio, we use a theoretical result by Xiang et al. [35] The miss ratio function $mr(c)$ is the derivative of the footprint function $fp(l)$. Formally,

$$mr(c) = mr(fp(x)) = \frac{fp(x + \Delta x) - fp(x)}{\Delta x}$$  \hspace{1cm} (3)

Intuitively, if footprint $fp(x) = c$, then $x$ is the average time for a program to populate the cache of size $c$ with the data in the last $x$ accesses. After an additional time $\Delta x$, the footprint is increased by $fp(x + \Delta x) - fp(x)$, which are new data and therefore misses in the cache. The ratio is the miss frequency, hence the miss ratio. Xiang et al. gave the condition for correctness and showed that it is accurate on real hardware with set-associative caches [35].

Footprint can be used to compute reuse distance [35]. Reuse distance has been used as a shorter name for LRU stack distance [21]. It shows the hit or miss by each memory access for all sizes of fully-associative LRU cache as well as set-associative [26, 16] and non-LRU cache [25]. It has many uses in program optimization and memory management [37].

The reuse distance for a group of threads running in shared cache is called concurrent reuse distance [19, 24, 31, 32]. Shared footprint can derive the footprint for every thread group, its shared cache miss ratio in all cache sizes and its concurrent reuse distance.

#### 5.2 Cache Sharing Spectrogram

Assume the cache is fully-associative LRU. Given a program execution, if we stop at a random point and look at the data accessed in the most recent $l$-long window, the footprint
is $sfp_{1+}(l)$, which means the accesses in the window fills the cache of size $sfp_{1+}(l)$. The shared footprints $sfp_k(l)$ are the portions in this cache accessed by $k$ threads. We call $sfp_k(l)$ the sharing spectrum for this cache size. If we show the sharing spectrums in all cache sizes, we have the sharing spectrogram. An example was shown earlier in Figure 5. The spectrogram captures data sharing in different cache sizes. As an instrument for generating the spectrogram, shared footprint analysis may be called the cache sharing spectrograph.

5.3 Effective Cache Size Scaling

Cache has a fixed size, and hence we have a fundamental tension in multicore scaling: the cumulative CPU power scales, but the total cache capacity does not. This problem cannot be solved by building processors with larger cache. In fact, the larger the cache, the more disproportionately the single-thread performance may benefit.

There is a unique problem in multicore performance scaling. As a program uses more cores, the size of shared cache does not change. With sufficient sharing, however, a thread may keep the same amount of data in cache even when there are more threads running. While the physical cache size does not change, the effective cache size may scale with the number of threads.

We compute the effective cache size. Take the cache spectrum for cache size $c$. The effective cache size counts each portion by multiplying the size by the number of sharers, i.e. $k$ sharers footprint is counted $k$ times. An equivalent way is to calculate the size of data contained in the cache for each thread and compute their arithmetic sum. The first calculation needs only the $k$ sharers footprint, while the second calculation needs thread group sharing (Section 3.4). The first method is easier and sufficient.

6 Evaluation

This section evaluates the speed and accuracy of shared footprint analysis and the effect of interleaving.

6.1 Experimental Setup

To evaluate, we profile the parallel benchmarks from PARSEC and SPEC OMP Benchmark Suite [4] using the binary rewriting tool Pin from Intel [20]. We evaluate our tool on 8 Pthread benchmarks from PARSEC and 6 OpenMP benchmarks from SPEC OMP 2012. These 8 PARSEC benchmarks represent a wide range of parallel programming patterns such as data-parallel (blackscholes), pipeline-style (dedup) and task-parallel (facesim). The 6 SPEC OMP benchmarks are widely used for evaluation of OpenMP applications. We exclude some benchmarks because of 1) too small memory usage (swaptions from PARSEC); 2) too large memory usage for our test machine (mgrid331 and ilbdc on ref size from SPEC OMP); 3) too many threads created (x264 from PARSEC); and 4) failure to compile (bt331 from SPEC OMP). Two machines are used for data collection and performance testing. One machine has 8 physical cores (2.27GHz Xeon E5520), with each 4 sharing 8MB LLC.
Table 2: Cost of measuring $k$-sharers and thread-group footprints using full-trace profiling (Section 3.3)

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>native (sec.)</th>
<th>$k$-sharers</th>
<th>time (sec.)</th>
<th>slowdown</th>
<th>thread group</th>
<th>time (sec.)</th>
<th>slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>0.460</td>
<td></td>
<td>146</td>
<td>317x</td>
<td></td>
<td>189</td>
<td>411x</td>
</tr>
<tr>
<td>bodytrack</td>
<td>0.860</td>
<td></td>
<td>241</td>
<td>280x</td>
<td></td>
<td>650</td>
<td>755x</td>
</tr>
<tr>
<td>canneal</td>
<td>0.404</td>
<td></td>
<td>57</td>
<td>142x</td>
<td></td>
<td>83</td>
<td>207x</td>
</tr>
<tr>
<td>dedup</td>
<td>6.649</td>
<td></td>
<td>1730</td>
<td>260x</td>
<td></td>
<td>2372</td>
<td>356x</td>
</tr>
<tr>
<td>facesim</td>
<td>1.974</td>
<td></td>
<td>828</td>
<td>419x</td>
<td></td>
<td>1263</td>
<td>639x</td>
</tr>
<tr>
<td>ferret</td>
<td>2.100</td>
<td></td>
<td>750</td>
<td>357x</td>
<td></td>
<td>1067</td>
<td>508x</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>0.679</td>
<td></td>
<td>342</td>
<td>503x</td>
<td></td>
<td>595</td>
<td>876x</td>
</tr>
<tr>
<td>streamcluster</td>
<td>1.627</td>
<td></td>
<td>1216</td>
<td>747x</td>
<td></td>
<td>2336</td>
<td>1435x</td>
</tr>
</tbody>
</table>

Table 3: Cost of measuring thread-group footprints using sampling

<table>
<thead>
<tr>
<th>benchmarks</th>
<th>input</th>
<th>$R$</th>
<th>$C$ (MB)</th>
<th>slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>native</td>
<td>$10^8$</td>
<td>32</td>
<td>19x</td>
</tr>
<tr>
<td>bodytrack</td>
<td>simlarge</td>
<td>$10^8$</td>
<td>32</td>
<td>778x</td>
</tr>
<tr>
<td>canneal</td>
<td>simlarge</td>
<td>$2 \times 10^8$</td>
<td>32</td>
<td>14x</td>
</tr>
<tr>
<td>dedup</td>
<td>simlarge</td>
<td>$2 \times 10^9$</td>
<td>32</td>
<td>245x</td>
</tr>
<tr>
<td>facesim</td>
<td>simlarge</td>
<td>$2 \times 10^9$</td>
<td>32</td>
<td>114x</td>
</tr>
<tr>
<td>ferret</td>
<td>simlarge</td>
<td>$2 \times 10^9$</td>
<td>32</td>
<td>47x</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>native</td>
<td>$10^{10}$</td>
<td>32</td>
<td>57x</td>
</tr>
<tr>
<td>streamcluster</td>
<td>native</td>
<td>$10^{10}$</td>
<td>32</td>
<td>10x</td>
</tr>
</tbody>
</table>

The other has 12 2-hyperthreading cores (2.53GHz Xeon E5649), with each 6 sharing 12MB LLC.

### 6.2 Profiling Cost

To measure the overhead, we use the 8-core machine. The baseline is the native execution time on 8 threads with no instrumentation. Three implementations in Section 4 are compared in Table 2. We only instrumented the build-in region of interest for PARSEC benchmarks.

The slowdown of $k$-sharers analysis is between 142x and 503x for 7 of the benchmarks and 747x for *streamcluster*. Thread group analysis may be twice as long. The slowdown factors are larger than usual in locality profiling. For example, the cost of reuse distance analysis is between 52 and 426 times (153 average) for SPEC 2006 as reported in a recent study [35]. However, the previous work profiles sequential programs. In our experiment, the baseline is unmodified 8-threaded parallel execution time. The relative overhead would be much closer if we compared with the sequential running time. Next, we show the improvement we can obtain through sampling.
We take a sample at regular intervals of length $R$ for footprint $C$. We fix $C$ to be 32MB, and the interval length $R$ ranges from $10^8$ to $10^{10}$, shown in Table 3. The overhead of sampling analysis is contained in 250x slowdown except for bodytrack. Most of the benchmarks have slowdown within 100x. Bodytrack shows one limitation: if the memory footprint of the application is small, the threshold $C$ may never be reached, and the sampling will degenerate into full profiling.

### 6.3 Effect of Interleaving

A parallel execution has many sources of non-determinism due to hardware and OS scheduling. The instrumentation and 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 [19].

Cache sharing spectrum can be used to quantify how thread interleaving affects cache utilization. To demonstrate, we run the 8-threaded PARSEC tests on two machines, one with 8 cores and the other 12 cores.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Sharing spectrum of 8-thread executions of 8 PARSEC benchmarks for 32KB, 256KB and 8MB cache.}
\end{figure}

Figure 5 shows the sharing spectrums for PARSEC programs for three cache sizes: 32KB, 256KB, and 8MB. In most programs, most of the cache, over 90%, contains unshared data. To magnify the shared portions, we plot the $y$-axis in a logarithmic scale to show the shared 10% or 1% of the cache.
Figure 6 shows the absolute difference between the spectrums on the two machines. Each bar is the Manhattan distance between two spectrums. One previous conjecture about interleaving 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 the 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.

Previous work collects sharing results by simulation. Bienia et al. found that because of different thread interleaving, the number of memory accesses varies by ±4.7%, and the amount of sharing changes by ±15.2% in 4-way set associative 4MB cache [4]. Simulation is cache specific. With shared footprint, we can now show the interleaving-induced variation in all cache sizes.

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

6.4 Thread Group Composition

We examine all 4-thread groups in each of the benchmarks on 8-threaded runs. It would be difficult to see clearly if we plot the results for all 70 4-thread groups for all the benchmarks, but 2 groups for each program can be shown clearly. Showing the two thread groups with the smallest and the largest footprint. The space between them demonstrates the range of the remaining 68 results.

To evaluate the accuracy, we compare the thread group footprint predicted from the shared footprint with the directly measured footprint by running Xiang et al.’s algorithm [34]
on the (interleaved) accesses from only the threads in the thread group. We re-run a program for prediction and measurement, so the difference may due partly to interleaving. We plot the predicted and measured footprints in Figure 7. Each graph in Figure 7 has two pairs of footprint curves for comparison.

Figure 7: 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 footprint of the other 68 groups falls in between. 6 PARSEC benchmarks are shown here. The other two, canmeal and blacksholes, are similar to streamcluster.

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 8-thread execution once and derives the 70 footprints. In fact, it derives the footprint for all thread groups, which is numbered \( 2^8 - 1 = 255 \) for each program. The benefit increases exponentially as we analyze programs with more threads.

The benchmarks are grouped vertically for 3 types of parallelism: pipeline-parallel (ferret and dedup), task-parallel (facesim and bodytrack) and data-parallel (fluidanimate and streamcluster). We discover that these three categories also correspond to different degrees of thread symmetry. The data behavior is most asymmetrical among pipeline threads but most symmetrical among data-parallel threads.
Without compositional analysis, there was no effective way to check for thread symmetry or asymmetry, since it would require testing every thread group. As explained earlier, all group testing is flawed because thread behavior may change if run again. All group simulation is too expensive since the number of runs needed grows exponentially to the number of threads.

7 Applications

This section shows two uses made possible by the shared footprint.

7.1 Predicting Miss Ratio

Shared footprint can be converted to cache miss ratio as discussed in Section 5. We apply Formula 3 to predict the cache miss ratio and present the accuracy of prediction in this section. We compare the miss ratios predicted from shared footprint with the results read from hardware counters. The following hardware events are measured [17]:

- MEM\_INST\_RETIRED.LOADs
- MEM\_INST\_RETIRED.STOREs
- OFFCORE\_RESPONSE\_0\_DATA\_IN.L3\_MISS (MSR Encoding: 0x7033)

The measured miss ratio is the ratio of the off-core event count divided by the sum of the memory instruction counts.

The results is shown in Figure 8 for 8-thread executions of 14 benchmarks selected from PARSEC and SPEC OMP. For applications with symmetric threads\(^1\), we present the miss ratio of a single thread. For applications with asymmetric threads, e.g. ferret and dedup, the miss ratio of the thread with the heaviest workload is presented. Other threads have a similar prediction accuracy. For PARSEC benchmarks, we applied the analysis both with and without sampling. Their running configurations are given in Table 3. For SPEC OMP benchmarks, we run only the sampling version on the ref input size (except ilbdc, which we run train size).

As Figure 8 shows, sampling analysis produces fairly accurate predictions in most cases, except for bots-spar with 1 LLC, md with 2 LLCs and fluid-animate with 1 LLC and 2 LLCs. The full-trace analysis is used for just the first 8 programs and gives similarly accurate prediction in 7 programs but not facesim at 2 LLCs. The reason is that shared footprint expresses the average behavior and does not capture phase behavior. Sampling is effective in addressing this problem, because the shared footprint is measured separately for different samples. As a result, it more accurately predicts the miss ratio of facesim at 2 LLCs. Note that facesim in Figure 1 is different because it was tested on 12MB LLCs. Its miss ratio on 1 or 2 8MB LLCs is similar, so is its performance. Sampling analysis predicts facesim fairly accurately. It is sufficient to capture the average footprint for most benchmarks.

\(^1\)All threads execute the same code.
Another possible source of error is in the composition. We found the composition is actually accurate, as we have evaluated in Section 6.4. The phase behavior is the main source of error for full-trace footprint analysis. Interestingly, Xiang et al. did not find it a problem when predicting the miss ratio for SPEC 2006 programs, even though those programs also have phases [35]. The different findings indicate a difference in the type of phase behavior in parallel code than in sequential code. It will be interesting to study what type of phase behavior affects the miss-ratio prediction.

### 7.2 Optimizing Thread-Core Mapping

It is beneficial to co-locate threads that share data. For example, a technique called faithful scheduling separately schedules threads from different applications [23]. Within the same application, there may still be too many threads to run on the same processor. It is a difficult task to find the best way to group threads since the number of possibilities grows exponentially with the number of threads. On the other hand, optimization may significantly improve performance, especially for programs with asymmetric threads like dedup.

Figure 9 shows that grouping dedup’s threads differently can produce up to 60% performance difference. Finding the optimal thread-core mapping is not an easy task because the number of possible mappings grows exponentially with the thread count. Shared footprint analysis can serve as a model to quickly evaluate the cache performance of each mapping. In our model, we label each mapping with a vector of the last level cache miss ratios (one entry for one cache), which is composed from shared footprint analysis, and rank them based on their miss ratios. From the ranking, some mappings can be deemed as “inferior” to others, meaning they have higher miss ratios on all the target caches than some other mapping. We call them “dominated mappings” (the gray dots in Figure 9) and the rest “dominating mappings” (the black dots in Figure 9). We then focus on the dominating mappings and
search for the best one within them. We experiment with this method on dedup with 8 threads on two shared caches. Our experiment shows that 2/3 of the mappings were identified as being dominated. From the remaining 1/3 mappings, exhaustive testing is used to find the optimal one. Compared to exhaustively testing all mappings, this method is 37% faster.

8 Locality in Shared Cache: A Synthesis

Locality theory has two types of definitions: locality of individual accesses and locality of execution windows. The access locality is measured by the reuse distance, and the window locality by footprint. Locality analysis addresses two types of sharing: a parallel mix of sequential programs share the cache but not data, and a multi-threaded program shares both the cache and the data. We review the past studies on either type of sharing targeting either type of locality.

Reuse Distance in Shared Cache without Data Sharing The first models were given by Suh et al. [27] for time-sharing systems and Chandra et al. [7] for multicore. 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 [33]. Reuse distance is expensive to measure, although the cost can be reduced by sampling [24, 36, 3],
OS and hardware support [38, 6, 28] and parallelization [24, 22, 8, 15]. Recent theories use reuse time. CAPS and StatStack are first models to analyze shared cache entirely from reuse time, therefore with a linear-time complexity. In CAPS, the composition is based on distinct data blocks per cycle (DPC) [18]. In StatStack, the composition is done by first composing the private reuse time and then converting it to reuse distance [13].

**Footprint without Data Sharing** Denning and others established the working set theory for memory allocation [9]. Thiebaut and Stone defined footprint as a program’s data in cache [29]. Falsafi and Wood redefined it to mean data blocks accessed by a program, so the data in cache is its “projection” [14]. Early studies used footprint to model interference in time-shared cache [29, 1, 14]. The footprint was measured for a single window length [29, 14] or estimated for all lengths [27, 7], including the working set theory (property P2 in [10]) in 1972 and recently DPC in CAPS [18]. Xiang et al. gave a linear-time algorithm to precisely measure the footprint for all-length windows [34]. Their higher-order theory (HOTL) can convert between footprint and reuse distance [35], so the footprint models are now as efficient as CAPS and StatStack.

Shared footprint solves the more difficult problem than Xiang et al. [34], because it measures not just the footprint but also the number of sharers. It subsumes the previous solution, which is now a sub-case, i.e. $sfp_1$. 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 [2, 12]. 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 [30, 5].

Recent solutions developed the concept of concurrent reuse distance [19, 11, 31, 32], also called multicore reuse distance [24]. 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 [19, 11]. 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 [31]. 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 [32].

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 [14]. Shared footprint in

---

2The naming in StatStack is different. Its reuse distance is our reuse time, and its stack distance is our reuse distance.
this technical report 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 effective cache size and cache sharing spectrum (Section 5). 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 6.4) and thread-group miss ratio (Section 7.1), both are new and impossible with previous techniques.

9 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.

References


