Cache Performance Analysis of Traversals and Random Accesses

Richard E. Ladner[*] James D. Fix[†] Anthony LaMarca[‡]

To appear in the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms

Abstract

This paper describes a model for studying the cache performance of algorithms in a direct-mapped cache. Using this model, we analyze the cache performance of several commonly occurring memory access patterns: (i) sequential and random memory traversals, (ii) systems of random accesses, and (iii) combinations of each. For each of these, we give exact expressions for the number of cache misses per memory access in our model. We illustrate the application of these analyses by determining the cache performance of two algorithms: the traversal of a binary search tree and the counting of items in a large array. Trace driven cache simulations validate that our analyses accurately predict cache performance.

1 Introduction

The concrete analysis of algorithms has a long and rich history. It has played an important role in understanding the performance of algorithms in practice. Traditional concrete analysis of algorithms is interested in approximating as closely as possible the number of "costly steps" an algorithm takes. For example, in a sorting algorithm, because comparisons are fairly costly, the traditional measure of its performance is the number of comparisons it takes to sort. In an arithmetic problem, such as performing the Fourier Transform, a traditional measure is the number of arithmetic operations the algorithm takes.

This paper proposes and investigates another arena for concrete analysis of algorithms. We call this new area *cache performance analysis of algorithms* because we are interested in approximating as closely as possible the number of *cache misses* an algorithm incurs. In modern processors cache misses are very costly steps, in some cases costing 50 processor cycles or more [10]. In the tradition of concrete analysis of algorithms, cache misses are genuinely costly, and it is appropriate to analyze them.

The key application of cache performance analysis is towards the cache conscious design of data structures and algorithms. In our previous work we studied the cache conscious design of priority queues[13] and sorting algorithms[14], and were able to make significant performance improvements over traditional implementations by considering cache effects.

The goal of the work in this paper is to assemble a useful set of analysis tools for understanding the cache performance of algorithms. From the cache's perspective, an executing algorithm is simply a sequence of memory accesses. Unfortunately, an algorithm's memory access pattern can be quite complicated and there is no way to analyze arbitrary access patterns. Accordingly, we model algorithms as a combination of simple access patterns whose cache performance characteristics can be quantified. In this paper we focus on two basic access patterns, a *traversal* of a contiguous segment of memory and a *system of random accesses*.

We consider two types of traversals: a simple *scan traversal* that traverses memory sequentially and a *permutation traversal* that visits every block in the contiguous segment a fixed number of times. Both traversals are common patterns in algorithms, the former occurring in many array-based algorithms and the latter in traversals of dynamic data structures where the order of the data is independent of the memory layout. For permutation traversals, we give a simple expression for the expected number of cache misses per access in theorem 5.1, assuming the permutation order is chosen uniformly at random. In practice, the expression is a good predictor of the cache performance of an arbitrary permutation traversal.

A system of random accesses is defined by a set of random processes that access memory in a stochastic, memoryless fashion. In this paper we analyze systems of random accesses running for a finite number of steps. This contrasts our previous paper [13] in which we only analyzed the performance of a system of random accesses running for an infinite period. We apply the analysis to obtain theorem 7.2, which describes the cache performance of such a system interacting with a scan traversal. Such an access pattern arises

[*]Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, ladner@cs.washington.edu

[†]Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, fix@cs.washington.edu

[‡]Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, lamarca@parc.xerox.com

when modeling basic algorithms such as radix sort, heap sort, rehashing, Huffman coding, and parsing. Quantifying the interaction between the two access patterns is important, as it yields better predictions than that obtained from analyzing each access pattern independently.

Finally, we give example applications of our analyses to understanding the cache performance of tree traversals and frequency counting. Using a trace driven cache simulation we show that our analyses accurately predict the cache performance of each.

2 Related Work

Memory hierarchies have been studied from a variety of perspectives for a long time. A number of authors have established models of memory hierarchies then designed and analyzed algorithms for these models [2, 3, 4, 6]. In addition, there is a long history of external memory algorithms, for example, for sorting and searching [8, 15, 12, 21, 22, 23]. One characteristic of much of this previous work is that in addition to performing calculations, the algorithms can explicitly move data to and from the cache. While this makes for an interesting, powerful programming model, modern machines do not typically provide any abilities beyond simple prefetching for moving data between memory levels. Rather, we assume that items are only brought into the cache when they are accessed and are only removed from the cache when they are evicted by another conflicting access. This paper focuses on direct-mapped caches, a cache design commonly found in modern machines. In direct-mapped caches, the data item brought into the cache can only go to one place. The result is that our analyses are both different and likely more useful in practice than cache analyses for fully associative caches [17].

There have been several studies [1, 17, 19, 20] that have tried to quantify the cache performance of programs by summarizing or analyzing actual memory access traces. Our work differs from this in that we do not examine the trace of a program, but just the actual algorithm itself. A number of studies have analyzed different memory hierarchy alternatives by modeling program or multiprogram access patterns by a simple stochastic model, the Independent Reference Model (IRM) [5, 16, 11]. Rather than apply such a model to entire programs, we apply and extend the IRM cache analysis of Rao [16] to algorithm subcomponents that exhibit random access behavior.

3 The Model

We choose a simple model of cache memory for our analysis. We assume there is a large memory that is divided up into M *blocks* and a smaller cache that is divided up into C blocks. Memory is indexed 0 to $M-1$ and the cache is indexed 0 to $C-1$. Although $C \leq M$, we assume that C is fairly large, at least in the hundreds and more likely in the tens of thousands. In this study we only examine *direct-mapped caches*. In a direct-mapped cache each block of memory maps to exactly one block in the cache. We assume the simple mapping where memory block x is mapped to cache block $x \bmod C$. At any moment of time each block y in the cache is associated with exactly one block of memory x such that $y = x \bmod C$. In this case we say that block x of memory is in the cache. An access to memory block x is a *hit* if x is in the cache and is a *miss*, otherwise. As a consequence of a miss the accessed block x is brought into the cache and the previous block residing at cache location $x \bmod C$ is evicted.

We model an algorithm as a sequence of accesses to blocks in memory. We assume that initially, none of the blocks to be accessed are in the cache. In reality an algorithm reads and writes to variables where a variable is stored as part of a block or as multiple blocks. A read or write to a variable that is part of a block is modeled as one access to the block. A read or write to a variable that occupies multiple blocks is modeled as a sequence of accesses to the blocks. We do not distinguish between reads and writes because we assume a *copy back* architecture with a *write buffer* for the cache [9]. In the copy back architecture writes to the cache are not immediately passed to the memory. A write to cache block $y = x \bmod C$ is written to memory block x when a miss occurs, that is, when block z is accessed where $y = z \bmod C$ and $z \neq x$. A write buffer allows the writes to location x to propagate to memory asynchronously with high probability.

The cache performance of an algorithm is measured by the number of misses it incurs. Naturally, an algorithm's overall performance is not just a function of the number of misses, but also a function of the number of memory accesses, the runtime penalty incurred due to each cache miss, and the cost of other operations. In this paper we concentrate on just cache misses because it is an important factor in overall performance.

4 Empirical Validation Methodology

For the two example applications– tree traversal and frequency counting– we have performed experiments to validate our analyses. These involved measuring the number of misses incurred by an actual implementation of each on a simulated cache. To do so, we implemented each application in C on a DEC Alpha, and monitored the memory accesses made using Atom [18]. Atom is a utility for instrumenting program executables. In particular, one can insert cache simulation code at each read and write to memory. By running the instrumented program, we are able to measure the number of memory accesses made and the misses that would occur for various cache configurations.

One benefit of performing the experiments, other than validating the analyses, is that we were able to verify that the patterns we analyze contribute most of the cache misses in our example applications. Though we do not model accesses to local variables and to the program stack, the experiments demonstrated that these had minimal impact on the cache miss count. One caveat of interpreting our experimental results is that they are derived from a cache simulation: For example, they do not measure misses due to context switches, or effects due to virtual to physical address mapping.

5 Traversals

One of the simplest memory access patterns is a *traversal with block access rate K*. A traversal with block access rate K accesses each block of a contiguous array of N/K blocks exactly K times each (we always assume that K divides N). Hence, there are a total of N accesses in a traversal. We consider two different traversals, a *scan traversal* and a *permutation traversal*. A scan traversal starts with the first block, accesses it K times, then goes to the next block accessing it K times, and so on to the last block. Scan traversals are extremely common in algorithms that manipulate arrays. If B array elements fit in a block then a left-to-right traversal of the array is a scan traversal with block access rate B.

The scan traversal is easy to analyze. Every K accesses is a cache miss when a new block is brought into the cache, yielding $1/K$ cache misses per access.

PROPOSITION 5.1. *A scan traversal with block access rate K has $1/K$ cache misses per access.*

5.1 Permutation Traversals

A permutation traversal starts by accessing a memory block chosen uniformly at random. At any point in the permutation traversal, if there are k accesses remaining and memory block x has j accesses remaining, then memory block x is chosen for the next access with probability j/k. Another way to think of a permutation traversal is as a permutation of a multiset chosen uniformly at random. Consider the multiset S that contains exactly K copies of x where $0 \leq x < N/K$. Let $\sigma = \sigma_1\sigma_2 \cdots \sigma_N$ be a permutation of S. If $\sigma_i = x$ then the i-th access in the permutation traversal is to block x. Permutation traversals are also very common in algorithms. For example, suppose a linked list is constructed where the nodes were allocated randomly and B nodes fill a block. A traversal of the linked list is essentially a permutation traversal with block access rate B.

The analysis of permutation traversal, summarized in the following theorem, is more complicated.

THEOREM 5.1. *Assuming all permutations are equally likely, a permutation traversal with block access rate K of N/K contiguous memory blocks has $1/K$ misses per access if $N \leq KC$ and*

$$1 - \frac{(K-1)C}{N} \tag{5.1}$$

expected cache misses per access if $N > KC$.

Proof. Let us fix a particular cache block x. Let $m_1, m_2, \ldots, m_n$ be the memory blocks that map to cache block x in the region accessed by the traversal. Depending on the value of x, n will be $\lceil N/CK \rceil$ or $\lfloor N/CK \rfloor$. During the permutation traversal, nK accesses will be made to x. To determine the expected number of misses incurred by accesses to x during the traversal, we need only consider the stream of accesses to memory mapped to x.

For $1 \leq i \leq nK$, define a random variable $B_i = j$ whenever the i-th access that maps to x is to location m_j. For $1 \leq i \leq nK$ and $1 \leq j \leq n$, let $X_{ij} \in \{0,1\}$ be a random variable that indicates whether the i-th access that maps to x is a hit to location m_j. The first access to x is always a miss, so $X_{1j} = 0$ for all j. For $i > 1$ we have the following:

$$\mathrm{E}[X_{ij}] = \Pr[B_{i-1} = j \text{ and } B_i = j] = \frac{K(K-1)}{nK(nK-1)}$$

For a traversal, the expected number of hits at x is then

$$\sum_{i=1}^{nK}\sum_{j=1}^{n} \mathrm{E}[X_{ij}] = n(nK-1)\frac{K(K-1)}{nK(nK-1)} = K-1.$$

Note that this value is independent of the choice of x.

The expected number of hits incurred by the traversal for all cache blocks is just the sum of the expected contribution for each cache block. For $N \leq KC$, the number of cache blocks to consider is N/K, so the total expected hits is $(K-1)N/K$ and the expected hits *per access* is $(K-1)/K$. For $N > KC$, there are C cache blocks to consider, so the total expected hits is $(K-1)C$ and the expected hits per access is $(K-1)C/N$. Considering the misses per access gives us the statement of the theorem. ∎

5.2 Example: Tree Traversals

To illustrate our result for permutation traversals, we apply it to predicting the cache performance of a preorder traversal of an random binary search tree.

The binary search tree implementation we consider is pointer-based. Each node of the tree consists of a key value, a data pointer, and pointers to its left and right children. It is common in memory efficient implementations of data structures for all the data of a certain type to be allocated from the same region of memory, so we assume further that the nodes of the tree are allocated contiguously in memory. The keys of the nodes, hence the structure of the tree, are determined independently of the node allocation order. In a recursive implementation of a preorder traversal, we start at the root node, examine its key, then recursively visit the nodes of the left subtree, and then the nodes of the right subtree. Assume that exactly L tree nodes fit in a cache block. A preorder traversal, then, is a permutation traversal with $K = 3L$.

Note that even if the tree is arbitrary, the permutation traversal that arises from its preorder traversal is not completely arbitrary. When the key of a node is visited, the next access will always be to the left child pointer. Furthermore, the right child pointer will be accessed next for the majority of nodes (the leaves), or may be accessed soon after. For this reason, we model the accesses to the keys as a permutation traversal with $K = L$, and the remaining accesses to the child pointers as hits. If n is the number of tree nodes and C is the cache size in blocks, theorem 5.1 gives the total number of misses to be n/L when $n < CL$ and $n - (L-1)C$ otherwise.

To validate this analysis, we implemented preorder traversal and measured its cache performance using the methodology described in section 4. In this implementation, a tree node consisted of a key, data pointer, and left and right child pointers, totaling 32 bytes. The simulated cache was a megabyte in size with 64 byte blocks, so $L = 2$ and $C = 2^{14} = 16,384$. Figure 1 shows the misses per tree node measured for a preorder traversal of a randomly generated tree for varying sizes n. Note that our model very closely predicts the number of misses, even though the model does not consider misses due to accessing the right child pointer, accessing data on the execution stack, or accessing other auxiliary data, as their number is not significant.

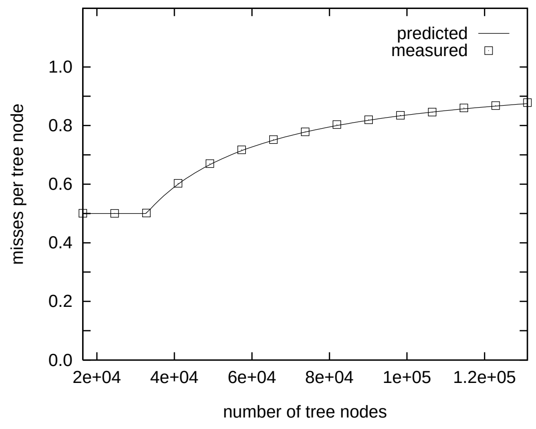

Figure 1: Comparison of the measured cache performance of a preorder tree traversal with that predicted by our model.

6 Random Access

Another simple access pattern in memory is *random access.* In a random access pattern each block x of memory is accessed statistically, that is, on a given access block x is accessed with some probability. In most algorithms that exhibit random access, the probability that x is accessed on a given access depends on the history of past accesses. However, often a random access pattern can be analyzed approximately by assuming the *independent reference assumption* [7]. In this model each access is independent of all previous accesses.

In our previous paper [13] we called the analysis of a set of random access patterns *collective analysis* and in this paper we adopt the collective analysis approach in order to examine algorithms that exhibit random access. We begin by introducing a *system of random accesses.* In a system of random accesses the cache is partitioned into a set R of regions and the accesses are partitioned into a set P of processes. In practice, the processes are used to model accesses to different portions of memory that map to the same portion of the cache. For example, if we have two data items that conflict in the cache, the accesses of each would correspond to two distinct processes.

We let λ_{ij} be the probability that region $i \in R$ is accessed by process $j \in P$. Let r_i be the size of region i in blocks, so that $C = \sum_{i \in R} r_i$. Furthermore, let λ_i be the probability that region i is accessed, that is, $\lambda_i = \sum_{j \in P} \lambda_{ij}$. The following proposition is a restatement of a theorem in our previous paper [13] which is an extension of a theorem of Rao [16].

Proposition 6.1. *In a system of random accesses, in the limit as the number of accesses goes to infinity, the expected number of misses per access is*

$$1 - \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2.$$

It is helpful to define the quantities

$$\eta_i = \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2,$$

the probability that an access is a hit in region i, and

$$\eta = \sum_{i \in R} \eta_i,$$

the probability that an access is a hit. The probability that an access is a miss is simply $1 - \eta$.

To illustrate application of collective analysis we consider the cache performance of a simple algorithm that can be modeled as random access. The algorithm uses two arrays: one that is the size of the cache, and the other one-third the size of the cache. The algorithm loops forever where on each iteration it chooses two random elements of the first array and stores their sum in a randomly chosen element of the second array.

To apply collective analysis to this algorithmm we divide the cache into two regions: Region 0 is one-third the size of the cache and holds blocks from the first one-third of the first array and all of the second array; Region 1 is two-thirds the size of the cache and holds the remaining blocks of the first array. We model the accesses to the arrays by two processes: Process 0 performs the reads from the first array and process 1 performs the writes to the second array. Note that process 0 accesses memory twice as frequently as process 1. The per-region access parameters are

$$\lambda_{00} = \frac{2}{9}, \lambda_{01} = \frac{1}{3}, \lambda_{10} = \frac{4}{9}, \lambda_{11} = 0$$

$$\lambda_0 = \frac{5}{9}, \lambda_1 = \frac{4}{9}$$

Note that this model is an approximation to the actual access pattern of the algorithm as the read and write process accesses are deterministically interleaved— we know that every third access is to the second array, for example. However, this model can yield a good approximation for the hit rate. From proposition 6.1, the predicted number of hits per access for this example are

$$\eta = \frac{9}{5}\left(\left(\frac{2}{9}\right)^2 + \left(\frac{1}{3}\right)^2\right) + \frac{9}{4}\left(\frac{4}{9}\right)^2 = \frac{11}{15}.$$

This yields an expected miss ratio of about 26.7%, which is within .2% of the miss ratio measured from an implementation of this algorithm using the methodology of section 4. We will see another more straightforward use of collective analysis in section 7.3.

6.1 Random Access for a Finite Period

Proposition 6.1 gives the expected miss ratio if we think of a system of random accesses running forever. However, in some cases we are interested in the number of misses that occur in N accesses, rather than in the long run. This case arises naturally when we know that every N accesses "flushes" the cache. We have the following lemma that is a key to all the results that follow:

LEMMA 6.1. *In a system of random accesses, for each block in region i, the expected number of misses in N accesses is*

$$(6.2) \quad \rho_i = \frac{\lambda_i - \eta_i}{r_i} N + \frac{\eta_i}{\lambda_i}\left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^N\right).$$

Proof. Let x be a particular block in region i. Let ρ_{ik} be the probability that the k-th access is a miss at block x. This is just the product of the probability that x is accessed and the probability that an access to x is a miss:

$$\begin{aligned}\rho_{ik} \quad = \quad & \Pr[k\text{-th access is } x] \\ & \cdot \Pr[k\text{-th access is miss} | k\text{-th access is } x].\end{aligned}$$

The first term is just $\frac{\lambda_i}{r_i}$ since an access to any of the r_i blocks in region i are equally likely. To address the second term, we consider instead the conditional probability that the k-th access was a hit to x given that it was an access to x, that is, the *hit ratio* of x at access k. Let q_{ik} be this hit ratio. We have

$$\rho_{ik} = \frac{\lambda_i}{r_i}(1 - q_{ik}).$$

A hit occurs at the k-th access whenever x has been accessed before and the most recent access to x was made by the same process as the k-th access. Thus, we have

$$\begin{aligned} q_{ik} &= \left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^{k-1}\right) \sum_j \left(\frac{\lambda_{ij}}{\lambda_i}\right)^2 \\ &= \left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^{k-1}\right) \frac{\eta_i}{\lambda_i}. \end{aligned}$$

where the first term in the product is the probability that x has been accessed before the k-th access, and the second term is the probability the most recent access and the k-th access were made by the same process, given that they were accesses to x.

To compute ρ_i, note that ρ_{ik} is also the expectation that the k-th access is a miss at x. Since the value of ρ_{ik} is independent of the choice of x, ρ_i is just the sum of ρ_{ik} over all accesses k:

$$\begin{aligned} \rho_i &= \sum_{k=1}^{N} \rho_{ik} \\ &= \frac{\lambda_i - \eta_i}{r_i} N + \frac{\eta_i}{r_i} \sum_{k=1}^{N} \left(1 - \frac{\lambda_i}{r_i}\right)^{k-1} \end{aligned}$$

$$= \frac{\lambda_i - \eta_i}{r_i} N + \frac{\eta_i}{\lambda_i}\left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^N\right).$$

∎

From this lemma it is easily seen that the expected number of misses in all the N accesses is $\sum_{i \in R} r_i \rho_i$. Hence we have the theorem

Theorem 6.1. *In a system of random accesses, the expected number of misses per access in N accesses is*

$$(6.3) \quad 1 - \eta + \frac{1}{N}\sum_{i \in R} \frac{r_i \eta_i}{\lambda_i}\left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^N\right).$$

As N goes to infinity the expected number of misses per access goes to $1 - \eta$, the expected miss ratio from proposition 6.1. If there is only one process and one region then $\eta = 1$. As a consequence the expected number of misses per access simplifies to

$$\frac{C}{N}\left(1 - \left(1 - \frac{1}{C}\right)^N\right) \approx \frac{C}{N}\left(1 - e^{-N/C}\right).$$

7 Interaction of a Scan Traversal with a System of Random Accesses

In this section we apply the results of the previous sections to derive a formula for the expected number of misses per access when a scan traversal interacts with a system of random accesses.

Suppose we have a system of accesses that consists of a scan traversal with block access rate K to some segment of memory interleaved with a system of random accesses to another segment of memory that makes L accesses per traversal access. What we mean by this is that the accesses to memory have the pattern of one traversal access followed by exactly L random accesses, followed by one traversal access followed by exactly L random accesses, repeated for as long as the algorithm continues. In addition, the traversal accesses a block exactly K times before moving to the next block. If you like, the pattern of access is described by the regular expression $(t_1 r^L t_2 r^L ... t_K r^L)^*$ where a sequence $t_1 t_2 ... t_K$ indicates K accesses to the same block and r represents a random access. We assume that the system of random accesses has regions R and processes P and the probability that process j accesses region i is λ_{ij}. Region i has r_i blocks.

7.1 Scan Traversal with Access Rate 1

It is helpful to begin by considering the case when $K = 1$. In this case we are analyzing the access pattern described by the regular expression $(tr^L)^*$ where t indicates a traversal access and r indicates a random access. Assume that the total number of accesses is N where $(1+L)C$ divides N. There are exactly $N/(1+L)$ traversal misses. Consider a block x in region i. Every C traversal accesses the traversal captures the block x, that is, the traversal accesses a block in memory that maps to cache block x. During the next $C-1$ traversal accesses, a random access might be made to the block that was evicted from x by the traversal. By lemma 6.1 the expected number of misses per block of region i in the random accesses during C traversal accesses is

$$\rho_i = \frac{\lambda_i - \eta_i}{r_i} LC + \frac{\eta_i}{\lambda_i}\left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^{LC}\right).$$

The expected number of misses, both traversal and random accesses, during C traversal accesses is

$$C + \sum_{i \in R} r_i \rho_i.$$

This gives us the expected number of misses per access described in the following theorem.

Theorem 7.1. *In a system consisting of a scan traversal with access rate 1 and system of random accesses with L accesses per traversal access, the expected number of misses per access is*

$$(7.4) \quad \frac{1 + (1-\eta)L + \frac{1}{C}\sum_{i \in R} \frac{r_i \eta_i}{\lambda_i}\left(1 - \left(1 - \frac{\lambda_i}{r_i}\right)^{LC}\right)}{1 + L}.$$

It is interesting to set the parameters to some specific values to see the results. Assume there is one region of size C and two processes where each is equally likely to access a given block. In this case $r_1 = C$, $\lambda_1 = 1$, and $\eta = \eta_1 = 1/2$. For large size C the formula (7.4) evaluates to approximately

$$(7.5) \quad \frac{3 + L - e^{-L}}{2(1+L)}.$$

For $L = 1$ formula (7.5) evaluates to approximately .91 misses per access. As L grows the number of misses per access approaches .5 which is what one would expect with the system of random accesses without any interaction with a traversal.

7.2 Scan Traversal with Arbitrary Access Rate

In the numerator of expression (7.4) there are three terms corresponding to the three sources of misses. The first term, 1, accounts for the misses caused by the scan traversal itself. The second term, $(1-\eta)L$, accounts for the misses caused by the system of random accesses by itself. The third term, $\frac{1}{C}\sum_{i \in R} \frac{r_i \eta_i}{\lambda_i}(1 - (1 - \frac{\lambda_i}{r_i})^{LC})$, accounts for the misses caused by random access after the traversal has captured the block accessed. When $K > 1$ there is an additional source of cache misses. Consider two consecutive accesses by the scan traversal to the same block x. Between the two accesses are L random accesses. There is some chance that one of the L random accesses is to a block that maps to the same block in the cache as x does. If that happens there are two misses. First, the random access is a miss because the block belongs to the traversal and second, the next traversal access to block x is a miss.

For simplicity we assume that $K(1+L)C$ divides the total number of accesses N. In this case a fourth term in the numerator arises that quantitatively provides the expected number of misses caused during the $K-1$ traversal accesses to single blocks during the traversal. In most cases the fourth term will be negligible, but if L is large, it can be significant.

THEOREM 7.2. *In a system consisting of a scan traversal with access rate K and a system of random accesses with L accesses per traversal access, the expected misses per access is $S/K(L+1)$ where S is the sum of the following terms*

$$1 \tag{7.6}$$

$$(1-\eta)KL \tag{7.7}$$

$$\frac{1}{C}\sum_{i\in R}\frac{r_i\eta_i}{\lambda_i}\left(1-\left(1-\frac{\lambda_i}{r_i}\right)^{KLC-(K-1)L}\right) \tag{7.8}$$

$$\frac{K-1}{C}\sum_{i\in R}r_i\left(\frac{\eta_i}{\lambda_i}+1\right)\left(1-\left(1-\frac{\lambda_i}{r_i}\right)^{L}\right). \tag{7.9}$$

Proof. Let x be a specific block in region i of the cache. In every KC traversal accesses there are $(K-1)$ that access a block that maps to x (after the first access to the block). Hence, in that period there are $KC-(K-1)$ accesses that do not map to x at all or are the first access to a block that maps to x. During this period of the $KC-(K-1)$ accesses there are $KLC-(K-1)L$ random access by the system of random accesses. By lemma 6.1 there are

$$\begin{aligned}\rho_i &= \frac{\lambda_i-\eta_i}{r_i}(KLC-(K-1)L)\\ &\quad+\frac{\eta_i}{\lambda_i}\left(1-\left(1-\frac{\lambda_i}{r_i}\right)^{KLC-(K-1)L}\right).\end{aligned}$$

expected misses by the system of random accesses to memory the map to block x. The remaining $K-1$ traversal accesses map to block x. Between each of these accesses are L random accesses. Again by lemma 6.1 there are

$$\pi_i=\frac{\lambda_i-\eta_i}{r_i}L+\frac{\eta_i}{\lambda_i}\left(1-\left(1-\frac{\lambda_i}{r_i}\right)^{L}\right).$$

expected misses by the system of random accesses to memory that map to block x. In addition, with probability $\theta_i = 1-(1-\frac{\lambda_i}{r_i})^L$ there is an additional miss in the traversal caused by a random access that maps to x. Thus, for every KC traversal accesses there are $K(1+L)C$ total accesses and

$$C+\sum_{i\in R}r_i(\rho_i+(K-1)(\pi_i+\theta_i))$$

expected misses. This immediately yields the theorem. ∎

7.3 Example: Frequency Counting

A common algorithm component is frequency counting: given an array, determine the number of times each item appears in the array. For example, it is needed in Huffman and other encoding schemes, and is employed in efficient implementations of radix sort. In this section we apply the formula of theorem 7.2 to determine the cache performance of frequency counting.

Assume we have an array E whose values are elements of $\{0,1,\ldots,n-1\}$. To obtain a frequency count of the values, we have an array F of size n. We traverse the array E and increment the frequencies in F accordingly. If we view the value of each element of E as being chosen randomly with some probability, then frequency counting involves a scan traversal of array E in combination with random accesses to update F.

To demonstrate the analysis we assume that each value in $\{0,1,\ldots,n-1\}$ is equally likely to be in each position of the array E. In addition, let us assume that exactly B array entries of E fit into a block and that E requires storage at least twice the size of the cache. We assume that the frequency array F also has B frequencies per block, hence it uses n/B blocks. It is handy to define k and d such that $n/B = Ck+d$ where $0 \le d < C$.

Note that the traversal has access rate B and there is exactly 1 random access per traversal access, hence, referring to theorem 7.2, we have $K = B$ and $L = 1$. To specify the set of random processes associated with frequency counting, there are two regions, the first of size d and the second of size $C-d$. In the first region there are $k+1$ processes where each process j, $0 \le j \le k$, in region 1 represents the accesses to memory blocks Cj to $Cj+d-1$. Similarly, in the second region there are k processes were each process j, $0 \le j < k$, in region 2 represents the accesses to memory blocks $Cj+d$ to $C(j+1)-1$. In the case that $n/B \ge C$ the parameters of the system of random accesses are: $\lambda_{1j} = d/(n/B)$ for $0 \le j \le k$ and $\lambda_{2j} = (C-d)/(n/B)$ for $0 \le j < k$.

Hence we can calculate:

$$\begin{array}{rclrcl} r_1 &=& d & r_2 &=& C-d \\ \lambda_1 &=& (k+1)d/(n/B) & \lambda_2 &=& k(C-d)/(n/B) \\ \eta_1 &=& d/(n/B) & \eta_2 &=& (C-d)/(n/B) \\ \eta &=& C/(n/B). &&& \end{array}$$

In the case that $d=0$ the first region is actually empty. If $n/B < C$ then there is only one region of size d and only one process so that $\eta = 1$.

Using the values of the parameters determined above we can use theorem 7.2 to predict the cache performance of an implementation of frequency counting. The misses per access can be expressed as the sum of four terms $T_n + R_n + I_n + I'_n$, corresponding to the terms of $S/K(L+1)$ in the theorem. The first term T_n from expression 7.6 is the number of misses per access for the traversal acting alone.

$$T_n = \tfrac{1}{2B}.$$

The second term R_n from expression 7.7 gives the expected number of misses per access for the system of random accesses acting alone.

$$R_n = \begin{cases} 0 & \text{if } n/B < C \\ \frac{1}{2}(1 - \frac{C}{n/B}) & \text{if } n/B \geq C. \end{cases}$$

The third term I_n is the expected number of misses per access caused by the traversal periodically capturing each block in the cache. From expression 7.8, these misses per access are

$$I_n = \frac{n/B}{2BC}\left[1 - \left(1 - \frac{1}{n/B}\right)^{BC-(B-1)}\right]$$

when the frequency array fits in the cache and are approximately

$$I_n \approx \frac{C}{2n}\left[1 - \left(1 - \frac{1}{C}\right)^{BC-(B-1)}\right]$$

otherwise. The second case is exact for $d=0$, when the frequency array size is a multiple of the cache size. The contribution of I_n is maximized when $n/B = C$ and is negligible when n is large or small. The fourth term I'_n from expression 7.9 is the expected number of misses caused by the random accesses to a block that is mapped to one that is being traversed.

$$I'_n = \begin{cases} \frac{(B-1)}{BC} & \text{if } n/B < C \\ \frac{B-1}{2BC}(1 + \frac{C}{n/B}) & \text{if } n/B \geq C. \end{cases}$$

The term I'_n never exceeds $1/C$; if C is large then it is effectively zero.

To see the effect of these terms we plotted T_n, $T_n + R_n$, and $T_n + R_n + I_n + I'_n$ for realistic specific values of B and C. The left half of figure 2 shows these curves for $B = 4$ and $C = 2^{15} = 32,768$. The traversal misses per access are $T_n = 1/2B = 1/8$ for all n. Because C is so large I'_n is insignificant for all n. When n is small the total number of cache misses per access is near $1/8$ because $R_n = 0$ and I_n is near 0. When the frequency array is about the size of the cache, $n/B = C$, I_n is approximately $1/2B = 1/8$, and so the total misses per access are nearly $1/4$. When n is large, R_n approaches $1/2$ and I_n approaches zero, so the misses per access approach $5/8$.

To verify our analysis, we experimentally measured the cache performance of an implementation of frequency counting. For various values of n, the implementation generates a large array of elements chosen uniformly at random from $\{0, \ldots, n-1\}$, then counts their occurrence in an array of size n. We measured the cache misses and total accesses of performing frequency counting, once again in a cache simulator using Atom. In the simulation, both arrays held eight byte integers, and the simulated cache was one megabyte with 32 byte blocks. This gives us a cache with $C = 32,768$ blocks where each block holds $B = 4$ eight byte integers, matching the plotted parameters above. The right half of Figure 2 shows how closely the measured cache performance matches the performance predicted by the formula. For all frequency array sizes that we considered, the match was within .1%.

8 Conclusions

In this paper we introduced a framework for the analysis of the cache performance of algorithms as a legitimate study in the tradition of the concrete analysis of algorithms. We analyzed several common memory access patterns including traversals, random accesses, and traversals with random accesses combined. We experimentally validated the analytic formulas using trace driven cache simulation.

Naturally, there are many other memory access patterns that arise in algorithms, and new techniques need to be devised to analyze them. Extending this work to determine the cache performance of other memory access patterns will hopefully aid algorithm implementors in understanding the impact of a design choice, primarily in cases where memory access performance is an issue. An interesting challenge in extending this work is in determining the number of cache misses that occur when two or more access patterns occur in conjunction, that is, when the accesses from each pattern are interleaved in some way. Unlike traditional operation count analysis, the number of cache misses incurred by two

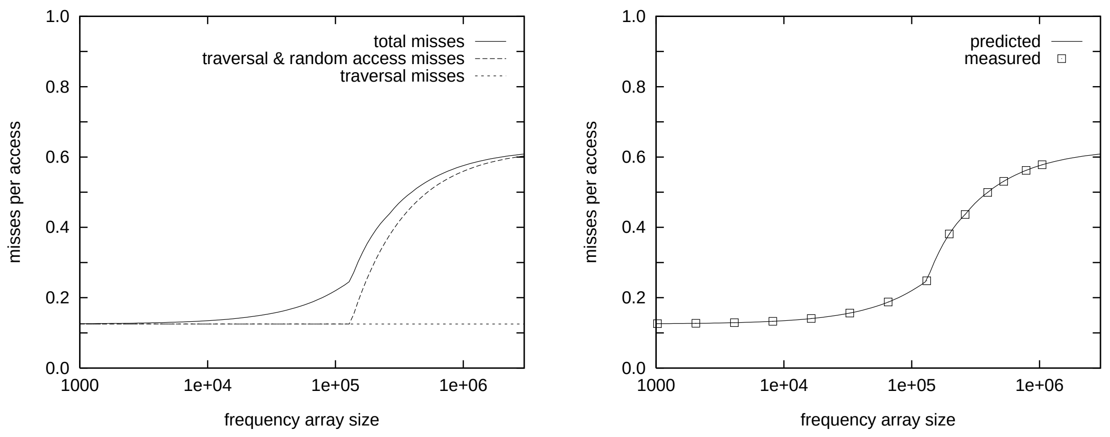

Figure 2: The cache performance of frequency counting. Left: a graph of the predicted misses per access showing the contributions of the terms T_n, R_n, and $I_n + I'_n$. Right: a comparison of the predicted performance with that measured from simulation.

interleaved access patterns is not necessarily the sum of their individual miss counts. The analysis of a scan traversal combined with random access is one example of this, and only full analysis allowed us to understand which additional misses caused by their interaction were a large contribution and which were not.

This study focuses on direct-mapped caches that fetch data upon access, however there are other kinds of caches that are used in modern architectures. Some processors support prefetching and nonblocking loads. With these features, a data item might be brought into the cache before it is needed, essentially reducing the number of misses incurred by an algorithm. To achieve this, prefetching often requires reference prediction by either the hardware or software. For scan traversals, this is straightforward, but it can be difficult for the random access patterns studied here. However, further study and understanding of the cache performance of memory access patterns could benefit prefetching analyses.

Some caches are k-way set-associative, where a given block of memory can map to any one of k blocks in the cache (k is typically 2 to 4). While the cache performance of traversals and uniformly random access patterns do not benefit from set-associativity [11], set associativity tends to reduce the cache misses per access for other access patterns and in practice [9]. It is our hope that this paper will encourage other researchers to begin studying the cache performance of algorithms with interesting memory access patterns and to study them in realistic cache architectures including direct-mapped caches and set-associative caches with a low degree of associativity.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[2] A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[3] A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *19th Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[4] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, 1971.

[6] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[7] E. Coffman and P. Denning. *Operating Systems Theory*. Prentice–Hall, Englewood Cliffs, NJ, 1973.

[8] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal of Computing*, 10(3):422–433, Aug 1981.

[9] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach, Second Edition*. Morgan Kaufman Publishers, Inc., San Mateo, CA, 1996.

[10] G. Herdeg. Design and implementation of the AlphaServer 4100 CPU and memory architecture. *Digital Technical Journal*, 8(4):48–60, 1996.

[11] W. F. King. Analysis of paging algorithms. In *IFIP Congress*, pages 485–490, August 1971.

[12] D. E. Knuth. *The Art of Computer Programming, vol III – Sorting and Searching*. Addison–Wesley, Reading, MA, 1973.

[13] A. LaMarca and R. E. Ladner. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, Vol 1, Article 4, 1996.

[14] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of*

the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pages 370–379, 1997.

[15] D. Naor, C. Martel, and N. Matloff. Performance of priority queue structures in a virtual memory environment. *Computer Journal*, 34(5):428–437, Oct 1991.

[16] G. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25(3):378–395, 1978.

[17] J.P. Singh, H.S. Stone, and D.F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.

[18] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, 1994.

[19] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

[20] O. Temam, C. Fricker, and W. Jalby. Influence of cross-interfernces on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, 1995.

[21] A. Verkamo. External quicksort. *Performance Evaluation*, 8(4):271–288, Aug 1988.

[22] A. Verkamo. Performance comparison of distributive and mergesort as external sorting algorithms. *The Journal of Systems and Software*, 10(3):187–200, Oct 1989.

[23] L. Wegner and J. Teuhola. The external heapsort. *The Journal of Systems and Software*, 15(7):917–925, Jul 1989.