All-Window Profiling and Composable Models of Cache Sharing

Xiaoya Xiang Bin Bao Tongxin Bai
Chen Ding
Computer Science Department
University of Rochester
Rochester, NY 14627
{xiang,bao,bai,cding}@cs.rochester.edu

Trishul Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Abstract

As multi-core processors become commonplace and cloud computing is gaining acceptance, more applications are run in a shared cache environment. Cache sharing depends on a concept called footprint, which depends on all cache accesses not just cache misses. Previous work has recognized the importance of footprint but has not provided a method for accurate measurement, mainly because the complete measurement requires counting data access in all execution windows, which takes time quadratic in the length of a trace.

The paper first presents an algorithm efficient enough for off-line use to approximately measure the footprint with a guaranteed precision. The cost of the analysis can be adjusted by changing the precision. Then the paper presents a composable model. For a set of programs, the model uses the all-window footprint of each program to predict its cache interference with other programs without running these programs together. The paper evaluates the efficiency of all-window profiling using the SPEC 2000 benchmarks and compares the footprint interference model with a miss-rate based model and with exhaustive testing.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Modeling techniques; D.2.8 [*Metrics*]: Performance measures

General Terms measurement, performance

Keywords data footprint, cache interference, reuse distance, composable models

1. Introduction

A basic question in multi-core processor design is whether to use partitioned or shared cache. Partitioned cache can be wasteful when only one program is running. Shared cache is risky since programs interfere with each other.

Current multi-core processors use a mix of private and shared cache. Intel Nehalem has 256KB L2 cache per core and 4MB to 8MB L3 cache shared by all cores. IBM Power 7 has 8 cores, with 256KB L2 cache per core and 32MB L3 shared by all cores. Cache sharing becomes a significant factor. The performance of a program may change drastically depending on what other programs are running.

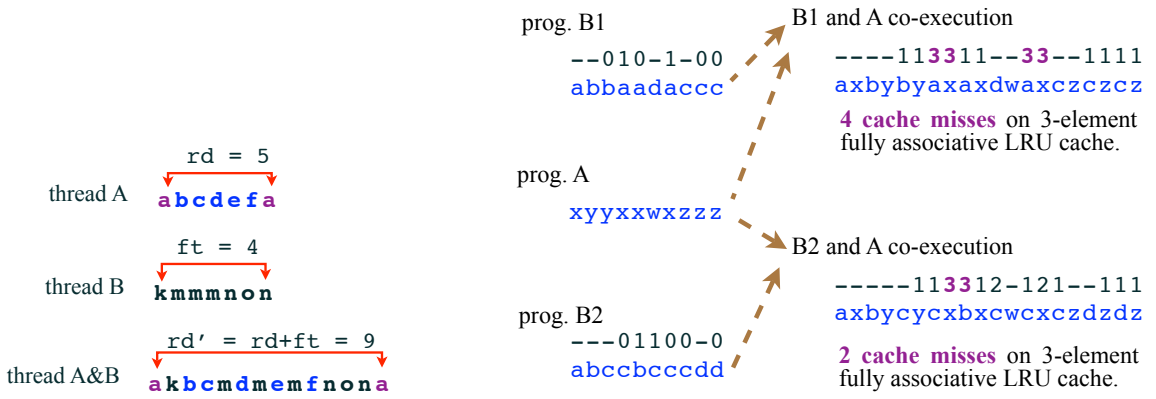
Cache sharing models can be loosely divided into on-line and off-line types. On-line models are used to minimize the interference among programs that are currently being executed on a given machine. Off-line models do not improve performance directly but can be used to understand the causes of interference and to predict its effect before running the programs (so they may be grouped to reduce interference). An off-line model has three advantages over on-line analysis:

- *All data accesses, not just cache misses.* To be efficient, on-line models consider just cache misses in select execution periods. Cache interference, as we will show, depends on more than just cache misses. An off-line model can measure the effect of all data accesses in an execution.
- *"Clean-room" statistics.* An off-line model measures the characteristics of a single program unperturbed by other programs. Such "clean-room" metrics avoids the chicken-egg problem when programs are analyzed together: the interference depends on the miss rate of co-running programs, but their miss rate in turn depends on the interference.
- *Composable models.* The clean-room, off-line models of individual programs can be composed to predict the interference in any program group. There are 2^P co-run combinations for P programs. The composable model makes 2^P predictions using P clean-room single-program runs rather than 2^P parallel runs. This property helps in model validation—errors cannot as easily cancel each other (and be hidden) by accident when each model is used in making 2^{P-1} predictions.

To accurately model cache sharing, we collect all-window statistics. In an execution of n run-time instructions, the number of different, non-empty windows is $\binom{n}{2} = \frac{n*(n+1)}{2}$ or $O(n^2)$. What's more, it takes $O(n)$ to count all data access in a window. There had been no prior solution that could measure such all-window statistics for large-scale traces.

The first part of this work is an algorithmic advance that makes all-window profiling possible. We use three techniques to reduce the cost of all-window analysis. The first is to count by footprint sizes rather than window sizes. The second is to use relative precision in footprint sizes. The third is to use trace compression. As a result, the new algorithm can count a large number of windows in

PPoPP'11 February 12–16, 2011, San Antonio, Texas, USA.



(a) In shared cache, the reuse distance in program A is lengthened by the footprint of program B.

(b) Programs B1 and B2 have the same miss rate. However, A and B1 incur 50% more misses in shared cache than A and B2. The difference is caused not by data reuse but by data footprint.

Figure 1. Example illustrations of cache sharing.

a single step. The asymptotic cost is reduced from $O(n^3)$ to significantly below $O(n \log n)$.

Based on all-window profiling, the second part of this work gives a algorithm to compute the circular effect between cache interference and program miss rate. We model the effect using an recursive equation, solve it using an iterative algorithm, and prove the convergence.

The third part of the work is an evaluation of all-window footprint in 14 SPEC2K benchmark programs, with traces as long as 100 billion data accesses. We show, for the first time as far as we know, the footprint distribution in up to a sextillion (10^{21}) windows. We show the quantitative difference between footprint and miss rate. In addition, we predict cache interference using the composable model and compare footprint-based prediction with one based on miss rate and one on exhaustive testing.

The rest of the paper is organized is as follows. Section 2 gives the background especially the asymmetrical effect of cache sharing. Section 3 describes the new algorithm for measuring all-window footprints. Section 4 develops the composable model of cache sharing. Section 5 evaluates all-window profiling and the composable model on SPEC 2K benchmarks. Section 6 discusses possible uses of the new algorithm. Finally, the last two sections discuss related work and summarize.

2. Background on Off-line Cache Models

Reuse windows and the locality model For each memory access, the temporal locality is determined by its reuse window, which includes all data access between this and the previous access to the same datum. Specifically, whether the access is a cache (capacity) miss depends on the reuse distance, the number of distinct data elements accessed in the reuse window. The relation between reuse distance and the miss rate has been well established [14, 19, 28]. The capacity miss rate can be defined by a probability function involving the reuse distance and the cache size. Let the test program be A.

$$P(\text{capacity miss by A alone}) = P(\text{A's reuse distance} \geq \text{cache size})$$

Footprint windows and the cache sharing model Off-line cache sharing models were pioneered by Chandra et al. [8] and Suh et al. [25] for a group of independent programs and extended for multi-threaded code by Ding and Chilimbi [11], Schuff et al. [20], and Jiang et al. [16] Let A, B be two programs share the same cache but do not shared data, the effect of B on the locality of A is

$$P(\text{capacity miss by A when co-running with B}) = P((\text{A's reuse distance} + \text{B's footprint}) \geq \text{cache size})$$

Given an execution window in a sequential trace, the *footprint* is the number of distinct elements accessed in the window. The example in Figure 1(a) illustrates the interaction between locality and footprint. A reuse window in program A concurs with a time window in program B. The reuse distance of A is lengthened by the footprint of B's window.

Footprint, locality, and the miss rate An implication of the cache sharing model is that cache interference is asymmetric if locality and footprint are different metrics of a program. A program with large footprints and short reuse distances may disproportionally slowdown other programs while experiencing little to no slow down itself. This is confirmed in our experiments. In one program pair, the first program shows near 85% slowdown while the other program shows only 15% slowdown.

Locality is determined by data access in reuse windows. There are up to n reuse windows in a trace of n data accesses. Footprint is determined by data access in all windows. There are $\binom{n}{2} = \frac{n*(n+1)}{2}$ footprint windows. Therefore, measuring footprint is computationally a much harder problem than measuring locality.

The conventional metric of cache performance is miss rate. The miss rate is defined for all windows but it measures only the number of misses not the total amount of data access. In particular, it does not count if an accessed datum is already in cache. The error may be large for shared cache, whose sizes are 4MB to 32MB in size on today's machines. In fact, neither reuse distance nor footprint can be determined by counting cache misses alone. Furthermore, the miss rate is machine dependent and needs to be measured for each cache configuration.

Miss rate $\neq$ cache interference Figure 1(b) shows three short program access streams. Programs B1 and B2 have the same set of reuse distances and hence the same capacity miss rate. However when running with program A, B1 causes twice as many capacity misses as B2. It shows that locality alone does not fully determine cache interference. Reuse distances and cache misses are not sufficient. We need to know data footprint.

In general when considering the interference by program B on program A, it is necessary to know B's footprint in windows of all sizes since the size of A's reuse windows can be arbitrary. This requires all-window footprint analysis, which we show next. A similar algorithm may be used to collect other types of all-window statistics such as all-window miss rates or all-window thread interleaving [10].

3. All-Window Footprint Analysis

Let n be the length of a trace and m the number of distinct data in the trace. We consider so-called "on-line" profiling, which traverses the trace element by element but does not store the traversed trace. Instead it stores a record of the history. The naive algorithm works as follows. At each element, it counts the footprint in all the windows ending at the current element. There are $O(n^2)$ windows in the trace. The naive algorithm counts each of them. The cost is $O(n^2)$. This does not include the cost of measuring the footprint in each window, which is up to n in size. In the rest of this section, we describe four techniques to reduce this complexity. In the title, we refer to each algorithm by the main idea and the asymptotic complexity.

3.1 Footprint Counting: The NM Algorithm

Assume that the first i accesses of a trace refer to m_i data. There are i windows ending at the ith element and i footprints from these windows. The i footprints have at most m_i different sizes. The first idea to reduce the cost is counting by m_i footprint sizes rather than by i windows. For each size, we count all footprints of this size in a single step. The cost for counting all footprints is $O(m)$ per access instead of $O(n)$. Instead of counting $O(n^2)$ windows one by one as in the naive algorithm, the NM algorithm counts the $O(n^2)$ windows in $O(nm)$ steps.

In the example in Figure 2, consider the second access of b (before "|"). It is the 6th access in the trace, so there are 6 windows ending there. However, only 3 distinct elements are accessed, so the 6 footprints must have one of the 3 sizes. Indeed, the footprint size of the 6 windows is 1,2,3,3,3,3 respectively.

aabacb | acadaadeedab

Windows ending at the second *b*
b, cb, acb, bacb, abacb, aabacb

Figure 2. There are 3 different footprint sizes for the 6 windows ending at the second b. The 6 footprints are counted in 3 steps by the NM algorithm.

To count windows by footprint sizes, we represent a trace as follows. At each point, we store the position of the last access of each datum seen so far. Then, all windows between a consecutive pair of last accesses have the same footprint and are counted in one step. For example in Figure 2, there is no other last access before access 4 (the last access of a), so the four windows, starting at 1 to 4 and ending at 6, have the same footprint and can be measured in one step. A method based on last accesses has previously been used by Bennett and Kruskal to measure reuse distance [2].

3.2 Relative Precision Footprint: The NlogM Algorithm

The use of large cache is affected mostly by large footprints. For a large footprint, in most cases we care about only the first few digits not the exact footprint. The second idea is to measure footprints in approximate sizes, in particular, in a relative precision such as 99% or 99.9%. The number of different footprint sizes becomes $O(\log m)$ instead of m.

To maintain a relative position, we store the last-access information of a trace in a *scale tree* structure developed by Zhong et al. [28] In a scale tree, each (constant-size) tree node represents a window in the trace. The number of nodes depends on the required precision. Asymptotically, a scale tree has $O(\log m)$ nodes [28]. If we consider last accesses as markers in the trace, the precise NM algorithm uses m markers, while the relative precision algorithm uses $O(\log m)$ markers. Hence, the cost is reduced to NlogM. The idea of the NlogM algorithm was first described in a 2-page poster paper without a complete algorithm in PPOPP 2008 [10].

3.3 Trace Compression: The CKM Algorithm

The third idea is to analyze windows ending within a group of elements in a cost similar to analyzing windows ending at a single element. A user sets a threshold c. The trace is divided into a series of k intervals called *footprint 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. It has been used by Kaplan et al. to shorten a trace while still preserving the miss rate behavior [17]. In footprint analysis, it allows us to traverse the trace interval by interval rather than element by element. The length of the trace is reduced from n to k. Next we describe the concepts and the algorithm and show that this use of compression does not lose precision in footprint analysis.

Definition *The footprint interval.* Given a start point x in a trace and a constant c, the footprint interval is the *largest time window* from x that contains accesses to exactly c distinct data elements.

Figure 3 (b) shows an example trace and its division into footprint intervals. The division is unique. It maximizes the length of each interval in order to minimize the number intervals. We define the length of an interval as the number of elements it has. In this example, the length is 9 for the first interval I_1 and 8 for the second interval I_2.

We denote the series of footprint intervals by I_i, data accesses by π, the first access s and the number of footprint intervals K. The trace partition is:

$$I(\pi, s, c) = \cup_{i=1}^{K} I_i$$

where $\forall i$, I_i is a footprint interval of size c; and $\forall i \neq j$, I_i and I_j do not overlap.

Windows contained within a footprint interval have a footprint size between 1 and c and can be easily measured. The more difficult question is how to measure the footprint of a window that spans more than one interval. To show the algorithm, we first introduce some new terms. Assuming the current point is the start of the current interval, we define

Definition NEXT-FIRST-ACCESSES, the first accesses to distinct data after the current point.

Definition PREVIOUS-LAST-ACCESSES, the last accesses to distinct data before the current point.

Definition SUB-INTERVALS. The current interval is divided by the next-first-accesses into sub-intervals.

Definition PAST-PARTITIONS. The trace before the current point is divided by the previous-last-accesses into past partitions.

Figure 3 shows for the first element in the second interval, the 3 next-first-accesses, the 3 previous-last-accesses, the 3 sub-intervals,

and the 3 past partitions. A past partition may span multiple intervals.

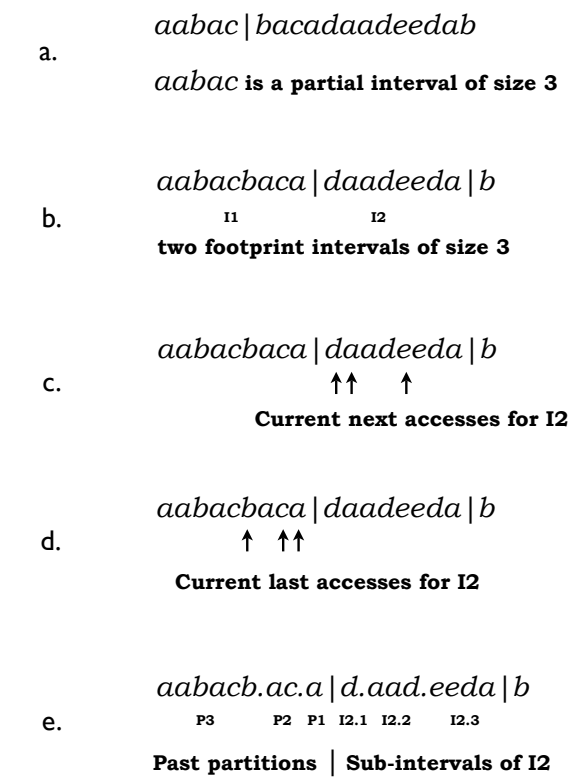


Figure 3. Illustrations of the definitions used in the CKM algorithm

To count distinct footprint sizes in all windows, it is sufficient to count between sub-interval and past partition markers, as is stated by the following theorem. Note that the footprint count for windows spanning multiple intervals is precise regardless of the choice of c.

Theorem 3.1. *All windows starting within the same past-partition and ending within the same sub-interval have the same footprint.*

Proof Suppose a time window starts from the past-partition p and ends at the sub-interval s. Suppose the successive data accesses within p are $p_a, p_{a-1}, ..., p_1$ and the successive data accesses within s are $s_1, s_2, ..., s_b$. By definition, the previous-last-access in p is p_1. The next-first-access in s is s_1. Let w^* be the window from p_1 to s_1. Let the footprint of w^* be fp^*.

Fix the start position at p_1 and consider the window $w1$ that ends at s_2. Since each sub-interval has only one next-first-access, s_2 is not a first access and does not add to the footprint. The footprint of $w1$ is fp^*. Similarly, we see that all windows starting at p_1 and ending at $s_i, i = 2, ..., b$ have the footprint fp^*.

Now let's fix the end position at s_1 and consider the window $w1$ that starts at p_2. Since each past-partition has only one previous-last-access, p_2 cannot be a previous-last-access and therefore does not add to the size of the footprint. The footprint of $w1$ is fp^*. Similarly, we see that all windows starting at $p_a, ..., p_2, p_1$ and ending at s_1 have the footprint fp^*. Composing these two cases, we have the stated conclusion. ∎

By Theorem 3.1, the measurement of all-window footprints is reduced to measuring the footprint in windows between every past partition and every sub-interval. The number of past-partitions is up to m. The number of sub-intervals is c. Therefore, the number of distinct footprints for windows ending at each footprint interval is at most cm. Since the number of the footprint intervals is k, the time cost is then $O(ckm)$, hence the CKM algorithm.

We define the *compression factor* as $\frac{n}{k}$, which is the length of the trace over the number of intervals. The compression factor is minimal when $c = 1$ and $k = n$ and maximal when $c = m$ and $k = 1$. When c increases from 1 and m, the compression factor changes monotonically, as shown by the following lemma.

Lemma 3.2. *k is a non-increasing function of c.*

Proof Suppose e_i is the end position of footprint interval $I_i, i = 1, ..., k$. We will prove by induction on i that e_i will never move backward when c is increased to $c*$.

In the base case, I_1 starts at the beginning of the trace, and e_1 moves only forward in time when c is increased. Therefore, the start position of I_2 does not move backwards.

Suppose start position of $I_i, i \geq 2$ does not move backwards, and the next interval starting with the new start position and ending with previous e_i has a footprint of no more than c. As a result, the new e_i will only move forward in time when c is increased to c^*. ∎

3.4 Putting It All Together: The CKlogM Algorithm

We combine the relative precision NlogM algorithm and the trace compression CKM algorithm as follows. Instead of using previous-last-accesses in the CKM algorithm, we use the the relative precision footprint sizes described in Section 3.2. The approximation divides the past trace into $O(\log m)$ past partitions. The number of next-first-accesses is still c, and the number of intervals is k. The total cost becomes $O(ck \log m)$, hence the CKlogM algorithm.

3.5 Comparison with Previous Footprint Analysis

Direct counting Agarwal et al. counted the number of cold-start misses for all windows starting from the beginning of a trace [1]. In time-sharing systems, processes are switched at regular intervals. The cached data of one process may be evicted by data brought in by the next process. Thiebaut and Stone computed what is essentially the single-window footprint by dividing a trace by the fixed interval of CPU scheduling quantum and taking the average amount of data access of each quantum [26]. It became the standard model for studying multi-programmed systems in 1990s (e.g. [12]). These methods are simple and effective as far as only a linear number of windows are concerned.

If we fix the window size m, there are $n - m$ windows of this size. We call them sliding windows. A hardware counter can quickly count consecutive, non-overlapping windows. There are $\frac{n}{m}$ (assuming m divides n) non-overlapping windows, which is a subset of sliding windows. If we use the average of non-overlapping windows to estimate the average of sliding windows, we have a sampling rate $\frac{1}{m}$. Sampling at this low rate (for large m) may be accurate, but we cannot tell for sure unless we have all-window results to compare with.

Iterative generation Two recent methods by Suh et al. [25] and Chandra et al. [8] derived the footprint from an iterative relation involving the miss rate. Consider a random window w_t of size t being played out on some cache of infinite size. As we increase t, the footprint increases with every cache miss. Let $E[w_t]$ be the expected footprint of w_t, and $M(E[w_t])$ be the probability of a miss at the end of w_t. For window size $t + 1$, the footprint either increments by one or stays the same depending on whether $t + 1$ access is a cache miss, as shown by the following equation:

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

To be precise, the term $M(E[w_t])$ requires simulating sub-traces of all size t windows, which is impractical. Suh et al. simplified the relation into a differential equation and made the assumption of linear window growth when window sizes were close [25]. Chandra et al. computed the recursive relation bottom up [8]. Neither method places a bound on how the estimate may deviate from the actual footprint. In addition, their approach produces the average footprint, not the distribution.

The distribution is important. Considering two sets of footprints, A and B. One tenth of A has size $10N$ and the rest has size 0. All of B has size N. A and B have the same average footprint N but their different distribution can lead to very different types of cache interference.

Sampling and statistical inference Beyls and D'Hollander used hold-and-sample and reservior sampling to profile reuse time [5]. As part of a continuous program optimization framework, Cascaval et al. sampled TLB misses to approximate reuse distance [6]. Recently, Schuff et al. used sampling to measure reuse distance on shared cache [20]. These methods are based on sampling.

Two techniques by Berg and Hagersten (known as StatCache) [3] and by Shen et al. [22, 23] were used to infer cache miss rate from the distribution of reuse times. Let each access represented by a random variable X_i, which is assigned 1 if the access i is a miss and 0 otherwise. Let accesses i, j be two consecutive accesses to some data. The reuse time is $j - i$. Let $P(k)$ be the probability that a cache block is replaced after k misses. The following equation holds

$$E[X_j] = P(X_{i+1} + X_{i+2} + ... + X_{j-1})$$

With the goal of inferring number of cache misses in every interval, the analysis problem is one of all-window statistics. However, there can be $O(n)$ variables in the equation. To produce an estimate, Berg and Hagersten assumed constant miss rate over time and random cache replacement [3]. Shen et al. assumed a Bernulli process and LRU cache replacement [22, 23] (and later used a similar model to analyze multi-threaded code [16]). While both methods are shown to be accurate and effective for miss rate prediction, the accuracy of the all-window statistics has not been verified. One can easily construct two example executions that have arbitrarily different footprints while showing the same reuse times. The two methods do not guarantee a precision for the miss-rate estimate.

In a poster paper, Ding and Chilimbi described two solutions: linear sampling and all-window measurement [10]. The sampling rate of linear sampling is $1/n$, where n is the length of the trace, which can be too low to be statistically meaningful. The second technique is equivalent to the *NlogM* algorithm in this paper, which is too slow as we will show in Section 5.

4. Cache Interference Prediction and Ranking

Shared cache is a dynamic system. The interaction between active programs is likely non-linear since the cause and effect form a loop. The memory access of one program affects the performance of its peers, whose effect "feed back" to itself. In this section we first express this relation in an equation system and then present our solution.

Let programs 1 and 2 of infinite length run on shared cache. Let their speed be cpi_1, cpi_2 when running alone, as measured by cycles per instruction. When they execute together, let the change in their speeds be δ_1, δ_2. If we assume uniform speed change, an original time window of size t in an individual execution will take time $t\delta_1$ in program 1 and $t\delta_2$ in program 2. We call this effect *execution dilation* and the term δ_i the dilation factor of program i.

If we know execution dilation, we can compute its effect on cache sharing by matching for example each reuse window of size t in program 1 with a footprint window of size $t\frac{cpi_1\delta_1}{cpi_2\delta_2}$. From this matching, we can compute the relative increase in the number of capacity misses in shared cache.

$$x_1 = \frac{P\left[d_1 + f_2\left(t(d_1)\frac{cpi_1\delta_1}{cpi_2\delta_2}\right) \geq C\right]}{P[d_1 \geq C]} \tag{1}$$

where d_1 is a reuse distance of program 1, $t(d_1)$ is the size of reuse window of d_1 when program 1 is running alone, $f_2(x)$ is the footprint of program 2 in a window of size x, and c is the size of shared cache. The relative increase in shared-cache misses for program 2 is similarly computed as x_2.

To connect caching effect with execution time, we need a timing model. An accurate model is highly unlikely to exist given the complexity of modern computers. For example, the penalty of a cache miss on an out-of-order issue processor depends on the parallelism with surrounding instructions, the overlapping with other memory accesses, and the choice of hardware or software prefetching. Instead of looking for an accurate model, we use a simple model that can capture some first-order effect in the relation between cache misses and execution time.

We use a linear model, $T = T^n n + T^p m^p + T^s m^s$, where T is the execution time, n is the number of instructions, m^p is the number of private-cache misses, m^s is the number of shared-cache misses, and T^n, T^p, T^s are the average cost of an instruction, a private-cache miss, and a shared-cache miss. Of the 6 factors, n, m^p, m^s are trace specific, and T^n, T^p, T^s are machine specific. The values of T^n, T^p, and T^s are learned by linear regression given the linear model and a set of measured execution times and predicted L1 and L2 cache misses from SPEC2K benchmarks. The product of $T^n n$ can be viewed as the execution time when all data access are cache hits. The other two factors, $T^p m^p, T^s m^s$, are the penalty from cache misses. We make the simplifying assumption that all misses have the same penalty. A similar assumption was used by Marin and Mellor-Crummey [18].

The following equation combines the cache model and the time model to produce the dilation factor for program i ($i = 1, 2$)

$$\frac{T^n n_i + T^p m_i^p + T^s m_i^s x_i}{T^n n_i + T^p m_i^p + T^s m_i^s} = \delta_i \tag{2}$$

Substituting x_i with its equation, we have two recursive equations of two unknowns, δ_i. The formulation can be extended to p programs by changing the equation for x_i to

$$x_i = \frac{P\left[d_i + \Sigma_{j \neq i} f_j\left(t(d_i)\frac{cpi_i\delta_i}{cpi_j\delta_j}\right) \geq C\right]}{P[d_i \geq C]} \tag{3}$$

and generating p equations with p unknowns.

We solve the recursive equations using an iterative method. We first set δ_i to be 1, use the equations to compute the new value of δ_i', and repeat until all δ_i stops changing ($|\delta_i' - \delta_i| < \varepsilon\delta_i$).

To analyze the convergence property of the solution, consider the 2-program run case. We symbolize x_i in Equation 3 as a function $x_i(\frac{\delta_1}{\delta_2})$. Substituting this function into Equation 2 for δ_1, δ_2 and putting one over the other we have

$$\frac{\frac{T^n n_1 + T^p m_1^p + T^s m_1^s x_1\left(\frac{\delta_1}{\delta_2}\right)}{T^n n_1 + T^p m_1^p + T^s m_1^s}}{\frac{T^n n_2 + T^p m_2^p + T^s m_2^s x_2\left(\frac{\delta_1}{\delta_2}\right)}{T^n n_2 + T^p m_2^p + T^s m_2^s}} = \frac{\delta_1}{\delta_2} \tag{4}$$

If we represent the left-hand size as a function F, the equation becomes $F(\frac{\delta_1}{\delta_2}) = \frac{\delta_1}{\delta_2}$. Following theorem ensures the convergence of our iteration model.

Theorem 4.1. *$\exists \delta_1, \delta_2$, such that $\frac{\delta_1}{\delta_2} = F(\frac{\delta_1}{\delta_2})$.*

Proof Let $r = \frac{\delta_1}{\delta_2}$. According to the definition of F, we have the following 3 statements immediately.

- $y = F(r)$ is non-decreasing. Notice, when r increases $x_1(r)$ increases while $x_2(r)$ decreases.
- $F(0_+) = C1 > 0$, where C1 is constant.
- $F(\text{inf}) = C2 < \text{inf}$, where C2 is constant.

Let $r_1 = max\ S_1$, where $S1 = \{r | \forall s \leq r, F(s) \geq s\}$, $r_2 = min\ S_2$, where $S2 = \{r | \forall s < r, F(s) \geq s, F(r) \leq r\}$. Obviously $r_1 \leq r_2$. we claim $r_1 = r_2$.

Assume $r_1 \neq r_2$ by contradiction. According to the definition of r_1, $F(r_1) \geq r_1$, which means $F(r_1) \notin S_1$. According to the definition of r_2, $F(r_1) \in S_2$, which means $F(r_1) \geq r_2$. Because $F(r_2) \leq r_2$, we have $F(r_1) \geq F(r_2)$, which contradicts the assumption that $F(r)$ is non-decreasing. Since $r_1 = r_2$ by definition, we conclude $F(r_1) = r_1$. ∎

Interference is a complex relation. To run n programs on a machine with p processors, there are $\binom{n}{p}$ choices of co-execution. With the asymmetrical effect of cache sharing, every program in every set may have a different interference. Our model, which we call, *PAW* (Profiling of All-Window footprints) model or *PAW* prediction, can be used to predict the interference and rank co-run choices so that a user can choose program combinations that have least interference. *PAW* statistics, locality and footprint, are machine independent. To make prediction, *PAW* requires two items of information from the target machine: the sequential running time of each program and the size of the shared cache. The process does not need any parallel testing.

5. Evaluation

5.1 Experimental Setup

We have tested 14 SPEC2K benchmarks, which include all programs that we could compile and run. Other programs have compilation errors due to either our configuration setup or the benchmark code. Each of the successful tests is compiled using a modified GCC 4.1 compiler with "-O3" flag. The compiler is modified to instrument each memory access and generate the data access trace when the program runs. The reuse distance analysis is the relative-accuracy approximation algorithm [28]. The precision is set to 90%. Since profiling is machine-dependent, we use a Linux cluster for profiling different benchmarks in parallel to save time. Each node has two 3.2GHz 64-bit Intel Xeon processors with 1MB L2 cache each and 6GB physical memory total. We use a set of newer machines for cache-interference tests. Each has a 2.0GHz Intel Core Duo processor with two cores sharing 2MB L2 cache and 2GB memory.

5.2 All-window Footprint

Figure 4 shows the full distribution of all-window footprint in 4 of the 15 benchmarks. While every program has a different looking footprint, we choose these 4 because they demonstrate the range of variations. Each graph shows window size, measured in number of data accesses with exponential scale, in the x-axis and footprint, measured in bytes, in the y-axis. It's worth to note, since our measurements are based on 64-byte blocks the footprint sizes are always multiples of 64. At each window size, each graph shows five footprint numbers: the minimal size `min`, the maximal size `max`, the `median`, and the two footprints whose size is higher than `90%` and `10%` of footprints. We connect the points into a line, so each distribution is drawn with five curves. In probability terms, a point (x, y) on the n% curve indicates that a window of x data accesses has n% chance to have a footprint smaller than y.

The footprint distribution shows highly summarized information about program data usage. For example, every point $< x, y >$ in the `min` curve shows the minimal footprint in all windows of size x is y. As a sanity check, a reader can verify that the minimal and the maximal footprints are monotone functions of the window size. The monotonicity should hold in theory but in an observation, it is not guaranteed if we sample a set of windows rather than measure all windows.

All-window footprint shows the characteristics of the working sets in a program. *Gzip* has constant-size median working sets for windows from 1 million to 256 million accesses, shown by Figure 4(a). In compression or decompression, the same data buffers are repeatedly used for each chunk of the input or output file. Constant-size working sets are a result of complete data reuse. *Bzip2* creates larger auxiliary data structures in order to improve compression ratio over a larger scope. Its working sets increase in size with the length of the execution window. The rate of increase, however, gradually decreases, as shown by Figure 4(b).

Shown in Figure 4(c), `Equake` has the clear streaming access pattern because its working set size increases linearly with the execution window length. In fact, it shows the most rapid growth, resulting in the largest footprint among all programs we have tested. 90% of footprints increase from 8KB to 8MB when window size increases from 30K to 32M data accesses. Its footprint interferes with many other programs when running in shared cache. `Twolf` in Figure 4(d) shows the smallest variation in footprint size. 80% of windows have nearly the same size footprint. These examples show that applications have different characteristics in their footprint distribution.

5.3 Comparison with Miss Count

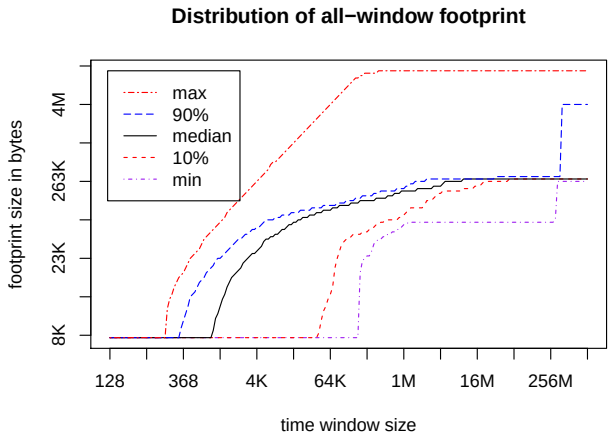
It is interesting to consider the relation between the number of misses in a window and the footprint of the window. On average, the number of misses is the miss rate times the window size. Therefore, the average miss count grows linearly. It is unbounded if a program executes forever. The footprint is bounded by the size of program data. The growth cannot be always linear.

This difference is shown in the example of *Gzip* result in Figure 5. The miss count should be in the unit of 64-byte blocks. The figure shows it in bytes growing linearly from 2^{-9} byte to 2^{26} (83MB or 1.3 million misses) for window sizes ranging from 1 access to 18 billion. For the same window sizes, the footprint grows from 64 bytes (1 block) to 83MB. This difference leads to different predictions of cache interference. We compare the accuracy of these predictions next.

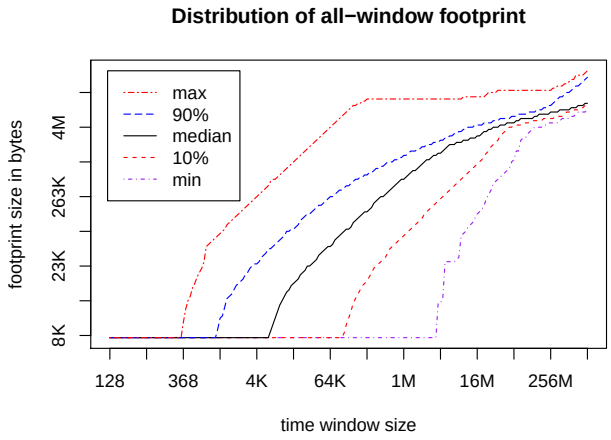
5.4 Prediction of Program Interference

For interference we run our test set of 15 programs on a dual-core machine mentioned earlier. There are $\binom{15}{2} = 105$ ways of pairing two of the 15 programs to share cache, 455 ways of choosing 3, and 1365 ways of choosing 4. We use all-window profiling on each of 15 programs in a sequential run and use the composable model to rank cache interference of co-run choices. The ranking is based on the predicted slowdown (the geometric mean in each group).

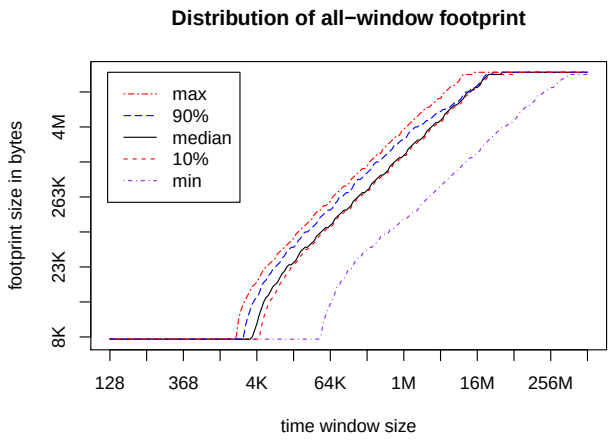
There are two questions about the prediction. The first is how accurate the model is in predicting the cache interference. It can be evaluated by measuring the change in the miss rate when programs are run together. The second is how useful the model is in predicting the performance of shared cache. It can be evaluated by measuring the change in the actual execution time. We show re-



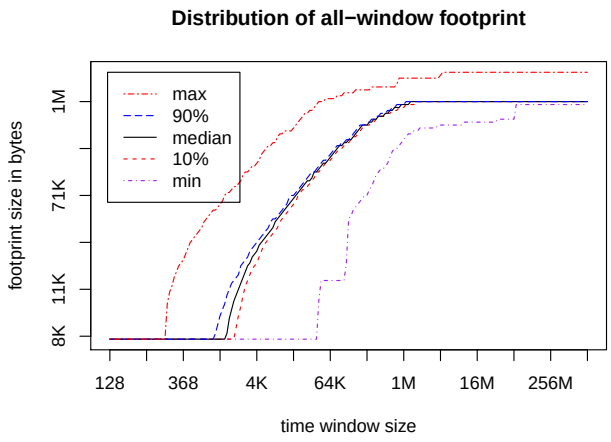
(a) **Gzip** has constant-size median working sets for windows from 1 million to 256 million accesses.



(b) **Bzip2** has a larger working set than *Gzip* for windows larger than 256 thousand accesses.



(c) **Equake** has a streaming access pattern. The footprint increases linearly with the window size.



(d) **Twolf** shows a narrow distribution —80% windows of the same length have nearly the same footprint.

Figure 4. The distribution of all-window footprint shows the characteristics of the working set in 4 SPEC benchmarks.

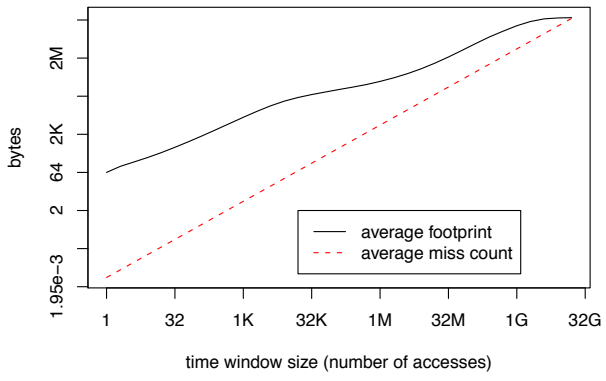
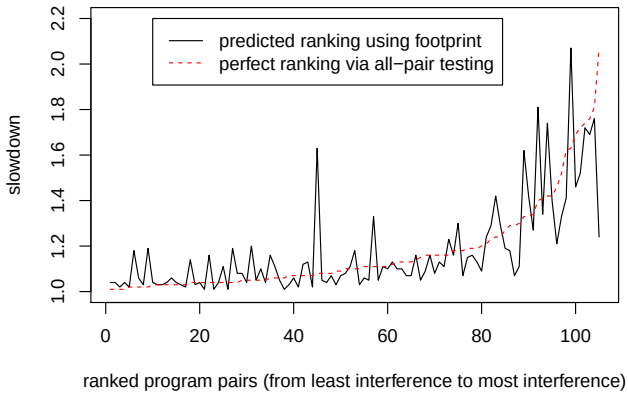


Figure 5. Comparison of linear miss count growth and non-linear footprint growth in *Gzip*.

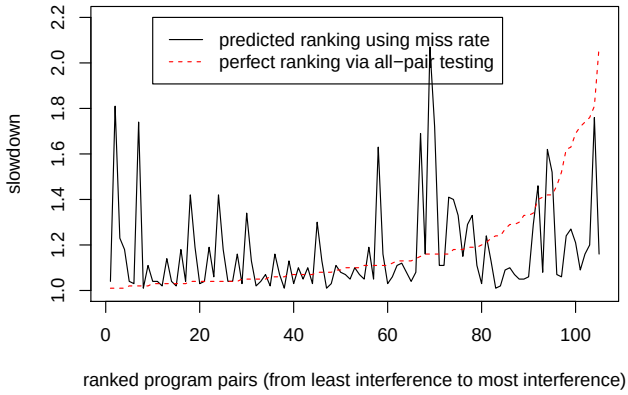
sults that answer the second question. To measure the interference, we run a pair of programs long enough and take the average slow down, following the method used in [27].

We evaluate the prediction by plotting an *interference-ranking graph*. Figure 6(a) shows two sets of results. The first is the actual slowdown of the program pairs ordered by the footprint-based prediction. The second is the slowdown when program pairs are ordered by results from all-pair testing. The second one is a monotone curve, which is the ideal result for a prediction method. The data points of the footprint-based prediction are connected into a line. It is not monotone, but it is generally the case that program pairs with a higher rank have a greater slowdown than program pairs with a lower rank. The method ranks high interference pairs better than it ranks low interference pairs. This can be attributed to the simple execution-time model. When the cache effect becomes significant, the footprint model shows its strength. In practical use, users are more concerned with large degradations.

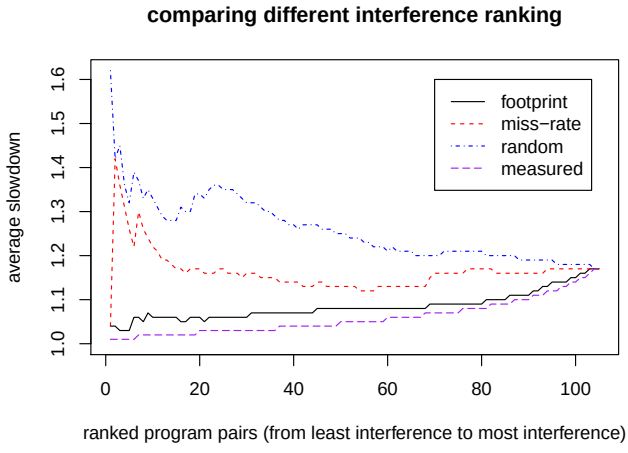
Figure 6(b) shows the effect of miss-rate based predictor in comparison with the ideal result. The prediction is obtained by



(a) Footprint-based interference ranking compared to exhaustive testing. The y-axis shows the slowdown (quadratic mean) by the x^{th} program pair.



(b) Miss-rate based interference ranking compared to exhaustive testing. The y-axis shows the slowdown (quadratic mean) by the x^{th} pair.



(c) Cumulative mean interference by 4 ranking methods. The y-axis shows the average slowdown (quadratic mean) for the first x pairs.

Figure 6. All-window footprint predicts the performance effect of all-pair cache sharing better than the miss rate does.

using the miss count instead of the footprint. The miss count is computed from the average miss rate, which is computed using the reuse distance in the sequential run. Miss rate may significantly underestimate the cache footprint. As an example, the pair, `art` and `twolf`, has the second lowest total miss rate (miss per second) in their sequential run, but when running together, `art` causes `twolf` to run 91% slower while itself actually runs 6% faster (we do not have an explanation for the speedup). Overall, the high-interference pairs (on the right half of the graph) do not show a rising trend, indicating the ranking fails to predict in co-run slowdown.

Figure 6(c) shows the cumulative mean slowdown of groups 1 to x at each point. The ideal result, based on the ranking from exhaustive measurement, increases from 1% slowly until near the end when it quickly rises to 17%. This is because most programs do not interfere significantly but for the few that do, they interfere badly. The mean slowdown for the first 85 pairs is less than 10%, but the next 20 pairs raise the average of all 105 pairs to 17%.

Footprint-based ranking shows a similar result to the ideal ranking. The cumulative mean slowdown is 6% and 3% respectively for the prediction and the ideal for the first 25 pairs and 8% and 5% for the first 50 pairs. The two curves move closer after the 62th pair. The difference in their average slowdown is no more than 2%. If we take the first 25 pairs, the cumulative slowdown in miss-rate based ranking is 17%, which is near 3 times the 6% cumulative slowdown by footprint. As a basis for comparison, Figure 6(c) also shows the effect of random ranking, which does not consider the effect of cache interference. Random ranking is obtained by ranking co-run pairs according to the lexicographic order of their program names.

5.5 Efficiency

Measurement speed We have tested 26 reference inputs of the 14 benchmark programs that we could compile. For brevity, Table 1 shows results for one reference input (the first one in the test order) of each benchmark. The programs are listed by the increasing order of their SPEC benchmark number, from 164 for *gzip* to 300 for *twolf*.

prog.	N (10^9)	M (10^3)	K (10^6)	N/K (10^3)	CKlogM Refs/sec (10^6)
gzip	24	232	7.2	3.4	1.9
vpr	41	159	6.9	5.9	2.9
gcc	16	360	2.3	7.3	3.5
mesa	31	28	1.8	17.5	8.4
art	30	11	12	2.4	1.7
mcf	14	315	27	0.50	0.3
equake	108	167	7.6	14.2	6.5
crafty	41	7.5	25	1.7	1.3
ammp	4.7	51	2.6	1.8	1.1
parser	78	102	21	3.7	1.9
gap	68	787	3.6	19.0	7.3
vortex	31	196	3.0	10.4	4.9
bzip2	42	530	7.8	5.4	2.4
twolf	106	12	48	2.2	1.8
median	36	162	7.4	4.6	**2.1**
mean	45	211	12.5	6.8	**3.3**

Table 1. Measurement speed by CKlogM ($C = 128$) for the reference input of 14 SPEC2K benchmarks. The speed closely correlates with the average interval length (N/K). The average speed is 3.3 million references per second.

The size of the execution is measured by the length of the data access trace and the number of data cache lines the program

touched, shown in the second and third columns of the table. The size of each data cache line is 64-byte. The number of intervals, K, and the average length of each interval, N/K, are shown next. Finally, the last column shows the measurement speed in the unit of million references per second.

The length of the traces range from 4 billion to 108 billion, the size of data ranges from 7 thousand to 787 thousand, the number of intervals ranges from 1.8 million to 48 billion, the average interval length ranges from 504 to 19 thousand, and the measurement speed ranges from 264 thousand data accesses per second to 8.4 million accesses per second. If we measure in terms of the number of windows measured, the speed ranges from 10^{15} (a quadrillion) windows per second to nearly 10^{18} (a quintillion) windows per second.

The speed closely correlates with the average interval length. The average length of k means that in a random time window, 128 distinct data elements are accessed by each k accesses. When the length is 500, the measurement speed is less than 300 thousand accesses per second. However, when the length is 19,000, the speed becomes 7.3 million accesses per second. The average length is determined by the length of the trace and the number of intervals. Hence the cost of the algorithm is determined by the number of intervals, K, more than any other factor.

Comparison between CKlogM and NlogM The NlogM algorithm was the first to attempt complete all-window measurement [10]. Here we compare NlogM with CKlogM for $C = 128$ and $C = 256$. Table 2 shows the result of 12 SPEC2K benchmarks (that the NlogM method could finish analyzing), including the length of the trace, the size of data, the measurement time by NlogM, and the speedup numbers by CKlogM for two Cs. We have to use test inputs because it takes too long for NlogM to measure larger inputs. The measurement time ranges from 414 seconds for *mcf* to 4 hours for *parser* by NlogM and from 3 and 2 seconds for *twolf* to 18 and 17 minutes for *ammp* by CKlogM when $C = 128$ and $C = 256$. The largest reduction is a factor of 316 for *twolf* from 10 minutes to 2 seconds. The smallest reduction is a factor of 8 for *mcf* from 7 minutes to 53 seconds.

prog.	N	M	NlogM time [sec]	CKlogM C=128 time (speedup)	CKlogM C=256 time (speedup)
gzip	804M	9K	12K	328(35)	246(47)
vpr	298M	5K	4K	84(49)	41(90)
gcc	255M	15K	4K	37(102)	19(198)
mesa	173M	25K	2.6K	10(259)	9(288)
art	1.0B	5K	12K	119(104)	108(114)
mcf	40M	5K	414	52(7.8)	41(10)
equake	342M	40K	5K	42(126)	33(161)
crafty	935M	8K	15K	739(20)	187(78)
ammp	818M	51K	13K	1129(12)	1036(13)
parser	929M	24K	14K	142(101)	98(147)
gap	277M	147K	5K	30(168)	20(252)
vortex	2087M	65K	–	537(N/A)	283(N/A)
bzip2	3029M	60K	–	660(N/A)	565(N/A)
twolf	76M	309	631	3(210)	2(316)
median	320M	12K	5178	47(**101**)	41(**131**)
mean	497M	28K	7343	201(**100**)	153(**142**)

Table 2. Time comparison for the test input of 14 SPEC 2K benchmarks. On average (excluding `Vortex` and `Bzip2` for which *NlogM* does not finish), the measurement time is reduced from 2 hours per program to 73 and 52 seconds when $C = 128$ and 256 respectively.

The average statistics is shown at the bottom of Table 2. On average across 12 benchmarks, the length of the trace is half billion memory accesses and the size of data is 28 thousand words. The average measurement time of NlogM is two hours. The average reduction by CKlogM is a factor of 100 to 73 seconds when $C = 128$ and a factor of 142 to 52 seconds when $C = 256$. In other words, *CKlogM reduces the average measurement time from two hours to one minute.*

5.6 Relation between c and Measurement Speedup

We define the *speedup factor* by $\frac{N}{CK}$. Intuitively, a larger c produces a greater speedup. However, this is not true. We have measured all power-of-two c values for all our tests and studied the relation between c and the potential speedup. When the speedup numbers for different c were drawn as a curve, we have observed variations in terms of whether a curve has a single peak or whether the shape changes when a program is run with different inputs. Figure 7 show the speedup factor for 4 programs whose footprint we have shown in Figure 4. The highest speedup is in tens of thousands for `bzip2` and `gzip`, one hundred thousand for `equake`, and (too large to shown in the figure) 1.6 million for `twolf`.

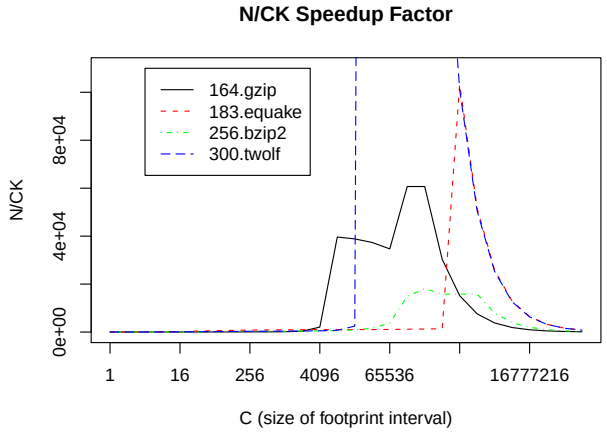


Figure 7. $\frac{N}{CK}$ gives the speedup factors for 4 SPEC CPU2000 benchmarks. The speedup factors for `twolf` are as high as 1.6 million but are not shown above 100K in order to make other numbers easy to see.

6. Potential Uses of All-window Profiling

The new algorithm and the accurate modeling of cache sharing may have a number of uses:

- *footprint-based program optimization.* Traditionally, the way to improve memory performance is to improve program locality. For shared cache, we may reduce the interference by reducing the size of program footprint even if the locality stays unchanged. All-window footprint can aid footprint-oriented techniques in two ways. The first is to identify "hot" spots in a program and help program tuning. For example, a tool can identify loops and functions in a program that have the largest footprint or a footprint larger than 90% of the windows of the same size. Second, all-window analysis can be used to fully evaluate the effect of a footprint-oriented program transformation.
- *sampling vs. all-window statistics.* Statistics such as the miss rate may differ depending on two factors, which windows are chosen to measure the miss rate, and how large the windows

are. All-window analysis can be used to determine whether the miss rate is uniform as a function of window sizes - whether the miss rate in small windows is the same as the miss rate in large windows. Similarly, it can determine whether the miss behavior in a subset of windows is similar to the miss behavior in all windows. In addition, On-line techniques may use the miss rate to approximate the footprint. Accurate footprint results can be used to calibrate approximation methods and determine their effectiveness and limitations.

- *cache vs. other resources.* Cache sharing is only one factor in performance. Others include the sharing of memory interface and the CPU scheduler [30]. An accurate estimate of cache interference can help us determine how important cache sharing is when compared against other factors, and how useful and portable a cache model may be in performance prediction.
- *working set characterization.* Working set is an important metric in workload characterization. It helps to have the full distribution of working-set size in all windows not just the average. With the distribution, we can evaluate statistically how good the average is as an estimate for an execution window.
- *batch scheduling.* The composable model may help ahead-of-time co-run scheduling. If a workload has a fixed number of program-input pairs, a batch scheduler can use the type of ranking results shown in Section 5.4 to choose which sub-group of tasks to start on the next available multi-core computer.

7. Related Work

We have discussed in Section 3.5 related work in footprint analysis. Next we review related techniques in modeling shared cache performance.

Cache sharing Chandra et al. modeled cache sharing between independent applications and gave the framework that we use in this paper [8]. Cache sharing in multi-threaded code is affected by two additional factors: data sharing and thread interleaving. The effect can be characterized by extending the concept of reuse distance [20, 21] or inferred using a composable model [11, 16]. Previous methods did not measure footprint precisely and did not model the circular interaction. The techniques in this paper measure the all-window footprints with a guaranteed precision and use an iterative algorithm to compute the circular interference among co-run applications.

Zhang et al. used a conflict graph to show the asymmetrical effect of cache sharing [27]. They did not explain the causes, which we model as the combined effect of footprint and locality.

Program co-scheduling Many techniques have been developed for selecting applications to run on shared cache. Snavely and Tullsen devised a scheduler that generates and evaluates a set of random schedules in the sampling phase and picks the best one for the remaining execution [24]. Fedorova et al. suspended program execution when needed to ensure a set of programs have equal share of cache [13]. The technique is based on the assumption that if two programs have the same frequency of cache misses, they have the same amount of data in cache. The two techniques are dynamic and do not need off-line profiling. However, on-line analysis may not be accurate and cannot predict interference in other program combinations.

Jiang et al. formulated the problem of optimal co-scheduling and showed that although the problem is NP-hard, an approximation algorithm can yield a near-optimal solution [15]. As inputs, their solution requires accurate prediction of co-run degradation. They gave a prediction method based on statistical inference of application footprint [16]. As discussed in Section 3.5, the result from statistical inference does not have a precision guarantee. The all-window analysis and the composable model can provide the needed prediction, with an accuracy similar to exhaustive testing (as shown in Section 5.4).

Zhuravlev et al. defined a metric called *Pain* to estimate the performance degradation due to cache sharing [30]. The degree of pain that application A suffers while it co-runs with B is affected by A's cache sensitivity, which is proportional to the expectation of its reuse distance distribution, and B's cache intensity, which is measured by the number of last level cache accesses per million instructions. The pain of A-B co-scheduling is estimated as the sum of the pains of A and B. Pain is a composable model. It differs from *PAW* in the notion of cache intensity. It is defined using a single window size. In computing B's interference of A, *PAW* considers all window sizes in B since a reuse window in A may have an arbitrary length.

Trace and static analysis Measuring footprint requires counting distinct data elements. The problem has long been studied in measuring various types of stack distances. A review of measurement techniques for reuse distance (LRU stack distance) can be found in [28]. Kaplan et al. studied the problem of optimal trace compression–to generate the shortest possible trace that still preserves same LRU behavior in large cache [17]. Our footprint measurement is similar in that both methods use intervals as the basic unit in analysis. More recently, Zhou studied random cache replacement policy and gave a one-pass deterministic trace-analysis algorithm to compute the average miss rate (instead of simulating many times and taking the average) [29]. Reuse distance can be analyzed statically for scientific code [4, 7] and recently MATLAB code [9]. The static techniques also compute the footprint in loops and functions. Unlike profiling whose results are usually input specific, static analysis can identify and model the effect of program parameters. These techniques are concerned with only reuse windows and cannot measure the footprint in all execution windows, which is the problem we have solved in this paper.

8. Summary

We have presented the first accurate all-window footprint analysis. The new algorithm uses footprint counting, relative precision approximation, and trace compression to reduce the asymptotic cost to $O(CKlogM)$. We have successfully measured all-window footprints in the SPEC2K benchmark suite in its full-length executions. It is the first time such complete measurement is made. The results show that applications differ in their footprints in terms of not just the size but also the relation with time and the compactness of the distribution. The previous fastest accurate method could only measure these programs on their test input, for which the new algorithm reduces the profiling time from 2 hours to 1 minute. The all-window analysis measures a quadrillion to nearly a quintillion footprints per second.

We have also presented an iterative algorithm to compute the non-linear, asymmetrical effect of cache sharing and developed a tool for ranking program co-run choices without parallel testing. The ranking is close to that from exhaustive parallel testing, reducing performance slowdown by as much as a factor of 3 compared to miss-rate based ranking.

Acknowledgments

We would like to thank Xiao Zhang for help with the experimental setup. The presentation has been improved by the suggestions from Xipeng Shen, Santosh Pande, and the systems group at University of Rochester especially Kai Shen. Xiaoya Xiang and Bin Bao are supported by two IBM Center for Advanced Studies Fellowships.

The research is also supported by the National Science Foundation (Contract No. CCF-0963759, CNS-0834566, CNS-0720796).

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, 1975.

[3] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[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. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, St. Louis, MO, 2005.

[7] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *International Conference on Supercomputing*, pages 150–159, 2003.

[8] 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*, pages 340–351, 2005.

[9] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *International Conference on Supercomputing*, pages 295–304, 2010.

[10] C. Ding and T. Chilimbi. All-window profiling of concurrent executions. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008. *poster paper*.

[11] 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.

[12] B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1):104–130, 1997.

[13] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2007.

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

[15] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 220–229, 2008.

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

[17] 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.

[18] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, 2004.

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

[20] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2010.

[21] M. Schulz, B. S. White, S. A. McKee, H.-H. S. Lee, and J. Jeitner. Owl: next generation system monitoring. In *Proceedings of the ACM Conference on Computing Frontiers*, pages 116–124, 2005.

[22] X. Shen and J. Shaw. Scalable implementation of efficient locality approximation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 202–216, 2008.

[23] 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.

[24] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[25] 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.

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

[27] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *Proceedings of the EuroSys Conference*, 2009.

[28] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6):1–39, Aug. 2009.

[29] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, pages 144–154, 2010. Springer Lecture Notes in Computer Science No. 6289.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–142, 2010.