

The Set-Associative Cache Performance of Search Trees*

James D. Fix[†]

Abstract

We consider the costs of access to data stored in search trees assuming that those memory accesses are managed with a cache. Our cache memory model is two-level, has a small degree of set-associativity, and uses LRU replacement, and we consider the number of cache misses that a set of accesses incurs. For standard tree access— searches and traversals— changing the degree of set-associativity has no effect on performance.

To explain this, we develop general stochastic access models, an adaptation of the independent reference model (IRM), and analyze the expected number of cache hits and misses incurred by these types of access. The models and analyses are accurate: we are able to exactly predict the cache performance of tree data structures. In addition, we prove why set-associativity is of little or no benefit for these types of memory access and give examples where direct-mapping performs better than set-associativity.

1 Introduction

Memory caches are crucial to the performance of today’s computer architectures. They are key in keeping the speed of accesses to data in memory competitive with the speed of calculations in the processor. As a result, an algorithm’s performance can be highly dependent on the hardware cache’s performance. Much recent algorithm work has focused on designing algorithms and data structures that use the cache effectively [14, 5, 9, 22, 20, 25, 26, 1, 34, 27, 7], developing models for cache memory [29, 4, 3, 2], analyzing the cache performance of existing algorithms [19], and adapting external memory algorithms for cache memory (see [31, 32, 33], e.g.). When programming for performance, thinking about the cache matters.

Algorithmic studies of caches often assume an unrealistic cache model, that of a fully associative cache, with some exceptions [29, 28, 27]. Instead, hardware caches are either direct-mapped or have a limited degree of set-associativity. Analyzing the performance of

set-associative caches can be more difficult since one has to take into account the layout of data structures in memory.

The conventional wisdom is that increasing the degree of set-associativity yields better performance (with decreasing returns) for typical cache workloads [17]. The implicit assumption in this heuristic argument is that the cache’s replacement policy is able to perform well for the given pattern of access, or at least better than that of a direct-mapped cache. For some memory access sequences this is not the case: In some cases the mapping of memory to cache blocks provides an excellent replacement policy implicitly, so that a direct-mapped cache performs as well as or better than a cache with some set-associativity using LRU replacement. For example, the accesses to a number of tree-based data structures like binary search trees and implicit heaps have a cache performance relatively unaffected by the degree of set-associativity.

1.1 Example: Search Tree Traversal One data structure access pattern we consider is traversal. A preorder traversal of a binary tree, for example, visits each node of the tree in an order determined by the tree’s structure. Even though a traversal visits each tree node’s data exactly once, there can be some benefit to having a cache. Figure 1 shows the misses per tree node measured for a preorder traversal of a randomly generated tree for varying sizes, recorded in caches of varying degrees of set-associativity but fixed size. There is little change in the cache performance when we vary the degree of set-associativity. In Section 3 we give a theorem that explains this performance.

1.2 Example: Search Tree Access Figure 1 shows the cache performance of a sequence of random searches in an ordered dictionary implemented as a binary search tree. To generate the data, random binary trees were constructed for various data set sizes, and then a random sequence of key lookups were made. For the resulting memory accesses made by each lookup sequence, we simulated and counted the hits that for a direct-mapped cache and D -way set-associative caches for $D = 2, 4, 8$, all of the same size. The graph shows

*A portion of this research was supported by NSF grant CCR-9732828.

[†]Department of Mathematics, Reed College, Portland, OR, 97202, USA, jimfix@reed.edu

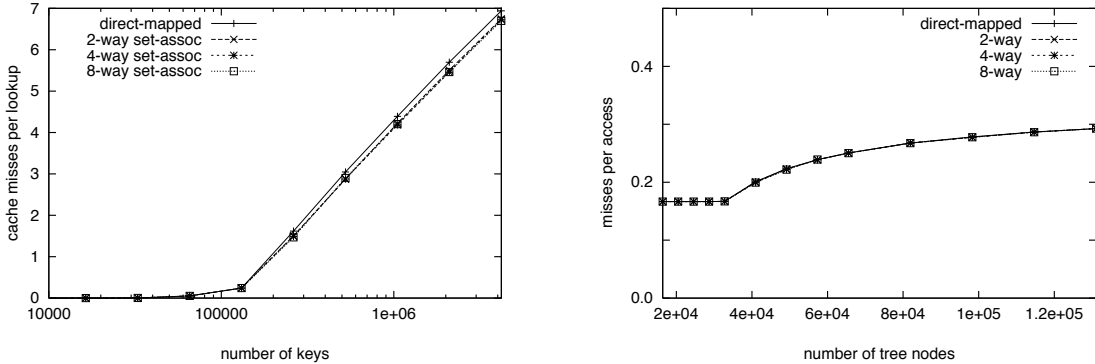


Figure 1: The performance of a random sequence of binary search tree lookups (*left*) and preorder tree traversal (*right*) in direct-mapped, two-way, four-way, and eight-way set-associative caches.

the number of misses per search in each cache for each tree size. For data set sizes where the resulting binary tree is larger than the cache, there is an improvement in the number of cache hits incurred by the set-associative caches over the direct-mapped cache. The key thing to note is that the improvement is *very slight*—no greater than 1.6% for the data sizes we considered. In addition, there is very little variation in performance among the set-associative caches for this data set. In Sections 4 and 5.1 we analyze a general class of access patterns that model this cache behavior.

1.3 Paper Overview In this paper we look at the access patterns of the above examples, provide stochastic models to predict their performance, and develop analyses that explain when set-associative caches provide no benefit in general. The work is an extension of the direct-mapped results of [19]. In addition, we adapt analysis of the independent reference model (IRM) [11, 24, 18, 15, 16], bounding the performance of systems of access in the IRM to characterize the memory layouts of data structures that do not benefit from set-associative caches.

2 Methodology

2.1 Cache Model We model the main computer memory as a sequence of stored values, addressed from 0 to $M-1$. A cache is a smaller memory that sits between the CPU and main memory, that serves as faster, redundant storage for the main memory. In current computing systems the cache is mostly transparent to computer algorithms. Algorithms cannot explicitly place values into the cache, this is determined by the caching hardware by its mapping and replacement policy.

In our model and in current systems, cache memory

and main memory are organized into distinct *blocks*. A block is a contiguous sequence of values; the number of which, B , is fixed by the architecture. Let C be the capacity of the cache in blocks. When the CPU requests to read a value stored at address i , it is handled by the cache. If the value at address i is in the cache, then the request is a *hit*, and the cache is able to return the value at location i without consulting main memory. If the value at i is not in the cache then the request is a *miss* and the cache requests block $x = \lfloor i/B \rfloor$ from main memory. Main memory responds with the contents of the block containing i —the values stored at addresses $xB, \dots, xB + B - 1$.

We are interested in the cache performance of algorithms. With this in mind, we model an algorithm as a sequence of read and write requests to and from memory, in particular, as a stream of blocks, numbered from 0 to $M/B - 1$, that contain the target locations of an algorithm’s memory requests. For any algorithm, we consider only its sequence of requests and count the hits and misses due to those requests. We do not differentiate between reads and writes.

In a D -way set-associative cache, a memory block can be stored in any of D designated blocks in the cache. D is the degree of set-associativity of the cache and is typically less than eight. The cache blocks are partitioned into C/D cache sets numbered from 0 to $C/D - 1$, where each set has D cache blocks. When a requested memory block x is brought into the cache, it gets stored in one of the D blocks in set $\mathcal{S}(x)$, the set mapping. Let cache set $s = \mathcal{S}(x)$. We say that x maps to s and write $x \rightarrow s$. Unless otherwise stated, the set mapping function is $\mathcal{S}(x) = x \bmod C/D$. We assume the cache uses a least recently used (LRU) replacement policy: when x is placed in the cache, of the D blocks in set s , the cache block holding the least

recently requested memory block is evicted and replaced by memory block x .

A *direct-mapped cache* is a set-associative cache with $D = 1$. In direct-mapped caches the mapping of memory blocks to the cache solely determines eviction: when the contents of block x are placed in the cache as a result of a miss, the unique memory block y with $\mathcal{S}(x) = \mathcal{S}(y)$ is evicted from the cache.

2.2 Empirical Methodology To measure the cache performance of algorithms, we implemented each in C on a Compaq (formerly DEC) Alpha workstation, and monitored the memory accesses made using Atom [30]. 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 were able to measure the number of memory accesses made and the misses that would occur for various cache configurations. The results are from simulation—for example, they do not measure misses due to context switches, or effects due to the virtual to physical address mapping. Note, however, that the memory streams are not artificial, they are the actual virtual address streams of the code being executed.

3 Modeling Tree Traversals

The cache misses involved in a traversal of an array are easy to analyze. Each array value is accessed exactly once, in the sequential order of its layout in memory. Assuming that the cache is initially empty, there is a miss for each memory block the array occupies. The performance of the traversal of a link-based data structure like a linked list or a tree is similar, but more complicated: Each node in the data structure is accessed exactly once, but the order of node traversal is determined by a layout in memory which is typically not sequential. We model the traversal of link-based data structures as *permutation traversal*.

A permutation traversal with access rate K accesses each of N/K blocks in a contiguous region of memory K times. The sequence of these N accesses is random. The possible block access sequences of a permutation traversal are a permutation of a multiset. Consider the multiset S that contains exactly K copies of block 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 . In a permutation traversal the sequence σ is chosen uniformly at random from all possible permutations of S . Figure 2 illustrates a permutation traversal of an array, where $B = 4$ values fit in a cache block, the access rate is $K = 4$, and $C = D = 1$.

We are able to analyze the expected number of misses incurred by a permutation traversal when the cache is initially empty. The following theorem, an extension of the direct-mapped case analyzed in [19], gives the expected number of misses per access made by a permutation traversal with access rate K :

THEOREM 3.1. *For any $D \geq 1$, in a D -way set-associative cache with C blocks that uses LRU replacement, 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*

$$(3.1) \quad 1 - \frac{(K-1)C}{N}$$

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

In the proof of Theorem 3.1, left for the full paper, there is a tradeoff when we compare a D -way set-associative cache with a direct-mapped one, as given by the following lemma [13]:

LEMMA 3.1. *For each cache set s in a D -way set-associative cache, the expected number of hits to s made by a permutation traversal is*

$$\eta_s = \begin{cases} n(K-1) & \text{for } n < D \\ D(K-1) & \text{otherwise,} \end{cases}$$

where n is the number of memory blocks that map to set s and K is the block access rate.

3.1 Tree Traversals as Permutation Traversal

We apply the above to determine the cache misses incurred by the preorder binary tree traversal experiment of Section 1.1. We consider the standard pointer-based implementation of a binary tree: each node of the tree consists of a key value, a data pointer, and pointers to its left and right children. We assume that the structure of the tree is independent of the nodes' layout in memory and that B tree nodes fit in a cache block. In a recursive implementation of preorder traversal, we examine the root node's key, recursively perform a traversal of the left subtree, then recursively traverse the right subtree. If the access to all of these data fields were truly a random permutation, we would model it as a permutation traversal with access rate $K = 3B$. Instead, since the references to the left and child pointers tend to be hits we assume an access rate $K = B$.

If T is the number of tree nodes and C is the cache size in blocks, Theorem 3.1 gives the total number of misses to be T/B when $T < CB$ and $T - (B-1)C$ otherwise. The curve in Figure 1 graphs these misses per tree node for $B = 2$ and $C = 2^{14} = 16,384$ and varying tree sizes T . We measured the performance of

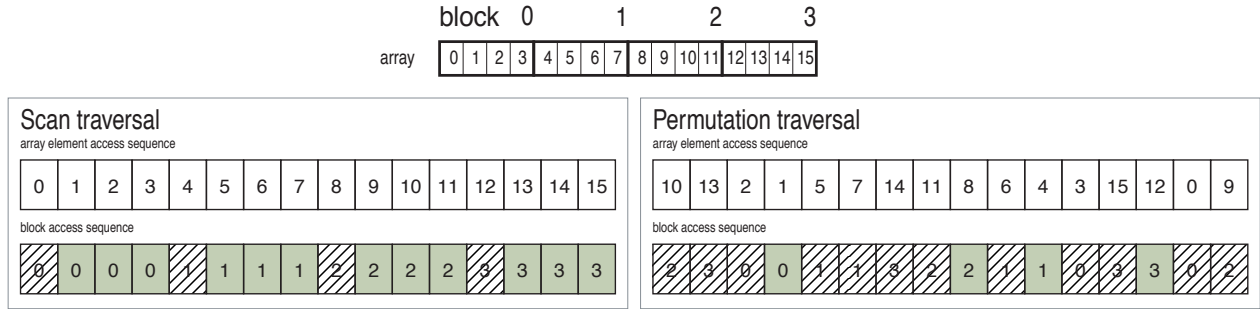


Figure 2: Two traversals of an array where $B = 4$ array elements fit in a block. *Top*: Array layout in memory. *Left*: an array traversal that results in a scan traversal. *Right*: an array traversal that results in a permutation traversal. In both sequences, the accesses that are hits are highlighted.

an implementation of this case using the methodology described in Section 2.2. The implementation’s performance is shown in the plotted points of Figure 1. In the implementation, a tree node used 32 bytes, and we simulated D -way set-associative caches of one megabyte in size with 64 byte blocks with $D = 1, 2, 4, 8$. As can be seen in the figure, our model accurately matches the actual performance.

4 Modeling Tree Accesses

To model non-traversal access to data structures, say, insertions, searches, and deletions, we define *systems of random access* [19]. In a system of random access each block of memory is accessed stochastically with some probability. In most algorithms the block accessed depends on the history of past accesses. However, it is often possible to assume independent reference, where each access is independent of the previous ones. This model of random access was first used to determine the cache performance of algorithms and data structures in LaMarca’s thesis [21], and the analysis used was described as *collective analysis*. It is an adaptation of the *independent reference model* (IRM) of Coffman [11] originally used to model workloads on systems.

Here, as in most applications of the IRM, we consider the cache performance of a system of random accesses as the number of accesses grows large. For each block of memory x , let p_x be the probability that block x is accessed. Similarly, if s is a cache block, let p_s be the probability that some memory block that maps to s is accessed, thus $p_s = \sum_{x \rightarrow s} p_x$. Analysis of the IRM with fully associative caches can be found in King [18] and its subsequent adaptation to set-associative caches appears in Rao [24] which we summarize with the following propositions:

PROPOSITION 4.1. (KING [18]) *If s is a set in a D -way set-associative cache and p_1, p_2, \dots, p_n are the access probabilities to memory blocks $1, 2, \dots, n$, the probability that an access to s is a hit for $n > D$ is*

$$\eta_s = \sum_{\substack{\text{distinct} \\ i_1, \dots, i_D \rightarrow s}} \frac{\left(\prod_{k=1}^D p_{i_k}\right) / p_s^D}{\prod_{k=1}^D \left(1 - \sum_{j=1}^{k-1} p_{i_j} / p_s\right)} \sum_{k=1}^D p_{i_k} / p_s. \quad (4.2)$$

PROPOSITION 4.2. (RAO [24]) *In a set-associative cache with sets S , the probability that an access is a hit is*

$$\eta = \sum_{s \in S} p_s \eta_s, \quad (4.3)$$

where p_s is the probability of an access to set s and η_s is given by (4.2).

For the remainder of this paper, we will consider only direct-mapped and 2-way set-associative caches, thus we derive Proposition 4.1 for $D = 1$ and $D = 2$. For a direct-mapped case, conditioned on the fact that an access is made to set s , the probability that a block $i \rightarrow s$ was accessed is p_i / p_s . It is a hit if the previous access to s was to i and so

$$\eta_s = \frac{1}{p_s^2} \sum_{i \rightarrow s} p_i^2. \quad (4.4)$$

For the two-way set-associative case, the access to $i \rightarrow s$ is a hit if the previous access to s was to i or if there was one intervening access to some $j \rightarrow s$. The probability that an access to s is a hit is then:

$$\eta_s = \frac{1}{p_s^2} \sum_{i \rightarrow s} p_i^2 \left(1 + \sum_{j \neq i; j \rightarrow s} \frac{p_j / p_s}{1 - p_j / p_s}\right). \quad (4.5)$$

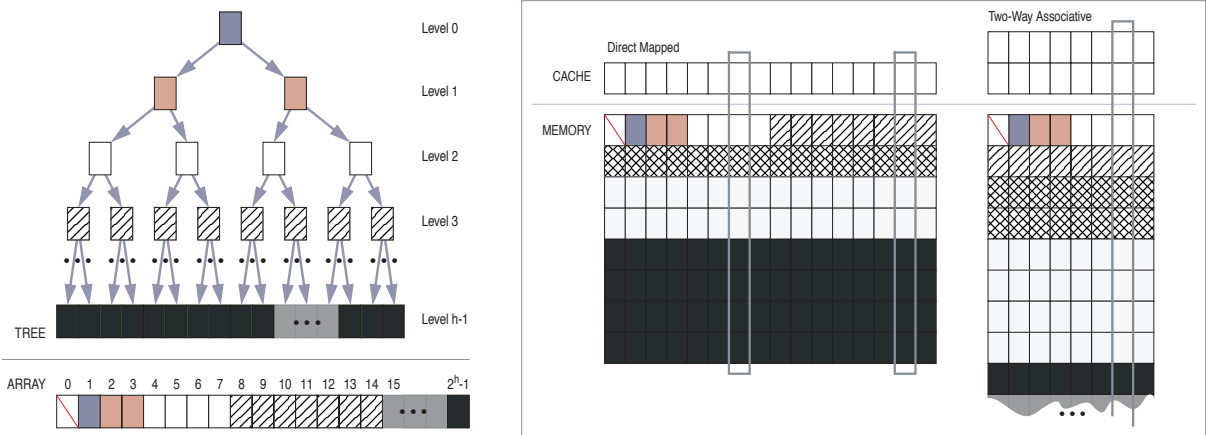


Figure 3: A static binary search tree and its layout in an array (*left*), its layout in memory and its mapping to a direct-mapped cache and a two-way set-associative cache (*right*).

Both of these are equivalent to (4.2) for their respective values of D .

4.1 Search Tree Access as Independent Reference To illustrate the above analysis, we use the IRM to predict the cache performance of accessing a full, static binary search tree. We implement the tree using the implicit pointer technique used for a binary heap [12]. The values in the tree are laid out in a one-dimensional array as shown in Figure 3. Skipping the zero-th element of the array, the value at the root of the tree is the first element in the array, its children are the next two elements in the array, their children are the next four elements, and so on.

The experiment we consider is the following: we build a full static binary search tree with h levels containing $2^h - 1$ keys. We perform 10^5 random key searches in the tree where each search goes to a leaf element, and each search path is equally likely. In our measurements we assume a cache with $C = 2^{18}$ blocks and $B = 1$. Figure 4 gives the number of hits made in these experiments for caches with set-associativity from one to eight, and for tree heights from $h = 12$ to $h = 19$.

The right side of Figure 3 shows a layout of the search tree in memory and its mapping into a direct-mapped cache of $C = 2^c$ blocks. For cache block $1 \leq i \leq 2^c - 1$, the tree values mapped to it are the following: one at level $\lfloor \log_2 i \rfloor$, one at level c , two at level $c+1$, four at level $c+2$ and so on until the last level of the tree. The right side of Figure 3 also shows the mapping that occurs if we have a two-way set-associative cache of the same size. The blocks that map to locations i and $i + 2^{c-1}$ in the direct-mapped cache are mapped to

the same cache set in the two-way set-associative cache. One such pair is highlighted in the figure.

We model the accesses to the tree as random access. The root will always be accessed with each key search, thus accessed every h tree accesses. We model this by assigning a probability of access $1/h$ to the root. Continuing this way, a value stored at level i is accessed with probability $1/(2^i h)$.

Using these probabilities as the block access probabilities for the IRM, we can calculate the expected hit ratios. We omit the calculations here. Figure 4 plots these hit rates for different binary tree sizes assuming direct-mapped and two-way set-associative caches. The model matches very closely the hit rates that we measured with Atom as shown in the graph.

5 Set Mappings

In the two motivating examples we have analyzed, set-associativity had minimal impact on cache performance. To gain an intuition for why, we consider the conditions necessary for set-associativity to be of use, in particular, we consider the mapping of memory blocks to cache sets.

Consider, for example, the mapping of the static search tree shown in Figure 3. The memory mapped to the two cache blocks highlighted in the direct-mapped cache are mapped to the same set in the 2-way set-associative cache. Note that the distribution of accesses to those memory blocks are the same. The cache blocks share the same “access load” in the direct-mapped cache. Our hypothesis, then, is that the set mapping \mathcal{S} is good enough in the direct-mapped case, and that there is little gain in having a LRU policy manage each set of two cache blocks in the 2-way set-associative

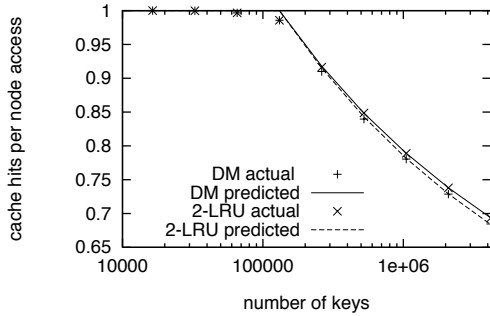


Figure 4: A comparison of actual and predicted cache hits for a sequence of binary search tree lookups.

cache.

This is clearly not the case with all random access sequences. Suppose we have a memory of four blocks, named W through Z , and a direct-mapped cache of two blocks s and t . Let block $W \rightarrow s$ and $X, Y, Z \rightarrow t$. Assume a system of random accesses where $p_W = p$ and $p_X = p_Y = p_Z = (1 - p)/3$ for some probability p .

In the left of Figure 5, we graph the expected hits per access of such a system as a function of p for the direct-mapped cache, and also a 2-way set-associative cache of two blocks. When $p < 1/4$, block W is not accessed as frequently as the other blocks, and so cache block s is underutilized. As a result, the set-associative cache performs best for these values of p , as it tends to keep the most recently accessed blocks, and thus likely the most frequently accessed blocks, in the cache. For $1/4 < p < 1$, however, the other blocks compete with block W in the set-associative cache. Since block W is most likely to be accessed and does not compete for s , the direct-mapped cache gives superior hit rates when $1/4 < p < 1$. In this case, it turns out that the direct-mapped cache has an optimal mapping of memory blocks to cache blocks [18, 15]. In [13] we give a generalization of this example where a direct-mapped cache yields arbitrarily better performance.

Consider a similar second example of a system of random accesses with two additional blocks $U, V \rightarrow s$. Let the probabilities of access be $p_U = 4p/7, p_V = 2p/7, p_W = p/7, p_X = 4(1 - p)/7, p_Y = 2(1 - p)/7$, and $p_Z = (1 - p)/7$, respectively, so that each cache block has the same distribution of accesses. We call this a load balanced system of random accesses with respect to the direct-mapped cache mapping. The expected cache performance for this system is given in the right of Figure 5. When p is close to 0 block s is not well-utilized and most of the accesses are to blocks mapped to t . The hit rate of the two-way set-associative cache

is nearly double that of the direct-mapped cache in this case. However, as p approaches $1/2$, the accesses to each set of blocks is more balanced and there is little difference between the two caches' performance. In fact, for $p = 1/2$ the direct-mapped cache yields slightly more expected hits per access.

We now investigate the performance of systems of random access like this example.

5.1 Load Balanced Mappings Consider two cache configurations: a direct-mapped cache with two blocks s and t , and a 2-way set-associative cache with two blocks. In addition, suppose we have a system of random accesses accessing $2n$ memory blocks. Blocks $1, 2, \dots, n$ are accessed with probabilities p_1, p_2, \dots, p_n , and blocks $n + 1, n + 2, \dots, 2n$ are accessed with probabilities q_1, q_2, \dots, q_n . In the direct-mapped cache configuration, let memory blocks $1, 2, \dots, n$ be mapped to cache block s and the remaining memory blocks be mapped to cache block t . As before, let $p_s = \sum_{i \rightarrow s} p_i$ and $p_t = \sum_{i \rightarrow t} q_i$. We say that the system of random accesses is *load balanced* in the direct-mapped cache configuration whenever $p_i = q_i$ for all i . This additional condition makes it such that each cache block in the direct-mapped cache configuration witnesses the same access behavior. In this case, then, $p_s = p_t = 1/2$.

From (4.4) the direct-mapped cache configuration's steady state hit rate is

$$\eta_D = 4 \sum_{i=1}^n p_i^2$$

From (4.5) the two-way set-associative cache configuration's steady state hit rate is

$$\eta_A = 2 \sum_i p_i^2 \left(1 + 2 \sum_{j \neq i} \frac{p_j}{1 - p_j} \right)$$

In a load balanced system of random accesses there is no advantage gained by a two-way set-associative cache over a direct-mapped cache as given by the following theorem:

THEOREM 5.1. *If a system of random accesses is load balanced then $\eta_D \geq \eta_A$.*

The proof, left for the full paper, uses the method of Lagrange multipliers to find the minimum of $\eta_D - \eta_A$ and to show that it is non-negative [13]. Although direct-mapped caches are better for load balanced distributions the difference appears to be slight: the largest absolute difference we observed was less than 0.043 hits per access, an improvement of less than 6.5%.

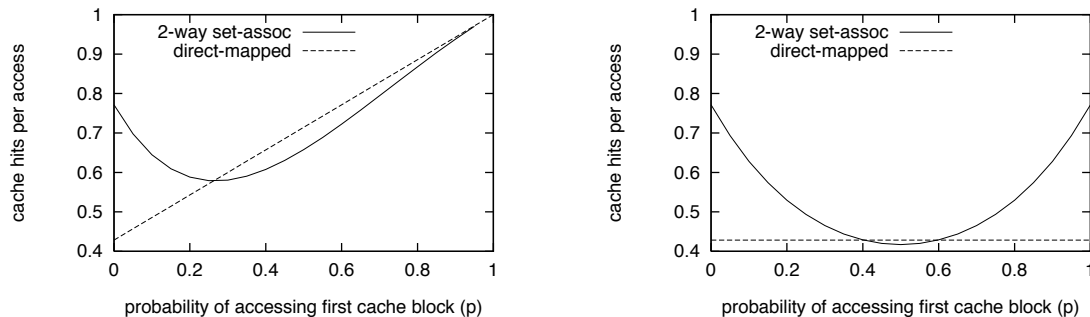


Figure 5: Hits per access for systems of random access illustrating the effect of optimal memory layout (*left*) and load balance (*right*) on cache performance.

6 Conclusions

Random access models like permutation traversal and the IRM have been found useful for accurately predicting the direct-mapped cache performance of access to common algorithms and data structures [19, 20, 22]. Here, we expand the analysis for set-associative caches. The experimental studies show that set-associativity has little impact on performance, and these models explain why.

We focused on set-associative caches for precise cache performance analysis rather than fully associative caches ($D = C$) because they are the caches used in practice. Assuming full associativity can be reasonable for algorithm design, however, especially when constant factors are ignored. In [14] it is shown that a fully associative cache can be simulated in a direct-mapped cache with only a constant factor hit in performance. Also, a design that works well in a fully associative cache can turn out to have good performance in practice.

Clearly, our goal here is not to discourage set-associative cache design as practice, especially with more realistic workloads, suggests that set-associativity is an effective design. Rather, we feel that our results suggest when set-associativity matters—for example, when the direct-mapping of blocks is sub-optimal [13]. Our models might also suggest that cache-conscious data layout tools designed for direct-mapped caches [23, 10, 6, 8] might have limited impact in set-associative caches.

7 Acknowledgements

The author would like to thank Richard Ladner for numerous helpful discussions, for initiating investigation of the work described here, and especially for suggesting the study of set-associative caches.

References

- [1] Anurag Acharya, Huican Zhu, and Kai Shen. Adaptive algorithms for cache-efficient trie search. In *Proceedings of the First Annual Workshop on Algorithm Engineering and Experiments (ALENEX99)*, 1999.
- [2] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.
- [3] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.
- [4] Bowen Alpern, Larry Carter, Ephraim Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2/3):72–109, 1994.
- [5] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *IEEE Symposium on Foundations of Computer Science*, pages 399–409, 2000.
- [6] B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.
- [7] T. Chilimbi. Cache-conscious data structures—design and implementation. Ph.D. thesis, University of Wisconsin-Madison, 1999.
- [8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.
- [9] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.
- [10] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.

- [11] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
- [12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1990.
- [13] J.D. Fix. Cache performance analysis of algorithms. Ph.D. thesis, Department of Computer Science, University of Washington, 2002.
- [14] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.
- [15] S. Gal, Y. Hollander, and A. Itai. Optimal mapping in direct mapped cache environments. *Mathematical Programming*, 63:371–387, 1994.
- [16] S. Gal and B. Klots. Optimal partitioning which maximizes the sum of the weighted averages. *Operations Research*, 43:500–508, 1995.
- [17] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.
- [18] W. F. King. Analysis of paging algorithms. In *IFIP Congress*, pages 485–490, August 1971.
- [19] R. Ladner, J. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses, 1999.
- [20] LaMarca and Ladner. The influence of caches on the performance of sorting. In *Proceedings of the Eleventh Annual Symposium on Discrete Algorithms*, 1997.
- [21] A. LaMarca. Caches and algorithms. Ph.D. thesis, Department of Computer Science, University of Washington, 1996.
- [22] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1:4, 1996.
- [23] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, 1990.
- [24] G. S. Rao. Performance analysis of cache memories. *Journal of the Association of Computing Machinery*, 25(3):378–395, 1978.
- [25] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *The VLDB Journal*, pages 78–89, 1999.
- [26] Jun Rao and Kenneth A. Ross. Making b + - trees cache conscious in main memory. In *SIGMOD Conference*, pages 475–486, 2000.
- [27] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.
- [28] Peter Sanders. Accessing multiple sequences through set associative caches. In *Proceedings of the Twenty-Sixth International Conference on Automata, Languages and Programming*, pages 655–664, 1999.
- [29] Sandeep Sen and Siddhartha Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the Eleventh Annual Symposium on Discrete Algorithms*, pages 829–838, 2000.
- [30] 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, 1994.
- [31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory i: Two-level memories. *Algorithmica*, 12(2/3):110–147, 1994.
- [32] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory ii: Hierarchical multilevel memories. *Algorithmica*, 12(2/3):148–169, 1994.
- [33] Jeffery S. Vitter. External memory algorithms: Dealing with massive data. *ACM Computing Surveys*, to appear.
- [34] L. Xiao, X. Zhang, and S.A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics*, 5, 2000.