All-Window Profiling of Concurrent Executions

Chen Ding[†‡] and Trishul Chilimbi[†]
[‡]Computer Science Department, University of Rochester
[†]Microsoft Research

Categories and Subject Descriptors C.4 [*performance of systems*]; D.3.4 [*programming languages*]: processors

General Terms measurement, performance

Keywords data footprint, thread interleaving, concurrent systems

Abstract

This paper first demonstrates the need for all-window profiling in a concurrent execution, then presents an approximate algorithm, and finally discusses related work.

1. Footprints

For a window over an execution trace, the footprint is the amount of data being accessed in the window. Footprint is a basic metric of program locality and has been used to compute the life-time of data blocks once they are loaded into cache [1] as well as the effect of cache sharing by multiple programs [3, 7]. Figuratively, the footprint determines how a program treads out its old data and how multiple programs step over each other in cache.

We used a sampling method to collect the data for the paper. At each memory access, with equal probability the method picks a range r and a window size x within the range. The size x uniquely determines the window, which includes the curent access and the previous $x-1$ accesses. Then the algorithm measures the volume of data in the window. It ensures that all time ranges are sampled equally. However, since the total number of windows for a trace of length n is $O(n)$, the sampling rate is only $\frac{n}{n^2} = \frac{1}{n}$. The exceedingly low sampling rate raises the question whether we can measure all $O(n)$ windows to verify the accuracy of sampling.

Consider an execution trace of a commercial server application. The execution consists of 22 concurrent threads for a total of 1.7 billion memory accesses. Figure 1 shows the instruction footprints for thread 40, which accounts for 29% of instruction accesses. Both axes use a fine-grained logarithmic scale we call 8-wide histogram, where each bin of the (base 2) logarithmic scale is divided into 8 equal-size sub-bins (when the bin size is no smaller than 8). The x-axis shows 200 logarithmic ranges between 0 and $2^{(200/8+2)} = 2^{27}$ or 134 million instruction accesses. The y-axis shows 90 logarithmic ranges for the footprint up to 9,750 instruction blocks.

The five curves show the cumulative distribution of footprints: for each time window of size x, up to 0%, 5%, 50%, 95%, or 100% of footprints have a size under the y value marked by these five curves. The middle curve is labeled "expected" as it shows the median value. The other four curves show more extreme cases. For example, the "expected" curve shows that half of the windows of 800 instruction accesses touched 128 instruction blocks (8KB) or less, and the "0%" curve shows that there existed long periods of time, 100K accesses, where very few data, about 10 blocks, were accessed.

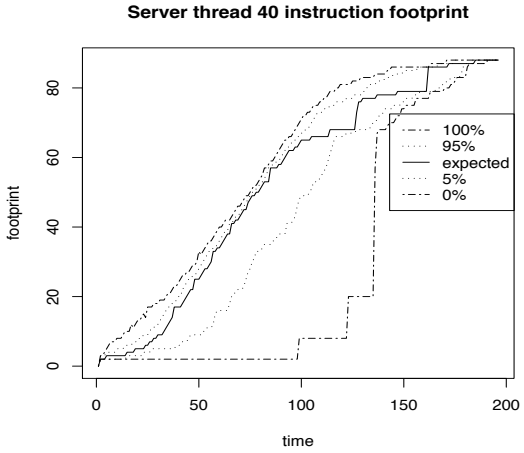

Figure 1. The 8-wide histogram for the instruction footprint of server thread 40, measured by sampling at rate $\frac{1}{n}$

The median footprint is not smooth and has small bumps and breaks. However, the area between "5%" and "95%" shows that the middle 90% of footprints follow a smooth, consistent upward trend. The rate of increase slows down and takes a shallower slope as the window size reaches 25K instruction accesses. The bi-linear shape is common in the histogram of reuse distances, where the point of the knee gives the size of the working set and signals a change of locality. The knee in the footprint curve represents not a change of locality but a change of interference. It shows that the rate of interference by the thread drops down over longer periods of execution. This is expected but the question for each program is where and how much the rate of interference changes. We will not have an complete answer unless we can measure the footprint for all windows.

2. Thread Interleaving

In modern concurrent applications, the execution of threads may not interleave uniformly. This is generally the case for client applications with asymmetrical functions for each thread, where some may execute ten times more instructions than others. Even for symmetric server workloads, the relative rate of execution of parallel threads may change from one phase to another. The degree of interleaving strongly affects the use of shared resources such as cache and memory.

Using the same $\frac{1}{n}$ sampling rate as in the case of footprint, we have measured the interleaving between two threads, thread 40 and thread 88, in the execution trace mentioned before. Since the two

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
ACM 978-1-59593-960-9/08/0002.

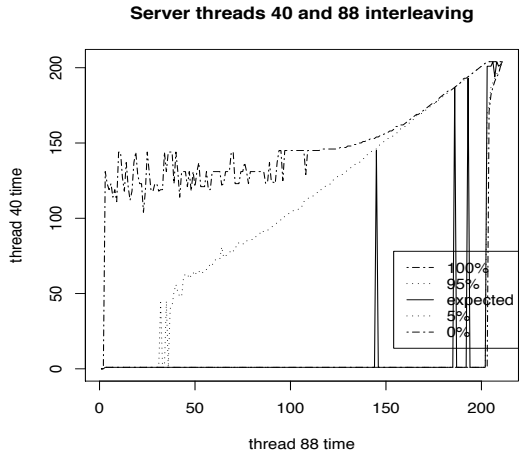

Figure 2. The 8-wide histogram of the interleaving between server threads 40 and 88, measured by sampling at rate $\frac{1}{n}$

threads are active server threads, one may expect uniform interleaving. However, the sampling result in Figure 2 shows that uniform interleaving is an exception rather than the norm. It happens only for 5% of windows of a size larger than 100 accesses. In most cases, the median degree of interleaving is zero, meaning that only one thread is executing. However, without all-window statistics, we cannot say for sure whether the interleaving is truly imbalanced in all windows and whether the overall imbalance is the same as what we observe from the samples.

3. Approximate All-Window Profiling

Given an n-element execution trace $t_1, t_2, \ldots, t_n$, the basic algorithm traverses the trace from left to right. At each element t_i, it counts all the windows ending at t_i. The c-approximate analysis guarantees that the measured result for each window be between c and 100% of the actual result, where c is between 0 and 1.

The trick of the analysis is to count multiple windows at each step. This is done by building on the idea of an approximate profiling algorithm by Ding and Zhong [4]. For each t_i, the algorithm maintains a division of the trace $t_1, t_2, \ldots, t_i$ in $O(\log i)$ time ranges, $r_1, \ldots, r_k$. It keeps track of the total count, either the number of data blocks or instructions, for each time range. A backward traversal of them from r_k to r_1 gives the cumulative count for windows that begin in r_i and end at t_i. This cumulative count of each r_i is used for all windows starting in r_i. Hence the algorithm counts all i windows in $O(\log i)$ instead of i steps.

To profile all footprints, we build on the Ding-Zhong algorithm directly. At each point in the trace t_i, the algorithm keeps $O(\log i)$ ranges organized in a search tree. Each range stores the number of last accesses made during the time range. An element a has its last access in time range r if a is accessed during time range r but not again till time i. The idea of storing last accesses is due to Bennett and Kruskal [2] and the use of search tree is due to Olken [5].

The algorithm maintains the partition of time ranges as follows. As it goes through the trace, it creates a new time range for each access. Periodically, it stops and compresses the time ranges. By choosing the length of the period to be proportional to $\log i$, it bounds the cost of each compression in $O(\log i)$ and the amortized cost for each access in $O(1)$. The exact formula for the periodic compression is the same as the one used by Ding and Zhong [4], which depends on the desirable precision c.

For thread interleaving, the algorithm similarly divides the trace into a logarithmic number of time ranges and maintains the division using periodic compression. However, there is no need to organize the time ranges in a search tree, unlike the case of footprint measurement.

4. Related Work

Agarwal et al. counted the number of cold-start misses for different size windows starting from the beginning of a trace [1]. For time-sharing environments, Suh et la. used the footprints to evaluate the effect of scheduling quantum on cache locality [7]. Chandra et al. modeled the parallel execution where the locality of one thread was affected by the footprint of another thread [3]. The last two methods approximated the average footprint by solving a recursive equation. Let $E[w_t]$ be the average footprint for a window of size t, and $M(f)$ be the average miss rate for cache of size f (estimated from the reuse signature). For each memory access, the footprint either increments by one or stays the same depending on whether the accessed data is new or not. This is equivalent to checking whether the access is a miss in an cache with infinite size. The expected footprint at time $t + 1$ can then be computed from the footprint at t as follows

$$E[w_{t+1}] = E[w_t](1 - M(E[w_t]) + (E[w_t] + 1)M(E[w_t])$$

Suh et al. simplified it into a differential equation [7]. Chandra et al. computed the recursive relation bottom up. A third technique, recently developed by Shen et al., estimated the footprint using statistical equations based on the distribution of reuse times [6].

The previous methods compute or estimate the average but not the complete distribution. A drawback is that the average can be strongly influenced by a few large values. The problem is inherent since the previous methods do not actually measure all windows. Our new algorithm, though not yet implemented, would be able to overcome this limitation.

Acknowledgments

The authors wish to thank Bao Bin at Rochester and the reviewers of PPOPP 2008 for their comments, which helped to improve the presentation.

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] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, pages 353–357, July 1975.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2005.

[4] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

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

[6] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–61, 2007.

[7] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *International Conference on Supercomputing*, pages 1–12, 2001.