A Higher Order Theory of Locality

Chen Ding and Xiaoya Xiang

Department of Computer Science
University of Rochester
Rochester, NY 14627
{cding,xiang}@cs.rochester.edu

Abstract

This short paper outlines a theory for deriving the traditional metrics of miss rate and reuse distance from a single measure called the footprint. It gives the correctness condition and discusses the uses of the new theory in on-line locality analysis and multicore cache management.

Categories and Subject Descriptors D.2.8 [*Metrics*]: Performance measures

General Terms measurement, performance

Keywords data footprint, reuse distance, miss rate

1. Footprint as a Foundation

A footprint is the amount of data accessed in a time window. A performance tool often measures the footprint for an execution window, i.e. taking a snapshot. A complete measure, however, has to consider *all* execution windows, which is quadratic to the length of the execution. Only recently the complete measurement has become computationally feasible. The fastest solution is a linear time algorithm that computes the average footprint for all window lengths [3]. In this section, we show that the average footprint $\overline{fp}$ is "higher order" because it can be used to compute three other locality metrics: the lifetime lf, miss rate mr and reuse distance rd.

The average footprint Let W be the set of $\binom{n}{2}$ windows of a length-n trace. Each window $w =< l, s >$ has a length l and a footprint s. The footprint function $\overline{fp}(l)$ averages over all windows of the same length l. There are $n-l+1$ footprint windows of length l. The average is the total footprint in these windows divided by $n - l + 1$.

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 $\overline{fp}(2) = (2 + 1 + 1)/3 = 4/3$.

Data lifetime in cache Let the cache size be c. Once a program brings in a data block in cache, its lifetime measures how long before the program accesses c other data and evicts this block. The expected lifetime is the average length of time the program takes to fill up an empty cache.

We define the lifetime function $lf(c)$ as the inverse of the footprint function, shown below. The invariance $\overline{fp}(lf(c)) = \overline{fp}(\overline{fp}^{-1}(c)) = c$ symbolizes the conversion that when the footprint is the cache size, the footprint window is the lifetime.

$$lf(c) = \overline{fp}^{-1}(c)$$

The conversion is shown visually in Figure 1. From the average footprint curve, we find cache size c on the y-axis and draw a flat line to the right. At the point the line meets the curve, the x-axis value is the lifetime $lf(c)$.

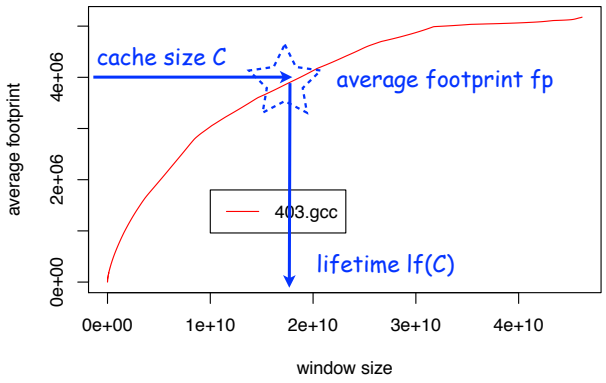

Figure 1: Finding data lifetime from data footprint

Miss rate It takes a program $lf(c)$ time to access c distinct data. It continues to access these data until the time $lf(c+1)$, when a new data block is accessed, triggering a capacity or compulsory miss. The time interval, $lf(c+1) - lf(c)$, is the miss-free period when the program uses only in-cache data. We use the interval as the average inter-miss gap $im(c)$. The miss rate, $mr(c)$, is its reciprocal as shown below.

$$mr(c) = \frac{1}{im(c)} = \frac{1}{lf(c+1) - lf(c)}$$

Cache conflicts (due to limited associativity) can be estimated based on reuse distance [1].

MSPC'12, June 16, 2012, Beijing, China.

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. The distribution function $rd(c)$ gives the fraction of data accesses that have reuse distance c. The capacity miss rate, $mr(c)$ is the total fraction of reuse distances greater than the cache size c. To compute $rd(c)$, we have

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

Correctness condition The derivations from footprint to lifetime and from lifetime to miss rate are not always correct. To understand correctness, consider the reuse distance and the footprint both as window statistics. The reuse distance is the footprint of its reuse window. A reuse window is special in that it starts and finishes with an access to the same datum with no intervening reuses. It is a subset of all windows (n out of $\frac{n(n+1)}{2}$). We define the average footprint over the reuse windows as $\overline{rfp}(l)$, the same way we define $\overline{fp}(l)$ over all windows. The correctness requires that the two functions be equal [2].

Theorem 1 (Correctness). *The footprint-based conversions are accurate if the average footprint of all reuse windows equals to the average footprint of all windows, for every window length l.*

Consider the trace "wxyzy" as an example. We show the average all-window footprint $\overline{fp}$ and reuse-window footprint $\overline{rfp}$ in one table. We inverse $\overline{fp}$ to get the lifetime and take the gradient to predict the miss rate, shown in the other table.

l	$\overline{fp}(l)$	$\overline{rfp}(l)$
1	1	1
2	2	2
3	$\frac{8}{3}$	0

c	$mr(c)$		accuracy
	fp pred	actual	
1	100%	100%	100%
2	66%	80%	83%

The prediction is accurate for cache size $c = 2$ but not $c = 3$, as stipulated by the correctness condition. As to real applications, an empirical evaluation shows good accuracy for the full suite of SPEC 2000 and 2006 benchmark programs [2].

2. A Higher Order Theory

In algebra, the term order refers 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 Figure 2.

As functions, footprint, lifetime, miss rate and reuse distance are mutually derivable, through differentiation from the footprint (as shown earlier) or in the reverse direction through integration. The footprint can be analyzed through sampling, e.g. by tracing a window of program execution periodically. By reducing the sampling frequency, the cost can be arbitrarily reduced. In addition when multiple programs run together, the total footprint is simply the sum of the individual footprints.

locality function	formal property	useful characteristics
3rd order: footprint, lifetime	concave/ convex	linear-time, amenable to sampling, composable (dynamic locality)
2nd order: miss rate	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 2: Locality metrics are mutually derivable.

In comparison, the miss rate is not composable. We cannot simply use the solo-run miss count as its miss count in a parallel execution, especially considering that the miss count changes depending on co-run peers. Neither is the reuse distance composable across multiple programs. According to the new theory, however, we can compute the miss rate and reuse distance from the footprint, and the footprint is not only composable but also measurable in real time. Therefore, the theory enables efficient and composable analysis of these other metrics and may have uses in:

- *On-line miss-rate curve analysis*. For a set of running programs, we can use on-line sampling to find out the miss rate of would-be solo execution for all cache sizes.
- *Cache conscious task regrouping*. Through the above on-line analysis, we can compose and compute the performance of all co-run combinations and re-schedule programs to minimize their interference in shared cache [4].
- *Optimal cache partitioning*. We can solve the generalized problem to determine what programs share what fraction of cache. The current policies of all sharing and all partitioned are just two extremes in a broad spectrum.

References

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

[2] X. Xiang, B. Bao, and C. Ding. Program locality sampling in shared cache: A theory and a real-time solution. Technical Report URCS #972, Department of Computer Science, University of Rochester, December 2011.

[3] 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.

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