# HOTL: a Higher Order Theory of Locality

Xiaoya Xiang Chen Ding Hao Luo
Department of Computer Science
University of Rochester
{xiang, cding, hluo}@cs.rochester.edu

Bin Bao *
Adobe Systems Incorporated
bbao@adobe.com

## Abstract

The locality metrics are many, for example, miss ratio to test performance, data footprint to manage cache sharing, and reuse distance to analyze and optimize a program. It is unclear how different metrics are related, whether one subsumes another, and what combination may represent locality completely.

This paper first derives a set of formulas to convert between five locality metrics and gives the condition for correctness. The transformation is analogous to differentiation and integration. As a result, these metrics can be assigned an order and organized into a hierarchy.

Using the new theory, the paper then develops two techniques: one measures the locality in real time without special hardware support, and the other predicts multicore cache interference without parallel testing. The paper evaluates them using sequential and parallel programs as well as for a parallel mix of sequential programs.

***Categories and Subject Descriptors*** C.4 [*Performance of Systems*]: Modeling Techniques

***General Terms*** Measurement, Performance, Theory

***Keywords*** Locality metrics, Locality modeling

## 1. Introduction

The memory system of a computer is organized as a hierarchy. Locality metrics are used in software and hardware to manage and optimize the use of the memory hierarchy. For locality analysis, the basic unit of information is a data access, and the basic relation is a data reuse. The theory of locality is concerned with the fundamental properties of data accesses and reuses, just as the graph theory is with nodes and their links.

An influential theory developed over the past four decades is the working-set locality theory (WSLT) [14]. In this paper, we develop a similar theory for cache locality (CLT). Cache locality metrics are many and varied. To quantify performance, we use the miss rate. To manage sharing, we use the footprint. To analyze and optimize a program, we use the reuse distance. Some metrics are hardware dependent, useful for evaluating a specific machine and managing it at run time. Others are hardware independent, useful for optimizing a program for all cache sizes. The two types of metrics are converging in multicore caching, where the total cache size is fixed but the available portion for each program varies.

In this paper we consider five locality metrics, with a short description here and the precise definitions in the next section.

- *Footprint*: the expected amount of data a program accesses in a given length window.
- *Inter-miss time*: the average time between two cache misses in a given size cache.
- *Volume fill time*: the average time for the program to access a given volume of data.
- *Miss ratio*: the fraction of references that cause cache misses.
- *Reuse distance*: for each data access, the amount of data accessed between this and the previous access to the same datum.

To denote them collectively, we insert 'et' between the last two, take the initial letters (except for the fill time from which we take one 'l'), and produce the acronym "Filmer".

We present a theory showing that the five Filmer metrics can be mutually derived from each other. The conversion involves taking the difference in one direction and the sum in the reverse direction. The theoretical relation is analogous to differentiation and integration. Hence we call it a *higher order theory* of locality (HOTL).

Similar conversions have been part of the working set theory, making it the first HOTL theory (Section 2.8). The working set theory was developed to analyze locality in the main memory. The new theory we develop is for cache memory. It endows each of the five cache locality metrics the collective strength of all its Filmer peers:

- *Efficiency*. If we can measure one Filmer metric on-line, we can calculate all the others at the same time.
- *Composability*. The miss rate does not compose in that when a group of programs are run together, the number of misses is not the sum of the misses of each member running alone. If another Filmer metric is composable, then we can compose the miss rate indirectly.
- *Hardware sensitivity*. If we can measure the effect of cache associativity and other hardware parameters on the miss rate, we can compute their impact on the other metrics.

The conversion methods we describe are not always accurate. The correctness depends on whether the footprint statistics in reuse windows is similar to the footprint in general windows, in other words, whether the reuse windows are representative of general windows. We call the condition the *reuse-window hypothesis*. The Filmer metrics capture different aspects of an execution: the reuse distance is per access, the footprint is per window, while the miss-

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



*ASPLOS'13*, March 16–20, 2013, Houston, Texas, USA.

ratio has the characteristics of both. Their conversion creates conflicts, and the reuse-window hypothesis is the condition for reconciliation.

Our recent work shows that one of the Filmer metrics, the average data footprint, can be computed efficiently [46]. In this work, we further improve the efficiency through sampling. More importantly, we apply the HOTL theory to convert it to reuse distance and predict the miss ratio. The purpose of the miss-ratio prediction is twofold: to validate the theory and to show a practical value. The main results are:

- *Real-time locality measurement.* The HOTL-enabled technique predicts the miss ratio for thousands of cache sizes with a negligible overhead. When tested on SPEC 2006 and PARSEC parallel benchmarks, the prediction matches the actual miss ratio measured using the hardware counters. Without sampling, the analysis is 39% faster than simulating a single cache size. With sampling, the end-to-end slowdown is less than 0.5% on average with only three programs over 1%.
- *Cache interference prediction.* The HOTL-enabled technique predicts the effect of cache sharing without parallel testing. For pair interference, the result can be characterized as half-and-half (Section 4.5).

Knowing the miss rate does not mean knowing the memory performance. The actual effect of a cache miss depends significantly on data prefetching, memory-bus arbitration, and other factors either in the CPU above the cache hierarchy or the main memory below. In this paper, we limit our scope to the models of data and cache usage and to methods that measure and reduce the number of cache misses.

## 2. The Higher Order Theory of Cache Locality

The theory includes a series of conversion methods and their correctness condition. We will refer to these methods collectively as the HOTL conversion for the Filmer metrics.

### 2.1 Locality Metrics

The working set theory defines the locality metrics to measure the intrinsic demand of a process [13]. The actual performance is the hardware response to the program demand. By defining locality metrics independent of their specific uses, the approach combines clarity and concision on the one hand and usefulness and flexibility on the other. We follow the same approach and say that a locality metric is *program intrinsic* if it uses only the information from the data access trace of a program. Throughout the paper, we use $n$ to denote the length of the trace and $m$ the total amount of data accessed in the trace.

A footprint is defined on a time window, and the miss ratio for a cache size. Since we do not know a priori in which window or cache the metrics may be used, we define the footprint and miss ratio metrics to include all windows and all cache sizes — they are functions over a parameter range.

The five metrics we consider are program intrinsic functions defined on a sequential data access trace. The time is logical and counted by the number of data accesses from the start of the execution. The cache is fully associative and uses the LRU replacement, with a fixed cache-block size. We will consider the physical time and set associative cache when we apply the basic theory. We use the term miss ratio if the time is logical and miss rate if it is physical.

### 2.2 Average Footprint

A footprint is the amount of data accessed in a time window. A performance tool often measures it for some execution window, i.e. taking a snapshot. A complete measure should consider *all* execution windows. For each length $l$, the average footprint $fp(l)$ is the average footprint size in all windows of length $l$.

Let $W$ be the set of all length-$l$ windows in a length-$n$ trace. Each window $w$ has a footprint $fp_w$. The average footprint $fp(l)$ is the total footprint in these windows divided by $n - l + 1$, the number of the length-$l$ windows.

$$fp(l) = \frac{\sum_{all\ w\ of\ length\ l} fp_w}{n - l + 1}$$

For example, the trace "abbb" has 3 windows of length 2: "ab", "bb", and "bb". The size of the 3 footprints is 2, 1, and 1, so $fp(2) = (2 + 1 + 1)/3 = 4/3$.

The footprint is composable in that the combined footprint of two programs is the sum of their individual footprints (assuming no data sharing). We have used this property when developing efficient models of cache sharing [45, 46]. Another useful property, which we will explore in Section 3, is that the footprint is amenable to sampling.

The working set theory defined the average number of pages accessed in a time window as the working set size and gave a linear-time method to estimate the size [13]. A number of other approximate solutions followed [9, 27, 36, 39]. Our recent work gave two algorithms to measure the footprints in all execution windows and compute either the distribution [45] or the average [46] of the footprints for windows of the same length. The average footprint, e.g. the one in the preceding example, can be computed precisely in linear time. We use the average footprint in this work. Our measurement algorithm [46] will play a critical role in the new theory in Section 2.7.

### 2.3 Volume Fill Time

Intuitively, we may consider the cache as a reservoir and the data access of a program a stream feeding into the reservoir with new content. Having a fixed capacity, the reservoir discharges (evicts) previous volumes as it receives the new flows. The key concept in this analogy is the volume fill time, the time taken for a stream to fill the reservoir.

The volume fill time is the time a program takes to access a given amount of data, or symbolically, $vt(v)$ for volume $v$. The metric is program intrinsic. To model hardware, we simplify and assume that the cache is fully associative LRU. Under the assumption, the volume fill time $vt(c)$ is the time for a program to fill the cache of size $c$. Whether the cache is empty or not, after $vt(c)$, the cache is populated with the data (and only the data) accessed in the last $vt(c)$ time. In the cold-start cache, all data will be brought in by cache misses. In the warm cache, the fraction of the data already in the cache will stay, and the rest will be brought in by cache misses. We call the volume fill time interchangeably as the *cache fill time*.

The fill time can be defined in two different ways. First, we define it as the inverse of the footprint function:

$$vt(c) = \begin{cases} fp^{-1}(c) & \text{if } 0 \le c \le m \\ \infty & \text{if } c > m \end{cases}$$

where $m$ is the total amount of program data. Within the range $0 \le c \le m$, the invariant $fp(vt(c)) = fp(fp^{-1}(c)) = c$ symbolizes the conversion that when the footprint is the cache size, the footprint window is the fill time. The conversion is shown visually in Figure 1. From the average footprint curve, we find the cache size $c$ on the y-axis and draw a level line to the right. At the point the line meets the curve, the $x$-axis value is the fill time $vt(c)$.

A careful reader may question the uniqueness of the fill time. For example for the trace "xx...x", it is unclear what should be the fill time $vt(1)$. When defined as the inverse function $fp^{-1}$,

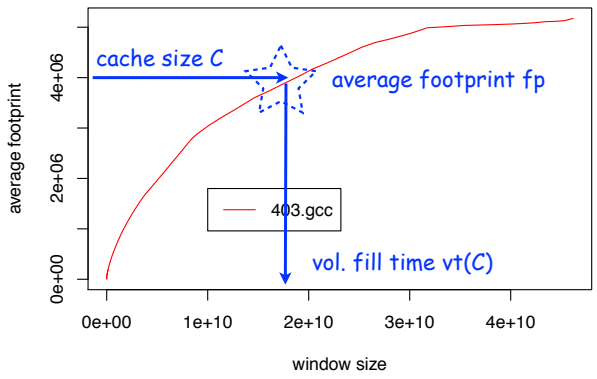



Figure 1: Defining the volume fill time using the footprint.

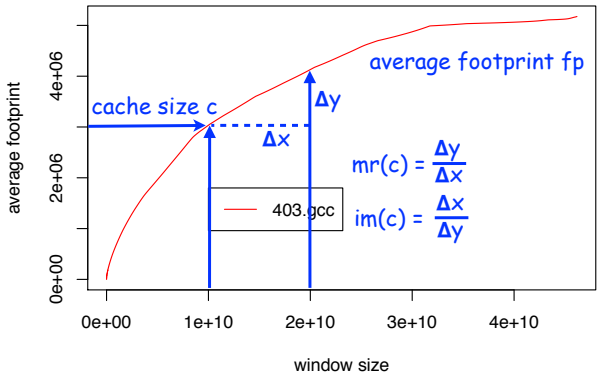



Figure 2: Equivalent conversions of the footprint to the miss ratio and the fill time to the inter-miss time.

the same problem happens if there are $x_1, x_2$ such that $fp(x_1) = fp(x_2)$. However, this problem does not occur using the footprint-based definition. We will prove later in Section 2.7 that the average footprint is a concave function. As a result, it is strictly increasing, and as its inverse, $vt$ is a proper function and strictly increasing as well. We call the footprint-based definition the *Filmer fill time*.

Alternatively, we can define the fill time in a different way. For the volume $v$, we find all windows in which the program accesses $v$ amount of data. The average window length is then the fill time. We refer to the second definition the *direct fill time*, since it is defined directly, not through function inversion.

Consider another example trace "abbc". The Filmer fill time is $vt_{Filmer}(1) = 1$, since all single-element windows access one datum. The direct fill time takes the 5 windows with the unit-size data access: "a", "b","b", "bb", and "c" and computes the average $vt_{direct}(1) = (1+1+1+2+1)/5 = 6/5$. The Filmer definition uses the windows of the same length. The direct definition uses the windows of possibly different lengths.

The cache fill time is related to the residence time in the working set theory [14]. Once a program accesses in a data block but stops using it afterwards, its residence time in cache is the time it stays in cache before being evicted.

In Appendix A, we give an algorithm to measure the direct fill time. In Section 4.4, we show that the direct definition has serious flaws and is unusable in practice. Unless explicitly specified in the rest of the paper, by fill time we mean the Filmer fill time.

### 2.4 Inter-miss Time and Miss Ratio

We derive the inter-miss time for fully associative LRU cache of size $c$. Starting at a random spot in an execution, run for time $vt(c)$, the program accesses $c$ amount of data and populates the cache of size $c$. It continues to run and use the data in the cache until the time $vt(c+1)$, when a new data block is accessed, triggering a capacity or a compulsory miss [24]. The time interval, $vt(c+1) - vt(c)$, is the miss-free period when the program uses only the data in cache. We use this interval as the average inter-miss time $im(c)$[1]. The reciprocal of $im(c)$ is the miss ratio $mr(c)$.

$$im(c) = \begin{cases} vt(c+1) - vt(c) & \text{if } 0 \le c < m \\ \frac{n}{m} & \text{if } c \ge m \end{cases}$$

Since the fill time is the inverse function of the footprint, we can compute the miss ratio from the footprint directly. The direct conversion is simpler and more efficient. In practice, we measure the footprint not for all window sizes but only those in a logarithmic series. Let $x$ and $x + \Delta x$ be two consecutive window sizes we measure, we then compute the miss ratio for cache size $c = fp(x)$:

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

Being a simpler and more general formula, we will use it in the theoretical analysis and empirical evaluation. To cover all cache sizes in practice, we use it as the miss ratio for all cache sizes $c \in [fp(x), fp(x + \Delta x))$.

The fill time ($vt$) conversion and the footprint ($fp$) conversion are equivalent. Figure 2 shows the two visually. For the same two data points on the footprint curve, let $\Delta x = x_2 - x_1$ be the difference in the window length and $\Delta y = y_2 - y_1$ be the difference in the amount of data access. The fill time conversion computes the inter-miss time $im(y_1) = \frac{vt(y_2) - vt(y_1)}{y_2 - y_1} = \frac{\Delta x}{\Delta y}$, and the footprint conversion computes the miss ratio $mr(fp(x_1)) = mr(y_1) = \frac{fp(x_2) - fp(x_1)}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$.

For associative cache, Smith showed that cache conflicts can be estimated based on the reuse distance [37]. Hill and Smith evaluated how closely such estimate matched with the result of cache simulation [25]. We next derive the reuse distance. Once derived, we can use it and the Smith formula to estimate the effect of cache conflicts and refine the miss ratio prediction.

### 2.5 Reuse Distance

For each memory access, the *reuse distance*, or *LRU stack distance*, is the number of distinct data used between this and the previous access to the same datum [31]. The reuse distance includes the datum itself, so it is at least 1. The probability function $P(rd = c)$ gives the fraction of data accesses that have the reuse distance $c$. The capacity miss ratio, $mr(c)$, is the total fraction of reuse distances greater than the cache size $c$, i.e. $mr(c) = P(rd > c)$. Consequently,

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

The reuse distance has extensive uses in program analysis and locality optimization. Any transformation that shortens a long reuse distance reduces the chance of a cache miss. At the program level, reuse distance analysis extends dependence analysis, which identifies reuses of program data [1], to count the volume of the intervening data [4, 8, 10]. At the trace level, the analysis can correlate the change in locality in different runs to derive program-level patterns and complement static analysis [21, 30, 49].

[1] In the working-set theory, the corresponding metric is the time between page faults and known as the lifetime.

To review the conversion formulas, let's consider the example trace "xyzxyz...". Assuming it infinitely repeating, we have $m = 3$ and $n = \infty$. The following table shows the discrete values of the Filmer metrics computed according to the HOTL conversion.

| $t$ | $fp(t)$ | $c$ | $vt(c)$ | $im(c)$ | $mr(c)$ | P(rd=c) |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 2 | 2 | 2 | 1 | 1 | 0 |
| 3 | 3 | 3 | 3 | $\infty$ | 0 | 1 |
| 4 | 3 | 4 | $\infty$ | $\infty$ | 0 | 0 |

### 2.6 The Higher Order Relations

In algebra, the term order may refer to the degree of a polynomial. Through differentiation, a higher order function can derive a lower order function. If we use the concept liberally on locality functions (over the discrete integer domain), we see a higher order locality theory, as shown in a metrics hierarchy in Figure 3.

| locality metrics | formal property | useful characteristics |
|---|---|---|
| 3rd order: footprint, volume fill time | concave/ convex | linear-time, amenable to sampling, composable (dynamic locality) |
| 2nd order: miss ratio, inter-miss time | monotone | machine specific, e.g. cache size/associativity (cache locality) |
| 1st order: reuse distance | non-negative | decomposable by code units and data structures (program locality) |

Figure 3: The hierarchy of cache locality metrics. The five locality metrics are mutually derivable by either taking the difference of the metrics when moving down the hierarchy or taking the sum of the metrics when moving up.

In the preceding sections, we have shown the series of conversions from the third order metric, the footprint, to the first order metric, the reuse distance. To compute a lower order metric, the HOTL conversion takes the difference of the function of a higher order metric. The inter-miss time is the difference of the fill times, and the reuse distance is the difference of the miss ratios.

The conversion formulas are all reversible. We can calculate a higher order metric by integrating the function of a lower order metric. For example, the miss ratio is the sum of the reuse distances greater than the cache size. The fill time is the sum of the inter-miss times up to the cache size.

The mathematical property is different depending on the order of the locality metric, as shown in the second column in Figure 3. Going bottom up, the reuse distance is a distribution, so the range is non-negative. For just compulsory and capacity misses, the miss ratio is monotone and non-increasing, i.e. the stack property [31]. The footprint has been shown to be monotone [46]. Later we will prove a stronger property.

Although the phrase higher order was not used, the working set theory was about the higher order relations between the working set size, the miss rate, and the reuse-time interval. In Section 2.8, we will compare the two higher order theories.

### 2.7 The Correctness Condition

The conversion from the footprint to the miss ratio is not always correct. To understand correctness, consider the reuse distance and the footprint both as window statistics. The reuse distance is the footprint of a reuse window. A reuse window starts and finishes with two accesses to the same datum with no intervening reuses. For a program with $n$ accesses to $m$ data, there are $n - m$ finite-length reuse windows. They are a subset of all windows. The number of all windows is $n$ choose 2 or $\frac{n(n+1)}{2}$. We define the average footprint over all reuse windows as $rfp(l)$, the same way we define $fp(l)$ over all windows.

In this section, we show the correctness condition: for the HOTL conversions to be correct, the two functions, $fp(l)$ and $rfp(l)$, must be equal.

To show this result, we introduce a different formula for predicting the miss ratio. To estimate whether an access is a miss for cache size $c$, we take the reuse window length $l$, find the average footprint $fp(l)$, and predict it a cache miss if and only if $fp(l) > c$. We call this method the *reuse-time conversion*. Let $P(rt = t)$ be the density function of the reuse time, that is, the fraction of reuse windows with the length $t$. The miss ratio predicted by the reuse-time conversion is as follows. We label the result $mr_{rt}$ to indicate that the prediction is based on the reuse time. The first access to a datum has the reuse time of $\infty$.

$$mr_{rt}(fp(l)) = P(rt > l) = \sum_{t=l+1}^{\infty} P(rt = t)$$

If we re-label $fp(l)$ as the working set size, the formula is identical to that of Denning and Schwartz (Section 2.8). However, the use of $fp(l)$ is an important difference. The reuse-time conversion is a modified version of Denning and Schwartz. We may call it an augmented Denning-Schwartz conversion.

Take the example trace "xxyxxz". Two of the average footprints are $fp(3) = 2$ and $fp(4) = \frac{7}{3}$. The reuse times, i.e. the length of the reuse windows, are $\infty, 2, \infty, 3, 2, \infty$. The reuse-time conversion is $mr_{rt}(2) = mr_{rt}(fp(3)) = \sum_{t=4}^{\infty} P(rt = t) = 50\%$. The Filmer conversion is based on the footprint. We call it $mr_{fp}$ and have $mr_{fp}(2) = fp(4) - fp(3) = 33\%$. In general for small traces, the reuse-time conversion is more accurate, as is the case in this example.

Next we prove that for large traces, the miss ratio prediction is the same whether using the reuse time or using the footprint. Then we will show the correctness condition of the entire HOTL theory as a corollary.

From the view of the locality-metrics hierarchy, the reuse-time conversion is bottom up from a first-order metric to a second-order metric. The footprint conversion is top-down from a third-order metric to the same second-order metric. If they meet and produce the same result, we have the equivalence relation across the entire hierarchy.

To prove the equivalence, we need the recently published formula that computes the average footprint from the reuse-time distribution [46].

**Lemma 2.1** (Xiang formula [46]).

$$\begin{aligned} fp(w) \quad &= m - \frac{1}{n-w+1}\Big(\sum_{i=1}^{m}(f_i - w)I(f_i > w) \\ &+ \sum_{i=1}^{m}(l_i - w)I(l_i > w) \\ &+ n\sum_{t=w+1}^{n-1}(t-w)P(rt=t)\Big) \end{aligned} \tag{1}$$

The symbols are defined as:

- $f_i$: the first access time of the $i$-th datum.

- $l_i$: the *reverse* last access time of the $i$-th datum. If the last access is at position $x$, $l_i = n + 1 - x$, that is, the first access time in the reverse trace.
- $P(rt = t)$: the fraction of accesses with a reuse time $t$.
- $I(p)$: the predicate function equals to 1 if $p$ is true; otherwise 0.

If we assume $n \gg w$, the equation can be simplified to

$$fp(w) \approx m - \sum_{t=w+1}^{n-1} (t - w)P(rt = t)$$

**Theorem 2.2** (Footprint and reuse-time conversion equivalence). *For long executions ($n \gg w$), the footprint conversion and the reuse-time conversion produce equivalent miss-ratio predictions.*

**Proof** Let the cache size be $c$ and $l$ and $l + x$ be two consecutive window sizes such that $c \in [fp(l), fp(l + x))$. The miss ratio by the footprint conversion is $\frac{fp(l+x)-fp(l)}{x}$.

Expand the numerator $fp(l+x)-fp(l)$ using the approximate equation from Lemma 2.1:

$$\begin{aligned}
&fp(l + x) - fp(l)\\
\approx& m - \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t) - m + \sum_{t=l+1}^{n-1} (t - l)P(rt = t)\\
=& \sum_{t=l+1}^{n-1} (t - l)P(rt = t) - \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t)\\
=& \sum_{t=l+1}^{l+x} (t - l)P(rt = t) + \sum_{t=l+x+1}^{n-1} (t - l)P(rt = t)\\
&- \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t)\\
=& \sum_{t=l+1}^{l+x} (t - l)P(rt = t) + x \sum_{t=l+x+1}^{n-1} P(rt = t)\\
\approx& \sum_{t=l+1}^{l+x} xP(rt = t) + x \sum_{t=l+x+1}^{n-1} P(rt = t)\\
=& x \sum_{t=l+1}^{n-1} P(rt = t)\\
\approx& x \sum_{t=l+1}^{\infty} P(rt = t)
\end{aligned}$$

The miss ratio, $\frac{fp(l+x)-fp(l)}{x}$, is approximately $\sum_{t=l+1}^{\infty} P(rt = t)$, which is the result of the reuse-time conversion. Note that the equation is approximately true also because of the earlier simplifications made to the Xiang formula. ∎

The two predictions being the same does not mean that they are correct. They may be both wrong. Since the correct calculation can be done using reuse distance, the correctness would follow if from the reuse time, we can produce reuse distance. In other words, the correctness depends on whether the all-window footprint used by the reuse time conversion is indeed the reuse distance. We can phrase the correctness condition as follows:

COROLLARY 2.3 (Correctness). *The footprint-based conversions are accurate if the footprints in all reuse windows have the same distribution as the footprints in all windows, for every reuse window length $l$.*

When the two are equal, using the all-window footprint is the same as using the reuse distance. We posit as a hypothesis that the condition holds in practice, so the HOTL conversion is accurate. We call it the *reuse-window hypothesis*.

Consider the following two traces. The second trace has a smaller difference between the all-window footprint $fp$ and the reuse-window footprint $rfp$. The smaller difference leads to more accurate miss ratio prediction by HOTL. The hypothesis does not hold in either trace, so the prediction is not completely accurate. As to real applications, we will show an empirical evaluation for the full suite of SPEC CPU2006 benchmark programs [23] and a number of PARSEC parallel programs [6].

| trace | $fp(2)$ | $rfp(2)$ | mr(1) | | error |
|---|---|---|---|---|---|
| | | | $pred$ | $real$ | $\lvert pred - real \rvert$ |
| wwwx | 4/3 | 1 | 1/3 | 2/4 | 17% |
| wwwwx | 5/4 | 1 | 1/4 | 2/5 | 5% |

Finally, we show another consequence of Theorem 2.2.

COROLLARY 2.4 (Concavity). *The average footprint $fp(x)$ is a concave function.*

Since $\frac{fp(l+x)-fp(l)}{x} \approx \sum_{t=l+1}^{\infty} P(rt = t)$, $fp(l)$ always increases but increases at a slower rate for a larger $l$. The function is obviously concave. In the higher order relation, the concavity guarantees that the miss ratio predicted by HOTL is non-increasing with the cache size (as expected from the inclusion property [31]).

### 2.8 Comparison with Working Set Theory

The first higher-order locality theory is the working set theory, pioneered in Peter Denning's thesis work [13]. His 1968 paper established the relation between the working set size, the miss rate, and the inter-reference interval (iri). The last one is the same as reuse time. The notion of reuse distance or the LRU stack distance was not formalized until 1970 [31]. Figure 4 shows the parallels between the working set locality theory (WSLT) and the new cache locality theory of this paper (CLT).

| HOTL hierarchy | working set locality theory (WSLT) | cache locality theory (CLT) |
|---|---|---|
| data volume (3rd order) | mean WS size s(T) | mean footprint fp(T), mean fill time vt(c) |
| miss rate (2nd order) | time-window miss rate m(T), lifetime L(T)=1/m(T) | LRU miss rate mr(c), inter-miss time im(c)=1/mr(c) |
| reference behavior (1st order) | inter-reference interval (reuse time) distribution P(iri=x) | reuse distance distribution P(rd=x) |

Figure 4: Comparison between two higher order locality theories: the working set locality theory (WSLT) for dynamic partitioned primary memory and the cache locality theory (CLT) for cache memory.

WSLT computes the metrics bottom-up. The base metric, $P(iri = x)$, is the histogram of the inter-reference intervals (reuse time), measured in linear time in a single pass of the address trace. The time-window miss ratio $m(T)$ is the sum of reuse time. The mean working set size $s(T)$ is the sum of $m(T)$.

$$m(T) = P(rt > T)$$
$$s(T+1) = s(T) + m(T)$$

Taking together, the working set size $s(T)$ is the second order sum of the reuse frequency.

The $s(T)$ formula was first proved by Denning and Schwartz in 1972 [15]. The formulation assumes an infinitely long execution with a "stationary" state ("the stochastic mechanism ... is stationary"). The working set, $w(t, T)$, is the number of distinct pages accessed between time $t - T + 1$ and $t$. The average working set size, $s(T)$, is the limit value when taking the average of $w(t, T)$ for all $t$. The proof is based on the fact that only recurrent pages with an infinite number of accesses contribute to the mean working set size.

In 1978, Denning and Slutz defined the generalized working set (GWS) as a time-space product [16]. The product, denoted here as $st(T)$, is defined for finite-length execution traces, variable-size memory segments, all cache replacement policies that observe the stack property. Interestingly, they found the same recursive relation. The GWS formula is as follows, where the last term is the extra correction to take into account the finite trace length.

$$st(T+1) = st(T) + Tm(T) - a(T)$$

Dividing both sides by $T$, we have the last term vanishing for large $T$ and see the same recursive relation for GWS in finite-length traces as $s(T)$ in infinitely long traces.

In the present paper, the same recurrence emerges in Section 2.7 as an outcome of Theorem 2.2. For the average footprint, we have effectively

$$fp(T+1) = fp(T) + m(T)$$

If we view the following three as different definitions of the working set: the limit value in 1972 [15], the time-space product in 1978 [16], and the average footprint in 2011 [46], we see an identical equation which Denning envisioned more than four decades ago (before the first proof in 1972). We state it as a law of locality and name it after its inventor:

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

As the relativity theory gives the relation between space and time, Denning's law gives the relation between memory and computation: the working set is the working memory, and the reuse frequency is a summary of program actions (time transformed into frequency and a spectrogram of time). The law states that the relation is higher order.

Our work augments Denning's law in two ways. First, it is the final step to conclusively prove Denning's Law — that it holds for the footprint working set in finite-length program executions. The 1972 proof depends on the idealized condition in infinite-length executions. Subsequent research has shown that the working set theory is accurate and effective in managing physical memory for real applications [14]. The new proof subsumes the infinitely long case and makes Denning's law a logical conclusion for all (long enough) executions. It gives a theoretical explanation to the long observed effectiveness of the working set theory in practice.

Second, we extend HOTL to include cache memory. For main memory, the locality is parameterized in time: the working set of a program in a time quantum. For cache, the primary constraint is space: the miss ratio for a given cache size. Denning et al. named them the "time-window miss ratio" and the "LRU miss ratio" and noted that the two are not necessarily equal [15, 16]. The following formulas show the two miss ratios:

| | |
|---|---|
| working set | $m(T) = P(rt > T)$ |
| cache locality | $mr(fp(T)) = P(rt > T)$ |

In the above juxtaposition, the only difference is the parameter to the miss rate function. In $m(T)$, the parameter is the time window length. In $mr(fp(T))$, the parameter is the cache size. Through the second formula, this work connects the cache size and the reuse frequency. In Section 2.4, we show how the time-centric and the space-centric views have different derivations but the same miss ratio. Then in Section 2.7, we give the reuse-window hypothesis as the condition for correctness, which implies the equality between the time-window miss ratio and the LRU miss ratio.

## 3. Sampling-based Locality Analysis

The footprint can be analyzed through sampling, e.g. by tracing a window of program execution periodically. Sampling has two benefits. First, by reducing the sampling frequency, the cost can be arbitrarily reduced. Second, sampling may better track a program that has significant phase behavior.

***Uniform sampling*** We implement footprint sampling using a technique pioneered by shadow profiling [32] and SuperPin [42]. When a program starts, we set the system timer to interrupt at some preset interval. The interrupt handler is shown in Figure 5. It forks a sampling task and attaches the binary rewriting tool Pin [29]. The Pin tool instruments the sampling process to collect its data access trace, measures all-window footprints using the Xiang formula [46]. In the meanwhile, the base program runs normally until the next interrupt.

**Require:** This handler is called whenever a program receives the timer interrupt
1: $pid \leftarrow fork()$
2: **if** $pid = 0$ **then**
3: Attach the Pin tool and begin sampling until seeing $c$ distinct memory accesses
4: Exit
5: **else**
6: Reset the timer to interrupt in $k$ seconds
7: Return
8: **end if**

Figure 5: The timer-interrupt handler for footprint sampling

***Footprint Sampling*** Footprint by definition is amenable to sampling. We can start a sample at any point in an execution and continue until the sample execution accesses enough data to fill the largest cache size of interest. We can sample multiple windows independently, which means they can be parallelized. It does not matter whether the sample windows are disjoint or overlapping, as long as the choice of samples is random and unbiased.

***The Associative Cache*** A program execution produces a series of $m$ samples at regular intervals, $x_1, x_2, \ldots, x_m$. We use them in the following way:

1. For each sample $x_i$, with trace length $n_i$, predict the miss ratio function $mr(x_i, c)$ for each cache size $c$ by the following:
   (a) Use the analysis of Xiang et al. [46] to compute the average footprint function $fp$.
   (b) Use the footprint conversion to compute the capacity miss ratio for cache size $c$.

(c) Use the miss-ratio conversion to compute the reuse distance distribution and the Smith formula [37] to estimate the number of conflict misses for cache size $c$.

2. For all $x_i$, take the weighted average and compute the miss ratio for all cache sizes for the program $mr(c) = \frac{\sum_{i=1}^{m} mr(x_i,c)*n_i}{\sum_{i=1}^{m} n_i}$.

***The Phase Effect*** The preceding design assumes phase behavior. Since different samples may come from different phases, combining their footprints would lose the phase distinction. To validate the conjecture, we will compare the phase-sensitive sampling with phase-insensitive sampling. The former, as just described, computes the miss ratio for each sample and then takes the average. The next design combines the footprint from all the samples and then computes the miss ratio. Specifically, the second design is as follows:

1. For each sample $x_i$, with trace length $n_i$,
   - Use the analysis of Xiang et al. [46] to compute the average footprint function $fp$.
2. For all samples $x_i$, take the weighted average and compute the $fp$ function for the program $fp = \frac{\sum_{i=1}^{m} fp(x_i)*n_i}{\sum_{i=1}^{m} n_i}$.
3. Use the footprint and miss-ratio conversions and the Smith formula [37] to estimate the number of cache misses.

***Comparison with Reuse Distance Sampling*** To be statistically sound, reuse distance sampling must evenly sample reuse windows. After picking an access, it needs to trace the subsequent program accesses until the next data reuse. When a reuse window is long, it does not know a priori how long to monitor, so it has to keep analyzing until seeing the next reuse or until the reuse distance exceeds the largest cache size of interest. The cut-off strategy is also used in footprint sampling.

Beneath this similarity lies two important differences. The reuse distance measures the locality by examining reuses. The footprint measures the locality by examining data accesses. Footprint sampling computes the distribution of all reuse distances from a single sample window using the HOTL conversion. The footprint analysis and conversion take linear time. In comparison, each reuse window sample produces just one reuse distance. It takes asymptotically higher time cost to measure the reuse distance in the sample (than it takes HOTL conversion to compute all reuse distances from the same sample). Hence the advantage of footprint sampling is algorithmic and computational, and this strength comes from the HOTL theory.

# 4. Evaluation

## 4.1 Experimental Setup

We have tested the full set of 29 benchmarks from SPEC 2006 and 8 from the PARSEC v2.1 suite. All programs are instrumented by Pin [29] and profiled on a Linux cluster where each node has two Intel Xeon 3.2GHz processors. PARSEC is run on a machine with two Intel Xeon E5649 processors. In simulation, we simulate a single-level cache, which is shared in the case of parallel code. On a real machine, the baseline is the program run time without instrumentation or any analysis.

For SPEC 2006, we use the first reference input provided by the benchmark suite. Table 1 shows for each SPEC 2006 program the length of trace $n$, the size of data $m$ and the time of the unmodified program execution. The length of SPEC 2006 traces ranges from 20 billion in *403.gcc* to 2.1 trillion in *436.cactusADM*. The amount of data ranges from 3MB in *416.gamess* to 1.7GB in *429.mcf*. For PARSEC, we test programs using the three provided input sizes: simsmall, simmedium and simlarge. We run each with 4 threads, a commonly used configuration.

| $benchname$ | $n$ $(10^{11})$ | $m$ $(10^7 bytes)$ | $T$ $(sec)$ |
|---|---|---|---|
| 400.perlbench | 4.2 | 24.4 | 457 |
| 401.bzip2 | 1.7 | 39.3 | 263 |
| 403.gcc | 0.2 | 40.4 | 72 |
| 410.bwaves | 14.4 | 98.2 | 1664 |
| 416.gamess | 4.8 | 0.3 | 444 |
| 429.mcf | 1.2 | 175.7 | 1172 |
| 433.milc | 3.8 | 74.2 | 1077 |
| 434.zeusmp | 5.7 | 51.9 | 1555 |
| 435.gromacs | 9.8 | 1.4 | 1272 |
| 436.cactusADM | 20.6 | 65.5 | 3411 |
| 437.leslie3d | 6.8 | 12.9 | 1212 |
| 444.namd | 6.8 | 4.7 | 915 |
| 445.gobmk | 0.9 | 2.7 | 173 |
| 447.dealII | 7.3 | 88.5 | 773 |
| 450.soplex | 1.0 | 16.2 | 604 |
| 453.povray | 4.8 | 0.3 | 493 |
| 454.calculix | 9.5 | 16.4 | 1512 |
| 456.hmmer | 4.9 | 4.2 | 303 |
| 458.sjeng | 7.0 | 18.2 | 1356 |
| 459.GemsFDTD | 8.6 | 86.9 | 1397 |
| 462.libquantum | 3.0 | 16.8 | 1391 |
| 464.h264ref | 2.6 | 2.7 | 143 |
| 465.tonto | 10.0 | 5.2 | 1312 |
| 470.lbm | 3.3 | 42.9 | 1491 |
| 471.omnetpp | 2.3 | 17.6 | 1048 |
| 473.astar | 1.4 | 29.5 | 512 |
| 481.wrf | 9.7 | 76.8 | 1895 |
| 482.sphinx3 | 8.9 | 5.1 | 1765 |
| 483.xalancbmk | 3.6 | 43.8 | 778 |

Table 1: The SPEC2006 integer and floating-point benchmarks. For each benchmark, $n$ is the memory trace length of whole execution, $m$ is the number of distinct data blocks (size in bytes) accessed during the execution, and $T$ is the execution time without any instrumentation or analysis.

Locality sampling is implemented using fork, as described in Section 3. The implementation does not yet recognize system calls, so sampling handles only 22 of the 29 sequential programs. Nor does the sampling implementation handle multi-threaded code. We evaluate miss-ratio prediction using the full trace of the 8 parallel programs.

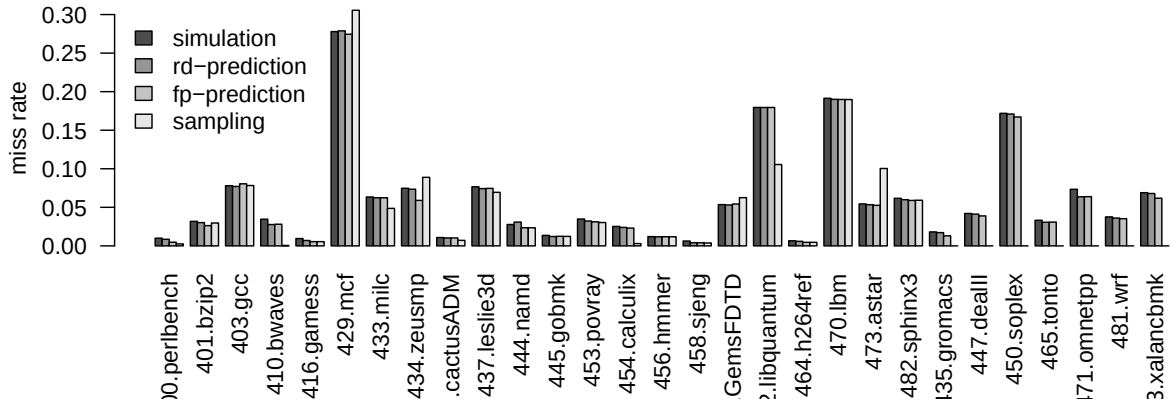
***3073 Cache Sizes*** In the analysis, the footprint and reuse distance numbers are bin-ed using logarithmic ranges as follows. For each (large enough) power-of-two range, we sub-divide it into (up to) 256 equal-size increments. As a result, we can predict the miss ratio not just for power-of-two cache sizes, but 3073 cache sizes between 16KB and 64MB.

## 4.2 Miss-Ratio Prediction

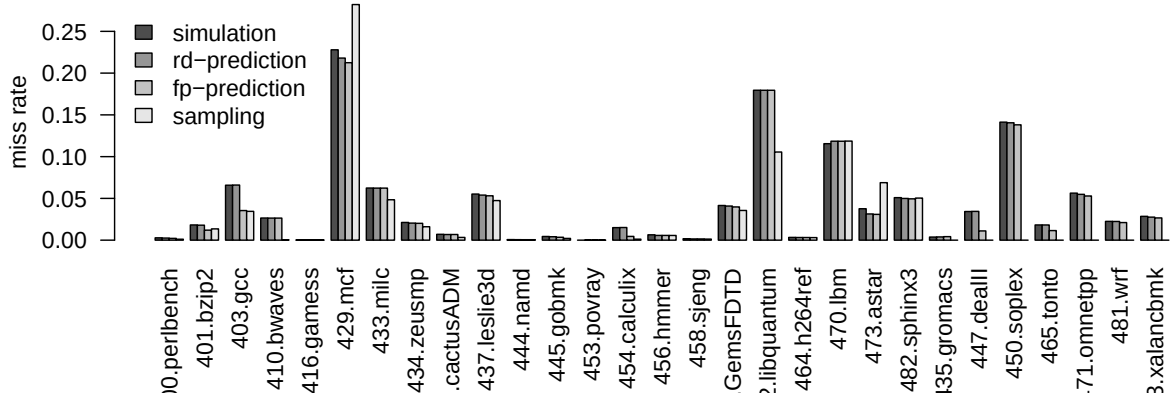
We first evaluate the accuracy and the speed of miss-ratio prediction, made by the Filmer conversion and locality sampling, tested on sequential and parallel programs, and verified through simulation and hardware counters.

### 4.2.1 Sequential Programs

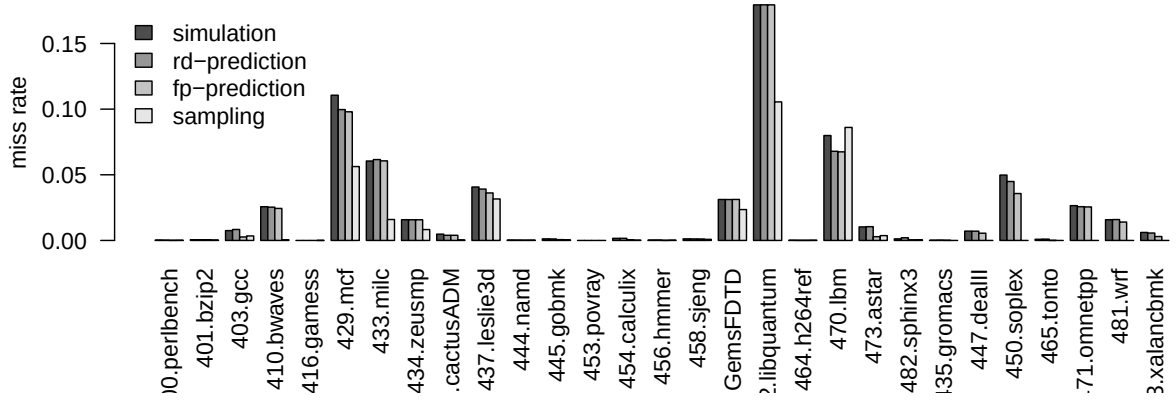
We first use cache simulation to evaluate the accuracy of Filmer-based miss ratio prediction. Instead of evaluating each of the 29




(a) 8-way, 32KB cache




(b) 8-way, 256KB cache




(c) 16-way, 8MB cache

Figure 6: Accuracy of the miss-ratio prediction by reuse distance, footprint (HOTL conversion in Section 2.4), and footprint sampling (Section 3) for 29 SPEC 2006 benchmarks, each on 3 (of the 3073) cache configurations, compared with cache simulation. (a) 8-way, 32KB cache. (b) 8-way, 256KB cache. (c) 16-way, 8MB cache. The sampling results are for 22 out of 29 programs (not the last 7). The average time cost of sampling is 0.5% (Table 2).

programs on 3073 cache sizes, we show results for 3 configurations: 32KB, 8-way associative L1D; 256KB, 8-way associative L2; and 8MB, 16-way associative L3. They are the cache configurations on our test machine. We will compare our prediction to the performance counter result later. The cache-block size is 64 bytes in all cases. The accuracy for the other 3070 cache sizes looks similar.

Figure 6 plots the measured and predicted miss ratios. The *rd-prediction*, which uses the measured reuse distance, is the most accurate. The other two, *fp-prediction* and *sampling* measure the footprint and then use the HOTL conversion in Section 2.4. The HOTL conversion *fp-prediction* closely matches the reuse-distance analysis *rd-prediction* in almost all cases, showing that the footprint-converted reuse distance is almost identical to the measured reuse distance—hence the validity of the reuse-window hypothesis.

***The Phase Effect*** The reuse distances of a program, when added together regardless of phases, predict the (capacity) miss ratio accurately, because an access is a cache capacity miss if and only if its reuse distance is greater than the cache size. On the other hand, the footprint should be affected by phases. As the footprint changes from phase to phase, it is possible that taking the global average might lose critical information.

A consistent result from the theory and the experiments is that the two are largely equivalent, as far as computing the miss ratio is concerned. The theory gives the conversion procedure from the footprint to the reuse distance. The experiments show, by the close match between *rd-prediction* and *fp-prediction* in Figure 6, the conversion is accurate for most programs. This suggests that reuse windows are representative of all windows, and this is why the prediction is accurate in spite of the phase effect.

Another evidence, for which we do not include the results in the paper, is that the two sampling designs in Section 3, phase sensitive and insensitive, produce almost identical predictions.

***Analysis Speed*** Table 2 compares the cost of four analysis methods: the simulation $sim$, the reuse distance $rd$ [49], the footprint $fp$ [46], and the footprint sampling $sp$. For simulation we could use the algorithm of Smith and Hill [25] to simulate all three configurations in one pass. For speed comparison, we ran the simplest simulator once for each configuration. The simulation cost in table 2 is the average of the three runs.

The cost for reuse distance analysis ranges from 52 times to 426 times with an average of 153 times. The footprint analysis costs about 7 times less, with slowdowns between 6 and 66 times and on average 23 times. Simulation for a single configuration has slowdowns from 14 to 80 times, with an average of 38 times.

Comparing the average, we see that measuring the footprint, the third order Filmer metric that can compute the second order metric miss ratio for all cache sizes, is 39% faster than simulating for a single cache size, before we use footprint sampling.

| $benchname$ | $sim$ | $rd$ | $fp$ | $samp$ | $cov$ |
|---|---|---|---|---|---|
| 400.perlbench | 49 | 219 | 34 | 0.24% | 3.1% |
| 401.bzip2 | 34 | 139 | 24 | 0.73% | 1.5% |
| 403.gcc | 24 | 88 | 15 | 0.55% | 0.1% |
| 410.bwaves | 57 | 196 | 35 | 2.14% | 0.5% |
| 416.gamess | 62 | 286 | 40 | 0.29% | 2.9% |
| 429.mcf | 10 | 56 | 6 | 0.14% | 0.03% |
| 433.milc | 21 | 74 | 9 | 1.53% | 0.04% |
| 434.zeusmp | 25 | 102 | 14 | 0.81% | 0.06% |
| 435.gromacs | 40 | 142 | 19 | - | - |
| 436.cactusADM | 40 | 167 | 21 | 0.00% | 1.1% |
| 437.leslie3d | 42 | 131 | 23 | 0.00% | 0.01% |
| 444.namd | 44 | 155 | 24 | 0.00% | 2.2% |
| 445.gobmk | 35 | 130 | 23 | 0.22% | 1.1% |
| 447.dealII | 54 | 209 | 34 | - | - |
| 450.soplex | 13 | 52 | 7 | - | - |
| 453.povray | 51 | 220 | 33 | 0.00% | 1.8% |
| 454.calculix | 39 | 127 | 19 | 0.11% | 1.8% |
| 456.hmmer | 80 | 426 | 59 | 0.00% | 0.8% |
| 458.sjeng | 34 | 152 | 23 | 0.82% | 0.4% |
| 459.GemsFDTD | 42 | 181 | 21 | 1.28% | 0.01% |
| 462.libquantum | 17 | 48 | 9 | 0.00% | 0.01% |
| 464.h264ref | 101 | 424 | 66 | 0.00% | 1.2% |
| 465.tonto | 52 | 168 | 30 | - | - |
| 470.lbm | 14 | 76 | 6 | 0.00% | 0.01% |
| 471.omnetpp | 17 | 69 | 10 | - | - |
| 473.astar | 15 | 73 | 11 | 0.80% | 0.9% |
| 481.wrf | 33 | 113 | 19 | - | - |
| 482.sphinx3 | 30 | 117 | 16 | 0.59% | 1.2% |
| 483.xalancbmk | 31 | 99 | 21 | - | - |
| average | 38 | 153 | 23 | 0.47% | 0.9% |

Table 2: Time comparison between different profiling methods and cache simulation for SPEC 2006. The baseline is the execution time without any instrumentation or analysis. The middle four columns show the slowdown compared to the baseline: $sim$ for simulating one cache size, $rd$ for reuse distance profiling, $fp$ for footprint profiling, and $samp$ for footprint sampling. The last column $cov$ gives the sampling coverage.

#### 4.2.2 Locality Sampling

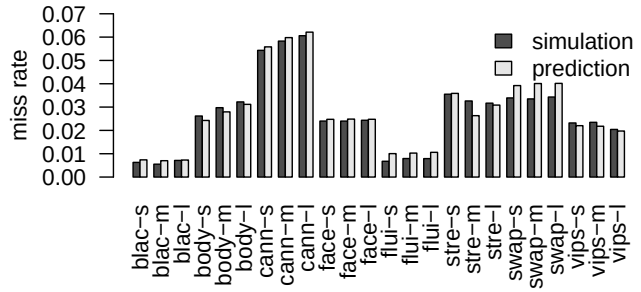
For this experiment, we choose somewhat arbitrarily the frequency of one sample every 10 seconds. The sample length is the volume fill time for the cache size. Sampling analysis is not always accurate. Visible errors are seen in *mcf*, *libquantum* and *astar* in Figure 6. The reason, as shown by the last column of Table 2, is that it covers less than 1% of the execution. The coverage is computed by the ratio of the number of sampled instructions to the total number of instructions (counted by our full trace profiling). The coverage is as low as 0.006% in *lbm*. The low coverage does not mean low accuracy. The prediction of *lbm* is 99% accurate for the 32KB cache, 97% for the 256KB cache, and 92% for the 8MB cache.

In Table 2, we show the slowdown in the end-to-end run time by the column marked $samp$. It ranges from 0% to 2.14%. Three programs have a visible cost of over 1%. They are *bwaves* 2.1%, *GemsFDTD* 1.3% and *milc* 1.5%. The reason for the relatively high cost may be the non-trivial interference between the sampling task and the parent task. Across all programs, the average visible overhead is below a half percent. If we measure the total CPU time, sampling takes between 0% and 80% of the original run time. The average cost is 19%, of which over 18% is hidden by shadow profiling.

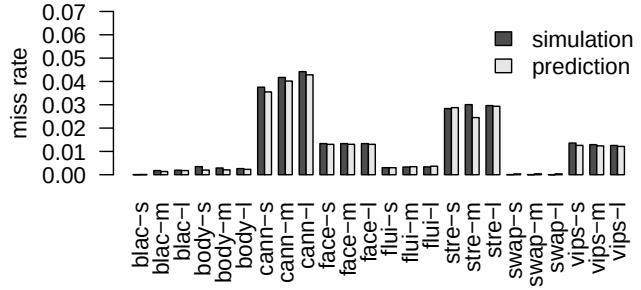
#### 4.2.3 Parallel Programs

Figure 7 shows that for 3 of the 3073 cache configurations and across the 3 input sizes, the predicted miss ratio matches closely with the simulated miss ratio, similar to the results we saw in the sequential programs. The accuracy shows that the reuse-window hypothesis holds for these threaded applications.

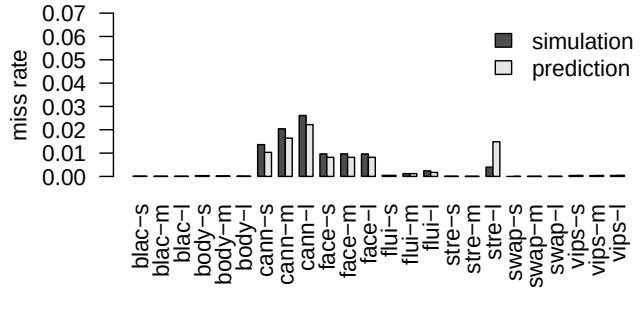
The last column of table 3 shows the slowdowns of footprint profiling, which ranges from 14 times to 159 times with an average of 113 times. We did not profile reuse distance for PARSEC because it took too long. We note that the footprint analysis shows 5 times as much overhead in 4-threaded tests as in sequential programs (159 times in PARSEC vs. 23 times in SPEC 2006). The reason is that our data analysis is still serial, so the overhead is proportional to the total amount of work. We plan to parallelize the




(a) 8-way, 32KB cache




(b) 8-way, 256KB cache




(c) 16-way, 8MB cache

Figure 7: Accuracy of the miss-ratio prediction for 3 (of the 3073) cache configurations and 3 input sizes, compared with cache simulation. (a) 8-way, 32KB cache. (b) 8-way, 256KB cache. (c) 16-way, 8MB cache.

footprint analysis in the future, building on recent work in parallelizing the reuse-distance analysis [12, 22, 33].

### 4.3 Validation on a Real Machine

In Figure 8, we compare the simulation result with the miss ratio measured by the hardware counters on our test machine. To measure the actual misses, we use Intel's VTune tool to record three hardware counter events named

```
OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM
MEM_INST_RETIRED.LOADS
MEM_INST_RETIRED.STORES
```

The measured miss ratio is the first count divided by the sum of the last two counts.

The figure shows a significant difference in *gcc*. The reason is that the simulation considers only data accesses but the hardware counter counts instruction misses in the data cache, which we believe are significant in *gcc*.

| *bench name* | *input size* | $n$ ($10^9$) | $m$ ($10^6 bytes$) | $T$ ($sec$) | slow-down |
|---|---|---|---|---|---|
| black -scholes | S | 0.1 | 0.4 | 0.093 | 129 |
| | M | 0.4 | 1.2 | 0.384 | 91 |
| | L | 1.6 | 4.4 | 1.542 | 88 |
| body -track | S | 0.3 | 8.1 | 0.285 | 129 |
| | M | 1.1 | 11.1 | 0.948 | 155 |
| | L | 4.0 | 14.8 | 3.35 | 111 |
| canneal | S | 0.6 | 43.0 | 1.525 | 19 |
| | M | 1.3 | 84.3 | 3.859 | 15 |
| | L | 2.7 | 164.9 | 8.804 | 14 |
| facesim | S | 12.7 | 344.2 | 7.448 | 139 |
| | M | 12.7 | 344.2 | 7.306 | 131 |
| | L | 12.7 | 344.2 | 7.86 | 116 |
| fluid -animate | S | 0.5 | 10.5 | 0.429 | 114 |
| | M | 1.3 | 20.6 | 0.983 | 145 |
| | L | 3.9 | 57.6 | 2.9 | 124 |
| stream -cluster | S | 0.5 | 1.2 | 0.722 | 87 |
| | M | 2.6 | 2.9 | 1.641 | 138 |
| | L | 9.6 | 9.5 | 6.951 | 173 |
| swapt -ions | S | 3.6 | 0.9 | 2.349 | 134 |
| | M | 1.4 | 1.2 | 0.935 | 114 |
| | L | 5.7 | 1.9 | 3.766 | 132 |
| vips | S | 1.0 | 13.5 | 0.748 | 159 |
| | M | 3.1 | 26.9 | 2.228 | 140 |
| | L | 8.6 | 15.7 | 7.332 | 103 |

Table 3: The PARSEC parallel benchmarks. For each benchmark, $n$ is the memory trace length of whole execution, $m$ is the size of program data (in bytes) accessed during the execution, and $T$ is the execution time without any instrumentation or analysis. The last column is the slowdown of the footprint analysis.

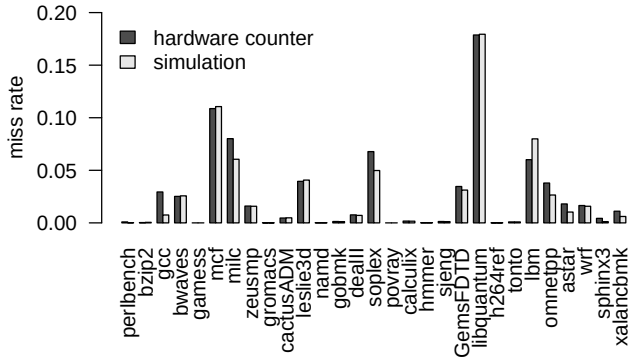



Figure 8: Comparison between hardware counter measured L3 cache miss ratio and the simulation result.

### 4.4 Direct Fill Time vs. Filmer Fill Time

The measurement of the direct fill time, definition in Section 2.3 and algorithm in Section A, takes so long that the only programs we could finish are 10 of the 11 SPEC 2000 integer benchmark programs. Table 4 compares the average time for these programs. An unmodified SPEC 2000 program runs for 3 minutes on average, the direct fill time analysis takes over 22 hours. The average overhead is more than 7 hours for each minute. In comparison, the per minute overhead is an hour and a half for reuse distance and 7 minutes if we first compute footprint and then derive the Filmer fill time.

| analysis | avg. time | avg. slowdown |
|---|---|---|
| direct fill time (Section A) | 22h12m11s | 446x |
| reuse distance | 3h57m36s | 84x |
| Filmer fill time (Section 2.3) | 22m4s | 8x |

Table 4: Speed comparison for 10 SPEC 2000 integer benchmarks. The average trace length $n$ is 47 billion, data size $m$ is 73MB, and baseline run time is 3 minutes and 16 seconds.

More problematic is that with the direct fill time, the predicted miss ratio is not monotone. Worse, the miss ratio may be negative. Consider an example trace with 100 a's followed by 11 b's, 1 c, 20 d's, 15 e's, 1 f and 320 g's. The average time to fill a 4-element cache, $vt(4)$, is 161.5, is longer than the average time to fill a 5-element cache, $vt(5)$, which is 149.5. Since the direct fill time decreases when the cache size $v$ increases, the predicted miss ratio is negative!

The preceding example was constructed based on an analysis of real traces. During experimentation, we found that the miss ratios of some cache sizes were negative. While most of the 3000 or so sizes had positive predictions, the negatives were fairly frequent and happened in most test programs. It seemed contradictory that it could take a program longer to fill a smaller cache. The reason is subtle. To compute the direct fill time, we find windows with the same footprint and take the average length. As we increase the footprint by 1, the length of these windows will increase but the number of such windows may increase more, leading to a lower average, as happened in the preceding example.

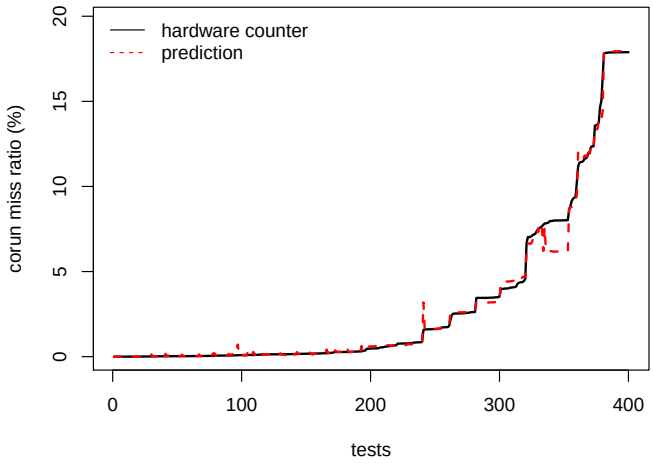
In contrast, the Filmer fill time is a positive, concave function (Corollary 2.4). Its miss-ratio prediction is monotone and can be measured in near real time (Section 4.2.2).

### 4.5 Predicting Cache Interference

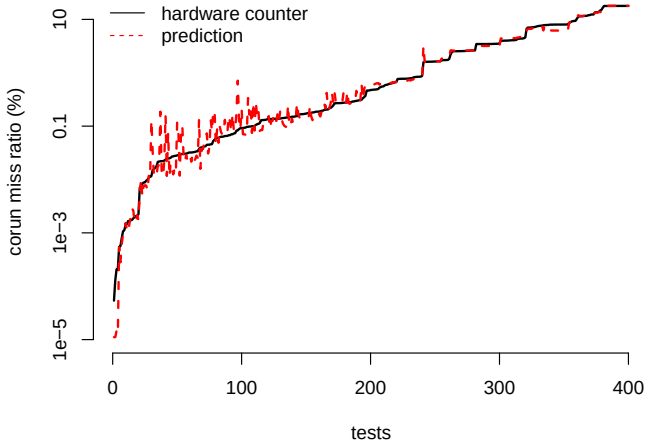
A complete 2-program co-run test for the 29 SPEC 2006 benchmarks would include $\binom{29}{2} = 406$ program pairs. To reduce the clutter in the graphs we show, we choose 20 programs. To avoid bias, we pick programs with the smallest benchmark ids. Since we profile data accesses only, we exclude perlbench and gcc because their large code size may cause significant instruction misses in the data cache. After the removal, we have 20 SPEC benchmark programs from 401.bzip2 to 464.h264ref. The trimming reduces the number of pair-run tests to $\binom{20}{2} = 190$.

Cache interference models were pioneered by Thiebaut and Stone [41], Suh et al. [39] and Chandra et al. [9], who computed the cache interference by the impact of the peer footprint on the self locality.[2] The footprint is measured for a single window length [41] and approximated for multiple lengths [9, 39]. Our subsequent work found a way to measure all-window footprints precisely and efficiently [17, 45, 46]. The self locality is measured by the reuse distance. As the measurement problem for the footprint is solved, the speed of reuse-distance analysis becomes the bottleneck. We found that by profiling up to two days for each program, the reuse distance analyzer by Zhong et al. [49] could finish only 8 SPEC 2006 programs [46]. The total modeling time was over 106 CPU hours, 94% of which was spent on the reuse-distance analysis. In

[2] Chandra et al. also gave a model that used only the reuse distance [9]. Zhuravlev et al. used it and two other such models and found that in task scheduling, they did not significantly outperform a simple model that used only the miss rate [51].




(a) linear scale miss ratios




(b) logarithmic scale miss ratios

Figure 9: The predicted and measured miss ratios of the 380 executions in 190 pair runs. The executions are ordered by the ascending miss ratio as measured by the hardware counters in exhaustive testing. For each execution, the solid (black) line shows the hardware counter result, and the dotted (red) line shows the prediction. The prediction takes about a *half* percent of the time of exhaustive testing. Just two executions have a significant error in both graphs, which are a *half* percent of all executions.

this study, we have measured reuse distance for all benchmarks (see Table 2 for measurement costs). Some programs took over 4 days.

Based on the new theory, we compute the reuse distance from the footprint and predict the co-run interference. Figure 9 compares the measured and predicted miss ratios. There are 190 pair runs for a total of 380 executions. The $x$-axis orders these executions by the measured miss ratios from the lowest to the highest. For easy viewing, we connect the points into a line. The measured curve is necessarily monotone. The prediction is to match the measurement.

Figure 9 has two graphs, showing the miss ratio in the linear scale in the upper graph and the logarithmic scale in the lower graph. The prediction is mostly accurate. The errors happen but for different executions in the two graphs. If an error is visible in the linear scale but not in the logarithmic scale, the error is significant in absolute terms but not in relative terms. Similarly, we have errors significant relatively but not absolutely. The two graphs show just two errors that are significant in both scales. In the other 378 (99.5%) executions, the prediction is either accurate or the error insignificant. From visual inspection, the error is significant in just 0.5% of all executions.

To make the prediction, the analysis needs 1 hour 4 minutes CPU time for sampling, almost as fast as we can run the 20 programs without analysis. In comparison, the exhaustive testing takes over 9 days (estimated) of CPU time. The cost saving is 99.5%.

To see interference in 3-program co-runs, the exhaustive testing has to re-test and collect results anew, but the modeling needs no additional testing. Indeed, the new model has been used in an on-line system to regroup eight programs to run on two quad-core processors (to have a higher performance or at least a more repeatable performance) [47]. The exhaustive testing of the 4-program co-runs in our 20-program suite would need 19 thousand test executions and have taken months of time.

To summarize the pair interference experiment, we can say that the result is half and half: the modeling takes half percent of the time and has a significant error in a half percent of executions.

## 5. Related Work

The concept of locality has evolved from an observation that a program does not use all the data at all times, to quantitative metrics that we can evaluate and compare but for which we must solve the dual problems of speed and precision.

***Locality sampling*** A publicly available system for locality sampling is the SLO tool developed by Beyls and D'Hollander [5]. SLO instruments a program to skip every $k$ accesses and take the next address as a sample. A bounded number of samples are kept in a sample reservoir. To track reuse windows, it checks each access to see if it is an access to some sampled datum. The instrumentation code is carefully engineered in GCC to have just two conditional statements for each memory access (one for address and the other for counter checking). Reservoir sampling reduces the time overhead from 1000-fold slow-down to only a factor of 5 and the space overhead to within 250MB extra memory. The sampling accuracy is 90% with 95% confidence. The accuracy is measured in the reuse time, not the reuse distance or the miss ratio.

To accurately measure reuse distance, a record must be kept to count the number of distinct data appeared in a reuse window. Zhong and Chang developed the bursty reuse distance sampling, which divides a program execution into sampling and hibernation periods [48]. In the sampling period, the counting uses a tree structure and costs $O(\log \log M)$ per access. If a reuse window extends beyond a sampling period into the subsequent hibernation period, the counting uses a hash-table, which reduces the cost to $O(1)$ per access. Multicore reuse distance analysis uses a similar scheme for analyzing multi-threaded code [35]. Its fast mode improves over hibernation by omitting the hash-table access at times when no samples are being tracked. Both methods compute the reuse distance accurately.

StatCache is based on unbiased uniform sampling [3]. After a data sample is selected, StatCache puts the page under the OS protection to capture the next access to the same datum. It uses the hardware counters to measure the time distance till the reuse. OS protection is limited by the page granularity. Two other systems, developed by Cascaval et al. [7] and Tam et al. [40], used the special support on IBM processors to trap accesses to specified data addresses. To reduce the cost, these methods used a small number of samples. Cascaval et al. used the Hellinger Affinity Kernel to infer the accuracy of sampling [7]. Tam et al. predicted the miss rate curves in real time [40].

***Locality measurement*** Reuse distance is a shorter name for the LRU stack distance defined by Mattson et al. [31]. The fastest precise method takes $O(n \log m)$ time, where $n$ is the length of the trace and $m$ is the size of data [34]. A variation of the algorithm powered the Cheetah cache simulator [38], widely distributed as part of the SimpleScalar tool set. By approximating long-distance reuses (with a guaranteed precision e.g. 99%), the cost can be reduced to $O(n \log \log m)$ [49]. This $n \log \log m$ algorithm is used in the two most recent sampling studies [35, 48]. In our experiments, the cost is several hundred times slowdown. The average cost reported in another study is as high as several thousand times slowdown (although with a different implementation) [35]. Zhong et al. gave a lower bound result indicating that the (asymptotic) cost cannot be further reduced for full reuse distance analysis [49]. Recent studies found efficient algorithms to parallelize the reuse distance analysis to run on MPI [33] or GPU [12, 22].

Time-based conversion [27, 36] and StatStack [19, 20] each gave a statistical formula to convert the reuse time distribution to miss rate, so did the working set theory [15]. These methods were not guaranteed to be correct or have a bounded error. This work gives a different conversion method based on the footprint formula and the correctness condition for the conversion.

If the cost of measuring $O(n)$ reuse windows was high, the cost of measuring $O(n^2)$ footprint windows was prohibitively high. In 2008, a sub-quadratic cost $O(n \log m)$ solution was proposed [17]. Later, the algorithm was implemented and made 70 times faster [45]. These two methods measure the full distribution, including for example, the maximum and the minimum sizes. Instead of the full distribution, Xiang et al. showed that the average footprint can be measured in linear time $O(n)$, and it is a monotone function [46]. Based on the HOTL theory in this paper, we have reduced the analysis cost to a negligible level using sampling and proved that the footprint function is concave.

***Program sampling*** Arnold and Ryder pioneered a general framework to sample Java code, i.e. the first few invocations of a function or the beginning iterations of a loop [2]. It has been adopted for hot-stream prefetching in C/C++ in bursty sampling [11] and extended to sample both static and dynamic bursts for calling context profiling [50]. Shadow profiling pauses a program at preset intervals and forks a separate process to profile in parallel with the base program [32, 42]. Before the new theory, the reuse distance analysis is not a good target for these techniques because of the uncertain length of the reuse windows. With the new theory, locality sampling becomes a similar task as frequency profiling. Like frequency profiling, the cost can be adjusted by simply changing the sampling rate.

***Filmer metrics in multi-threaded code*** The locality metrics in particular the footprint and the reuse distance have been extended to multi-threaded code by a number of studies, including composable modeling of shared footprint [18], statistical modeling in con-

current reuse distance [27], and direct measurement by multi-core reuse distance [35]. In a concurrent program, the reuse distance is affected by data sharing, thread interleaving and composition. These studies solved the problems by characterizing the relation between the private reuse distance (PRD) and the concurrent reuse distance (CRD). For loop-based code, Wu and Yeung gave a scaling model to predict how the reuse distance changes when the work is divided by a different number of threads [43]. These modeling techniques have found uses in co-scheduling [26] and multicore cache hierarchy design [44]. In this paper, we use footprint sampling and HOTL conversion in multi-threaded code and show the result that the reuse-window hypothesis holds there as it does in sequential code.

## 6. Summary

In this paper, we have compiled five Filmer metrics— the footprint, the inter-miss time, the volume fill time, the miss ratio curve and the reuse distance—and shown that they are mutually derivable. The derivations form a higher order relation. We prove that two of the miss-ratio derivations, by the footprint and by the reuse time, are mathematically equivalent. As a result, the correctness of the conversion depends on the reuse-window hypothesis. In addition, we prove that the average footprint is a concave function. We also give a direct definition of the fill time and show it to be unusable in practice. When comparing with the working set theory, we show the recurring theoretical result which we call Denning's law of locality. We show how the new theory complements and extends the previous theory.

Based on the new theory, we have developed a novel technique of locality sampling and used it to predict the miss ratio. When tested on the full suite of the SPEC 2006 benchmarks, the HOTL conversion predicts the miss ratio for over 3000 cache sizes at a speed 39% faster than cache simulation for a single cache size. The prediction is accurate compared to simulation and hardware counter results. Locality sampling obtains a similar accuracy by examining 0.9% of the execution and incurring a cost of less than 0.5% of the time of the unmodified code. When used to predict cache interference, the new technique takes 0.5% of the time of the exhaustive testing and predicts the interference accurately for 99.5% of the executions.

In summary, we have shown that the Filmer metrics can be measured in real time, and they are easy to compose and convert. We expect that the higher order theory and the sample technique will provide a new foundation for developing future techniques of locality analysis and optimization.

## Acknowledgments

The comparison with the working set theory was done in collaboration with Peter Denning. It was a rare privilege to discuss the field defining ideas with their creator. He was also the first to use the acronyms HOTL, WSLT and CLT when commenting on our paper and suggested the comparative view in Figure 4. Kim Hazelwood, Ramesh Peri and Tipp Moseley answered our questions about Pin and shadow profiling. We also thank Jacob Brock, Xipeng Shen, Donald Yeung, other colleagues, the reviewers of ASPLOS and the program committee especially P. Sadayappan for the careful review and constructive critiques, which are invaluable in improving the presentation of both the theory and the evaluation.

## References

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

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of PLDI*, pages 168–179, Snowbird, Utah, June 2001.

[3] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of SIGMETRICS*, pages 169–180, 2005.

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

[5] K. Beyls and E. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of HPCC. Springer. Lecture Notes in Computer Science Vol. 4208*, pages 220–229, 2006.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of PACT*, pages 72–81, 2008.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of PACT*, pages 339–349, 2005.

[8] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of ICS*, pages 150–159, 2003.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, pages 340–351, 2005.

[10] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of ICS*, pages 295–304, 2010.

[11] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of PLDI*, Berlin, Germany, June 2002.

[12] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng. A highly parallel reuse distance analysis algorithm on gpus. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[13] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, 1968.

[14] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), Jan. 1980.

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

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

[17] C. Ding and T. Chilimbi. All-window profiling of concurrent executions. In *Proceedings of PPoPP*, 2008. *poster paper*.

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

[19] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of HiPEAC*, pages 147–157, 2011. *best paper*.

[20] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of ISPASS*, pages 55–65, 2010.

[21] C. Fang, S. Carr, S. Önder, and Z. Wang. Path-based reuse distance analysis. In *Proceedings of CC*, pages 32–46, 2006.

[22] S. Gupta, P. Xiang, Y. Yang, and H. Zhou. Locality principle revisited: A probability-based quantitative approach. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[23] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[24] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, Nov. 1987.

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

[26] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of HiPEAC*, pages 201–215, 2010.

[27] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of CC*, pages 264–282, 2010.

[28] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation*, 13(1):1–38, 2003.

[29] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.

[30] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS*, pages 2–13, 2004.

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

[32] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of CGO*, pages 198–208, 2007.

[33] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

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

[35] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[36] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of POPL*, pages 55–61, 2007.

[37] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of ICSE*, 1976.

[38] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of SIGMETRICS*, Santa Clara, CA, May 1993.

[39] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of ICS*, pages 1–12, 2001.

[40] D. K. Tam, R. Azimi, L. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of ASPLOS*, pages 121–132, 2009.

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

[42] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of CGO*, pages 209–220, 2007.

[43] M.-J. Wu and D. Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of PACT*, pages 264–275, 2011.

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

[45] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPoPP*, pages 91–102, 2011.

[46] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.

[47] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache conscious task regrouping on multicore processors. In *Proceedings of CCGrid*, pages 603–611, 2012.

[48] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of ISMM*, pages 91–100, 2008.

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

[50] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of PLDI*, pages 263–271, 2006.

[51] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ASPLOS*, pages 129–142, 2010.

# A. Measuring the Direct Fill Time

As defined in Section 2.3, the direct fill time is the average length of all windows that have the same-size footprint. For a trace of $n$ accesses to $m$ data, the fill time algorithm counts all $O(n^2)$ windows but reduces the quadratic cost of counting in three ways. The solution is similar in design to all-window footprint measurement [17, 45].

***Counting by footprint size rather than window length*** The footprint size in a window is up to $m$, the size of data. Although there are up to $n$ windows ending at each element, there are at most $m$ different footprint sizes. By counting $m$ footprint sizes rather than $n$ windows, the algorithm reduces the counting cost from $O(n^2)$ to $O(nm)$.

Consider the example in Figure 10. Take the trace till the second access of $b$ (before |). It is the 6th access, so there are 6 windows ending there. Only 3 distinct elements are accessed, so the 6 windows have at most 3 different footprint sizes. From small to large, the 6 windows have a length 1 to 6 and footprints 1,2,3,3,3,3 respectively.

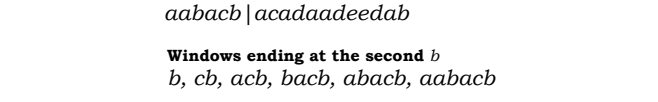



Figure 10: There are 3 different footprints for the 6 windows ending at the second $b$, so the 6 windows can be counted in 3 (instead of 6) steps.

***Relative precision footprint size*** By measuring data sizes with a relative precision, for example, 99% or 99.9%, the number of different footprint sizes becomes $O(\log m)$ instead of $m$. The cost of the algorithm becomes $O(n \log m)$.

***Trace compression*** A user sets a positive threshold $c$. The trace is divided into a series of $k$ intervals. Each interval has $c$ distinct elements (except for the last interval, which may have fewer than $c$ distinct elements). This is known as trace compression [28]. The algorithm traverses the trace interval by interval rather than element by element. The length of the trace is reduced from $n$ to $k$, and the cost becomes $O(ck \log m)$.

The algorithm computes the full distribution of the fill time $VT(v)$, from which we can compute the average fill time $vt(v)$. As far as we know, this is the first algorithm that computes the direct fill time with a guaranteed precision. We have implemented it and shown the results in the evaluation section.