Locality Analysis: A Nonillion Time Window Problem

Jacob Brock   Hao Luo   Chen Ding
Department of Computer Science
University of Rochester
Rochester, NY
{jbrock, hluo, cding}@cs.rochester.edu

ABSTRACT

The rise of social media and cloud computing, paired with ever-growing storage capacity are bringing big data into the limelight, and rightly so. Data, it seems, can be found everywhere; It is harvested from our cars, our pockets, and soon even from our eyeglasses. While researchers in machine learning are developing new techniques to analyze vast quantities of sometimes unstructured data, there is another, not-so-new, form of big data analysis that has been quietly laying the architectural foundations of efficient data usage for decades.

Every time a piece of data goes through a processor, it must get there through the memory hierarchy. Since retrieving the data from the main memory takes hundreds of times longer than accessing it from the cache, a robust theory of data usage can lay the groundwork for all efficient caching. Since everything touched by the CPU is first touched by the cache, the cache traces produced by the analysis of big data will invariably be *bigger* than big.

In this paper we first summarize the locality problem and its history, and then we give a view of the present state of the field as it adapts to the industry standards of multicore CPUs and multithreaded programs before exploring ideas for expanding the theory to other big data domains.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*

General Terms

Program Behavior, Performance

Keywords

Locality of References, Metrics, Big Data

1. INTRODUCTION

The memory systems in today's computers are in many ways simple; There are a few levels of cache, with the last level sometimes shared between cores, before the main memory and a storage device, and memory gets bigger, slower and cheaper the further it is from the cache. But despite the relatively intuitive nature of the memory hierarchy, the quantity of data that moves through it is immense. During a short program, the processor can make billions of data requests to the memory hierarchy.

With such a high volume of data, it is easy to understand that choosing which data to store and when is a very complex problem with very limited resources for a solution (since the CPU is busy using the data for its intended purpose). Most solutions to that problem rely on making on-line observations and predictions with minimal hardware resources (e.g. Dynamic Insertion Policy [17] and Re-Reference Interval Prediction [11]). This enforced minimalism has had the dual effect of requiring creativity for on-line analysis, and increasing the importance of off-line analysis.

2. LOCALITY AS A BIG DATA PROBLEM

The first hint that a theory of data locality would be useful came when virtual memory systems exhibited the problem of thrashing; So many page faults occurred in the RAM that the system would spend more time retrieving data than processing it [6]. The first step was to develop the working set model, which notes that a program uses only a subset of its data in any time period [4].

Locality analysis measures the active data usage. Given a window of execution, the *footprint* is defined as the amount of distinct data accessed in the window. For an execution of length n, we define $fp(T_w)$ as the average footprint of all windows of the length T_w. 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 average footprint for length-2 windows is $fp(2) = (2 + 1 + 1)/3 = 4/3$.

Footprint measurement is a big-data problem. The average footprint is the expected quantity of data in any size of trace window. To measure it, we must count the number of unique data in *every substring* of a trace. Assuming a typical benchmark program running for 10 seconds on a 3GHz processor, we have $3E10$ CPU cycles in the execution and need to measure the footprint in $4.5E20$ distinct windows.

As a reference, we show the scale of the footprint problem in Figure 1. As the length of execution increases from 1 second to 1 month, the number of CPU cycles (n) ranges from $3E9$ to $2E15$, and the number of distinct execution windows $\binom{n}{2}$ from $4.5E18$ to $5.8E29$, that is, from 4 sextillion to over a half nonillion.

Early methods addressed the immensity of this problem in three ways. The first is to measure a subset of windows, for example, all windows starting from the beginning of a trace [1] or windows of a single length (the CPU scheduling quantum) [21]. The second is an approximation under stochastic or probabilistic assumptions [2,3,7,13,19,20]. The

Big Data Analytics Workshop 2013 Pittsburgh, Pennsylvania USA

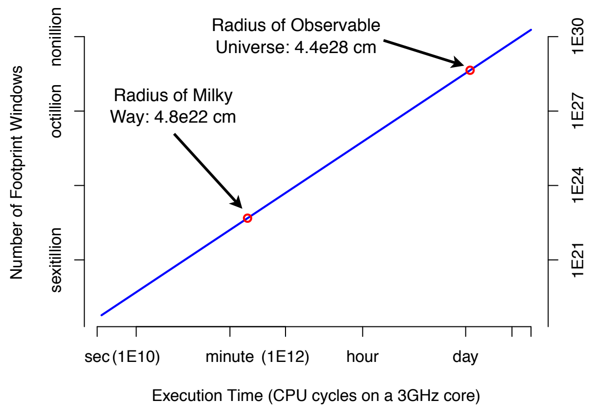


Figure 1: The scale of the problem shown by the number of footprint windows in a program execution

last is sampling [9]. With just a single-window length, the result is not complete. In approximation or sampling, the accuracy is unknown.

Our recent work developed new algorithms to measure the all-window footprint for either the full distribution [8,22] or just the average [23]. The average-footprint algorithm takes time linear to the length of the execution.

Next we give an overview of the recent advance in locality theory and outline ongoing work on multicore cache management. Both are based on the solution to the big-data footprint problem that we have just discussed.

3. PRESENT AND FUTURE OF LOCALITY

3.1 Higher Order Theory of Locality

The working set theory was pioneered in Peter Denning's thesis work [4]. His 1968 paper established the relation between the miss rate and the inter-reference interval (iri). For each access, the inter-reference interval is the time since the previous access to the same datum. Given a window length T, the time-window miss rate $m(T)$ is the fraction of data accesses whose inter-reference interval is greater than T:

$$m(T) = P(iri > T)$$

Using the footprint as the cache size, we showed a similar formula for the miss rate $mr(C)$ for fully-associative LRU cache of size C [25]. Let the window length T_c be such that $fp(T_c) = C$. The LRU miss rate is:

$$mr(C) = mr(fp(T_c)) = P(iri > T_c)$$

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 [16]. 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 LRU miss rate, $mr(C)$, is the total fraction of reuse distances greater than the cache size C, i.e. $mr(C) = P(rd > C)$. Consequently,

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

Figure 2 shows the footprint, miss rate curve, and the reuse distance for an example program *bzip2*. All three are program specific and can be used by programmers and hardware designers to quantify and optimize the cache usage of programs. The working set models are widely used in virtual memory management [5], the miss rate curve in computer design [10, 16] and memory management in OS and virtual machines [26, 28], and reuse distance in program analysis and optimization (see Section 6 in [27]).

Through the conversion formulas like the ones in this section, we have shown that the locality metrics of footprint, miss rate, and reuse distance are mutually convertible [25]. They form a higher-order relation where we can compute higher-order metrics by taking the sum in lower-order metrics or in the reverse direction, compute lower-order metrics by taking the difference in higher-order metrics. The conversion formulas and the properties including the correctness conditions are called collectively the *higher-order theory of locality*.

For the locality theory, it is vital that the footprint fp is uniquely defined for each window size. As described in Section 2, the footprint is so defined as a big-data problem of all-window analysis.

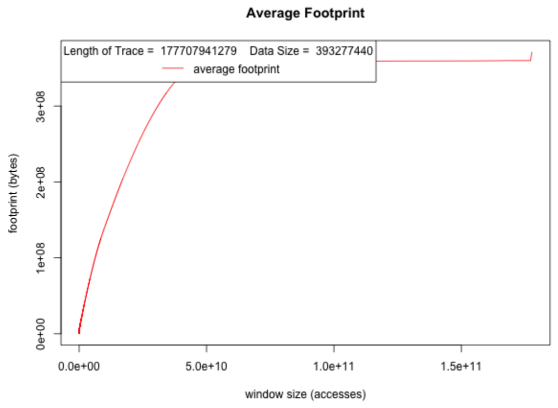
Among the locality metrics, the footprint is unique in its three properties. First, it is composable in that the aggregate footprint of a set of programs is the sum of their individual footprints (assuming no data sharing). Second, it can be measured efficiently through sampling, with 0.5% visible cost on average [25]. Using the locality theory, we can derive other metrics from the footprint. As a result, we can now measure the reuse distance and the miss rate curve efficiently and compose them for multi-programmed workloads.

3.2 Cache Sharing and Multithreading

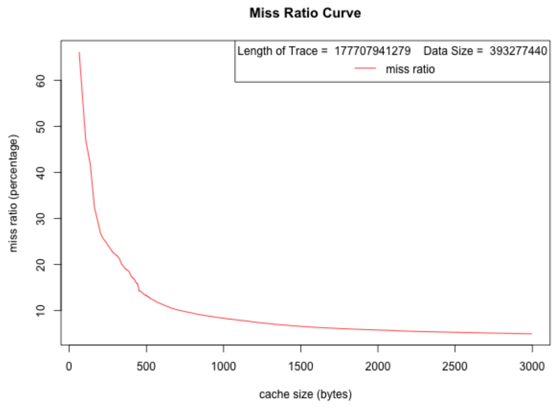
Cache sharing is done in two different ways: by partitioning the cache among the programs so that they each have effectively a cache some fraction of the size of the whole cache, or by allowing all programs to use the whole cache with an LRU-like replacement scheme. In the second scheme, either program's data may dominate the cache at any given time, depending on its memory demand. On a modern machine, a single thread can replace the entire contents of a 4MB cache in milliseconds, so the "ownership" of the cache can change, or even oscillate, extremely quickly.

A few papers have approached the problem of predicting shared cache performance. [14] presents a fairness-based model for cache sharing. In [3], a model for multithread cache contention is presented. [9] developed a trace-based locality theory for multithreaded programs, analyzing how the reuse distance profile of a thread changes with a co-run thread.

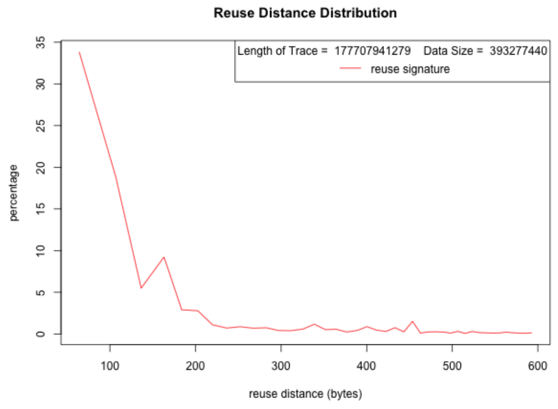
A new metric for quantifying the relationships between threads is the shared footprint of [15]. To illustrate, Figure 3 shows an interleaved trace with accesses by two threads. A window has non-zero shared footprint if a datum is accessed by two (or more) threads in the window. Therefore, the *shared footprint* gives the quantity of data that is *actively* shared. The *sharing ratio* is the ratio of shared footprint to the total footprint. This metric has potential to help with cache sharing design, identifying false sharing, and thread scheduling. Possible extensions to the idea include defining a separate shared footprint for every quantity of threads (not



(a) The average footprint function gives the average amount of distinct data in window of a given size. With a trace length of nearly 180 billion accesses, at each window length x the average footprint of $180 \times 10^9 - x$ windows is calculated.



(b) The miss rate curve for a fully associative LRU cache, derivable from the footprint function, shows what size cache is necessary for best efficiency.



(c) The reuse distance profile, derivable from the miss rate curve or the footprint function.

Figure 2: Examples of the locality metrics *footprint*, *reuse distance*, and *miss rate* for an example run of bzip2. The three metrics are mutually computable according to the higher-order theory of locality outlined in Section 3.1.

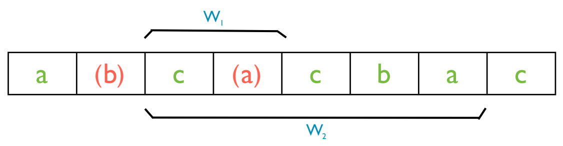


Figure 3: Shared footprint in an interleaved execution of two threads. The two accesses by one of the threads are colored red and marked with parentheses. The shared footprint is zero for W_1 because the two threads do not access the same datum and one for W_2 because both threads access a.

just shared vs. unshared), and quantifying the "popularity", or number of threads in which a block is active in a temporal window.

4. LOCALITY BEYOND THE CHIP

The lessons of locality theory can have implications beyond memory management. Any data set that can be formulated as a long stream of discrete data can be analyzed using methods developed for memory traces; this includes web search terms, website views, social media communications, product sales online or in stores, and even play calls in sports (e.g. series of pitches in baseball). As an example application, an average footprint curve or reuse distance signature might aid in identifying the authorship of written works, or even predicting their success. One can imagine that a book such as Dr. Seuss's *Green Eggs and Ham* would have a particularly uncommon footprint curve[1] [18].

Other applications might include classifying traffic flow patterns, or identifying "types" of customers for web retailers based on what products they view and buy, and how often they revisit these products. A series of short revisit times (analogous to reuse distance) might indicate that a customer is planning a purchase. Recognizing trends in searches and purchases can help retailers plan inventory in the same way that caches use prefetching. In the way that locality analysis has been used for multiprocessor scheduling (see [12,24,29]), it might in some way be applicable to public event scheduling for minimizing traffic congestion. While machine learning techniques have already been applied to most of these topics, locality theory can provide a new perspective on the data.

5. CONCLUSIONS

Developed to analyze long lists of data, locality theory has mature metrics that can be applied to a large class of big data problems. At the same time, the theory is still evolving to inspire, enable and accommodate new architecture. As locality theory matures with multithreading and multi-tiered metrics, and other areas of big data mature with the rise of social media, medical data, and market data, researchers in the often disparate fields of systems design and machine learning are being presented with a rare opportunity to identify parallel problems between the fields and draw from each other's lessons. This is an opportunity not to be missed!

[1]Seuss famously wrote the book using 50 distinct words in response to a bet he made with his publisher, who did not think it possible to write a successful story with such a limited vocabulary.

6. REFERENCES

[1] A. Agarwal, J. L. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

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

[3] 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.

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

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

[6] P. J. Denning. The locality principle. In J. A. Barria, editor, *Communication Networks and Computer Systems*, pages 43–67. 2006.

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

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

[9] 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.

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

[11] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[12] 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.

[13] 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.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of PACT*, pages 111–122, 2004.

[15] H. Luo, X. Xiang, and C. Ding. Characterizing active data sharing in threaded applications using shared footprint. In *Proceedings of the The 11th International Workshop on Dynamic Analysis*, 2013.

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

[17] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of ISCA*, pages 381–391, 2007.

[18] D. Seuss. *Green Eggs and Ham*. Random House Children's Books, 1960.

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

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

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

[22] 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.

[23] 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.

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

[25] X. Xiang, C. Ding, B. Bao, and H. Luo. A higher order theory of cache locality. In *Proceedings of ASPLOS*, 2013.

[26] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of OSDI*, pages 103–116, 2006.

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

[28] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of ASPLOS*, pages 177–188, 2004.

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