AN ANALYSIS OF THE EFFECTS OF SPATIAL LOCALITY ON THE CACHE PERFORMANCE OF BINARY SEARCH TREES

Thomas B. Puzak
The University of Connecticut
Storrs, Connecticut, USA

Chun-Hsi Huang
The University of Connecticut
Storrs, Connecticut, USA

Keywords: Cache Aware Binary Trees, Binary Tree Spatial Locality, Binary Tree Cache Performance.

Abstract: The topological structure of binary search trees does not translate well into the linear nature of a computer's memory system, resulting in high cache miss rates on data accesses. This paper analyzes the cache performance of search operations on several varieties of binary trees. Using uniform and nonuniform key distributions, the number of cache misses encountered per search is measured for Vanilla, AVL, and two types of Cache Aware Trees. Additionally, concrete measurements of the degree of spatial locality observed in the trees is provided. This allows the trees to be evaluated for situational merit, and for definitive explanations of their performance to be given. Results show that the balancing operations of AVL trees effectively negates any spatial locality gained through naive allocation schemes. Furthermore, for uniform input, this paper shows that large cache lines are only beneficial to trees that consider the cache's line size in their allocation strategy. Results in the paper demonstrate that adaptive cache aware allocation schemes that approximate the key distribution of a tree have universally better performance than static systems that favor a particular key distribution.

1 INTRODUCTION

Binary trees are an attractive data structure for representing large data sets because they exhibit an expected sub-linear search complexity. Unfortunately, because they are a pointer based data structure, binary trees perform poorly with respect to caches. This problem is exacerbated with balanced binary trees, which guarantee $\lg n$ search complexity, because the tree's balancing operations destroy any inherent spatial locality present when the keys were initially inserted.

The poor cache performance of a binary tree is in direct contrast with static data structures like arrays. While an array will exhibit good cache performance, especially with large cache lines, its search complexity is an undesirable $O(n)$. Therefore the choice to use a binary tree represents a trade-off, in which the user improves his search complexity at the cost of degraded cache performance.

The focus of this paper is to study the L1 data cache performance of various binary trees with special consideration given to the spatial locality of the data stored within the tree. Previous research has cited spatial locality as a reason for the observed experimental results measuring the cache performance of pointer based data structures, but no concrete measure of spatial locality in the data structures has been provided. This paper provides a method and formula for quantifying the degree of spatial locality inherent in a tree's structure based on the number of cache misses taken while accessing the tree.

Two classical binary trees, vanilla (naive) and AVL, are analyzed in this work. Additionally a *Cache Aware Binary Tree* is defined and studied. A cache aware tree operates by making the tree's topology correspond with the order in which its nodes are allocated. The goal of a cache aware tree is to ensure that the descendants of any particular node are more likely to be located in the same cache line as the node, thereby leading to higher cache hit rates.

In the experimental stage of this work, a program is run that constructs and then repeatedly searches a binary tree. Two key distributions are used to build and search the tree; the first is a uniformly random set, and the second is a highly nonuniform set of keys. The number of L1 data cache misses is used as the metric to evaluate cache performance.

The results demonstrate that for uniform key distributions, large cache lines do not benefit vanilla or

B. Puzak T. and Huang C. (2006).
AN ANALYSIS OF THE EFFECTS OF SPATIAL LOCALITY ON THE CACHE PERFORMANCE OF BINARY SEARCH TREES.
In *Proceedings of the First International Conference on Software and Data Technologies*, pages 94-101
DOI: 10.5220/0001315900940101

AVL trees. Only the cache aware tree, whose nodes are allocated with cache parameters in mind, experiences fewer cache misses as cache line size increases. Additionally, results quantitatively show the negative effects that the AVL tree's balancing operations have on spatial locality, while still highlighting the importance of a short search path. Furthermore, it becomes clear that the cache aware allocation scheme must adapt itself to the observed distribution of the input keys in order to be universally acceptable.

2 RELATED WORK

A function called *ccmorph*, defined in (Chilimbi et al., 1999b; Chilimbi et al., 2000) is designed to rearrange binary trees to make them more cache conscious. The work shows that by morphing a tree into a cache conscious form, the performance of searching the tree (measured in milliseconds) is improved

In (Bawawy et al., 2001) *ccmorph's* performance in conjunction with software based prefetching was studied. Additionally, (Hallberg et al., 2003) showed that cache conscious allocation, particularly with large cache lines, greatly outweighs the benefits of hardware or software prefetching.

The effects of different cache associativities on the cache performance of searching binary trees is studied in (Fix, 2003).

Efforts to improve the cache performance of Binary Space Partitioning (BSP) trees through intelligent allocation of nodes within the tree are studied in (Havran, 2000a; Havran, 1997; Havran, 2000b). The author demonstrates that by allocating multiple nodes in a single cache line and then by distributing those nodes breadth first in the growing tree, the running time of a depth first traversal is improved.

A purely empirical analysis on how the sizes and distributions of input and search keys affects the cache performance of binary trees was conducted in (Iancu and Acharya, 2001). In general it was found that balanced trees generally outperformed those that move nodes to the front, except with search key distributions that are highly skewed to access only a few keys repeatedly.

The work in (Oksanen, 1995) provides a means for predicting the number of cache misses on a particular search of the tree based on the strategy used to allocate the tree's nodes. Additionally, experiments are conducted to measure the performance improvement of tree accesses (in milliseconds) when cache conscious allocation strategies are employed.

3 CLASSICAL BINARY TREES

The vanilla tree is the most naive version of a binary tree. A vanilla tree's topology is wholly determined by the insertion order of the keys used to build the tree. While the expected depth and search complexity of a vanilla tree is $O(\lg n)$ when the tree is built on a uniformly random input distribution (Cormen et al., 1998), other distributions carry no such guarantee.

AVL trees (Adelson-Velskii and Landis, 1962; Weiss, 1999) guarantee $O(\lg n)$ depth and search complexity by adding an invariant to the vanilla tree requiring that for every node in the tree, the heights of that node's left and right subtrees may differ by at most one. Re-balancing of the tree is performed through a set of well known operations called AVL rotations.

4 CACHE AWARE BINARY TREES

The idea behind a cache aware binary search tree is to increase the degree of spatial locality exhibited in the tree's structure by packing nodes located on a particular search path into the same cache line.

Nodes in the tree are labeled either *used* or *unused*. When a key is inserted into a cache aware tree, the standard vanilla protocol is followed with two exceptions. First, if the key being inserted lands on an unused node then the key is placed into that node and the node becomes used. Second, if the key falls to a used leaf node, a block of memory equal to the size of a cache line is allocated. This block is broken up into nodes, which are then distributed into the tree according to some heuristic.

We define the term *local family* to be a group of nodes that were allocated as a block of memory and inserted into the tree together. These nodes all reside in the same cache line.

4.1 Balanced Subtree Ordered Local Families

In this type of cache aware binary tree (Type I Aware Trees), local families are distributed into the tree breadth first. All local families in the tree have identical topologies. Trees built in this fashion favor uniform key distributions, because each local family is of height $\lg(S_{CL})$ where S_{CL} is the size of a cache line. Figure 1 illustrates the process of insertion into a Type I Aware Tree with and without allocation.

In (Havran, 2000a), binary trees are constructed by packing multiple nodes into a cache line and then arranging them breadth first into a binary tree in order

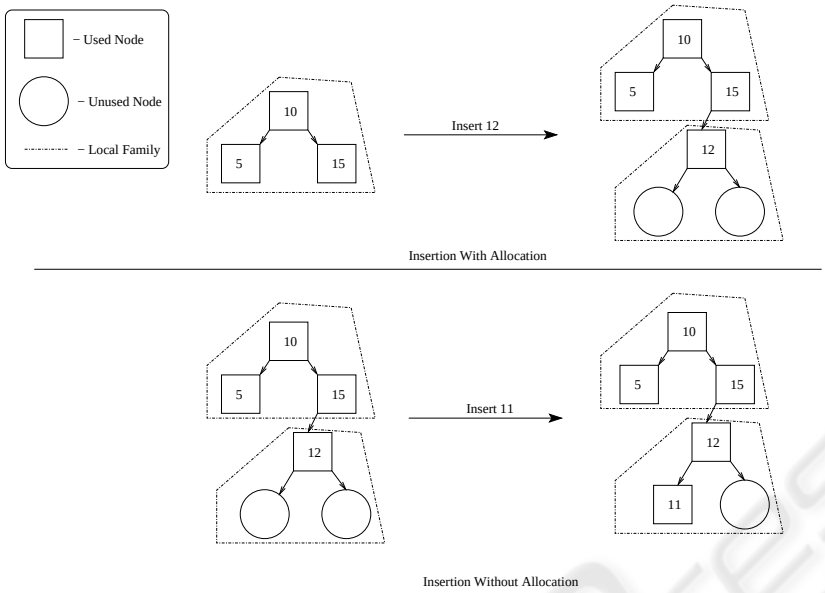

Figure 1: Insertion into a Type I Aware Tree.

to improve the time (CPU cycles) necessary to conduct complete depth first traversal of the tree. In equations 1 - 5 we build on this work to determine the expected number of cache lines accessed when the tree is searched.

We define h_C to be the height of a complete subtree or local family. The height of the root node is defined to be zero. Let S_{CL} be the size of a cache line, and S_N be the size of a node. Then the memory needed to contain a complete local family in the cache aware tree is:

$$M(h_C) = (2^{h_C+1} - 1)S_N \leq S_{CL} \quad (1)$$

From this we derive the complete height of a local family:

$$h_C = \lfloor (\lg(\frac{S_{CL}}{S_N} + 1)) - 1 \rfloor \quad (2)$$

The number of nodes, g_N, in the local family with height greater than h_C is then:

$$g_N = \lfloor \frac{S_{CL} - ((2^{h_C+1} - 1)S_N)}{S_N} \rfloor \quad (3)$$

Then the average height of the local family $h_A \geq h_C$ for $g_N > 0$ is:

$$h_A = (\lg(2^{h_C+1} + g_N)) - 1 \quad (4)$$

For a binary tree of height h_T the expected search depth is $d = h_T - 1$. However, on a search of the tree we expect to access $d + 1 = h_T$ nodes. Therefore the expected number of local families traversed L_{FT} in a single search of the tree is:

$$L_{FT} = \frac{h_T}{(h_A + 1)} \quad (5)$$

Since one local family exists in each cache line L_{FT} is equal to the number of cache lines we can expect to access on a single search of the tree. This value can be used to calculate the the average number of nodes accessed per cache line (see section **??**).

One limitation of subtree ordered local families is that they favor uniform key distributions. The balanced local family topology will have sub-optimal cache performance in nonuniform situations.

4.2 Insertion Path Ordered Local Families

Type II Aware Trees are an attempt to make the allocation of local families closer to optimal by placing unused nodes according to an approximation of the observed input key distribution. Nodes in a Type II Aware Tree contain two additional integer fields, l and r, representing the number of keys inserted into the left and right subtrees of each node respectively. Insertion into the tree without allocation is performed exactly as it is in the balanced subtree ordered case, except that l and r in each node traversed is updated.

When a key is inserted into the tree and allocation is required, two FIFO lists are used to store the last q values of l and r that were encountered as the key traveled down through the nodes of the tree. The value q can be any positive integer, and reflects how many predecessors of a new local family will be considered for determining the key distribution approximation. The contents of the two FIFO lists are used to compute $L = \sum_{i=1}^{q} l_i$ and $R = \sum_{i=1}^{q} r_i$.

The values L and R are used to compute the probabilities of distributing nodes to the left and right in the newly allocated local family: $P(L) = \frac{L}{(L+R)}$, $P(R) = \frac{R}{(L+R)}$. When a new local family is allocated, its root is placed in the tree, marked used, and the key being inserted into the tree is placed in this node. Then the unused nodes in the local family are distributed as descendants of the root node according to $P(L)$ and $P(R)$. The node distribution algorithm utilizes a biased coin flip where $P(L)$ and $P(R)$ are the values of heads and tails, respectively.

For local families of size j the probability of the root node $P(n_0) = 1$. The probabilities of nodes n_i, i∈[1..j-1] is defined with respect to the root as

$$P(n_i) = P(L)^g P(R)^h \tag{6}$$

where g and h are the number of left and right pointers traversed from n_0 respectively.

We consider only the last q nodes to determine $P(L)$ and $P(R)$ so that distant predecessors do not affect the topology of a new local family.

To determine the likelihood of a local family with given topology Υ we need to consider the number of ways that the topology can be constructed Ψ:

$$\Psi = \frac{N!}{\prod S_N} \tag{7}$$

where N is the number of nodes in the tree, and S_N a set whose elements consist of the number of nodes in every subtree of the tree. Then the probability of a local family with a particular topology Υ is:

$$P(\Upsilon) = \Psi \prod P_N \tag{8}$$

where P_N is a set of the probabilities of all of the nodes in local family, as calculated in equation 6.

Figure 2 illustrates the process of insertion into a Type II Aware Tree requiring allocation. In this example, $q = 2$ and the size of a local family is 4. It is important to remember that the newly allocated local family in Figure 2 is not the only possible topology that can result from the node distribution method. In this figure $P(L) = \frac{1}{3}$ and $P(R) = \frac{2}{3}$, so the topology depicted in the figure represents the expected node distribution.

5 EXPERIMENTAL DESIGN

Experiments are run using the SimpleScalar (Burger and Austin, 2001) simulation suite. Two rounds of experiments are conducted. The first round is designed to measure detailed metrics about cache performance. The second round computes metrics on the structure of a tree which are used to quantify its spatial locality.

In order to measure cache performance metrics a simple C program is run that builds a binary tree from the keys in an input file, and then repeatedly searches the tree for the keys in a separate file. The size of the cache blocks are multiples of 32 bytes, and the nodes in every tree are exactly 32 bytes. For Type II Aware Trees the value of q is set to 10. The program can run in two modes: *build-only* or *build-and-search*. In build-only mode the program terminates after building the tree, while in build-and-search mode the program terminates after building the tree and then searching for all of the keys located in the search key file.

5.1 Cache Performance Experiments

For the cache performance experiments the L1 is a 4-way set associative cache; each run is repeated with L1 block sizes of 32, 64, 128, and 256 bytes; for each L1 block size, a run is repeated with L1 cache sizes of 1, 2, 4, 8, 16, 32, 64, 128, and 256 Kb; the size of the L2 is always 1024 Kb; the L2 is always a unified 8-way set associative cache; and all caches utilize the LRU replacement algorithm. The parameters of the L2 were chosen so that L2 misses would have a minimal impact on the performance of the L1. At the program's termination SimpleScalar reports the total number of L1 data cache misses during execution.

To compute the average number of L1 data cache misses per search M, we first acquire α: the number of L1 misses taken in build-only mode, and β: the number of L1 misses taken in build-and-search mode. On n searches of the tree, M is computed as follows:

$$M = \frac{(\beta - \alpha)}{n} \tag{9}$$

5.2 Tree Structure Experiments

This round of experiments is designed to measure metrics about searching a binary tree, as well as to provide insight into the degree of spatial locality present in the tree. The metrics of interest here are: the average number of nodes accessed per search, the average number of cache lines accessed per search, the average number of nodes accessed per cache line, and a value for the amount of a cache line that is filled with accessed nodes.

To compute the average number of nodes accessed per search, λ, a global count c of each node encountered on every search operation is maintained. On n searches of the tree, the average number of nodes accessed per search is:

$$\lambda = \frac{c}{n} \tag{10}$$

To determine the average number of cache lines accessed per search, γ, the cache line address of each

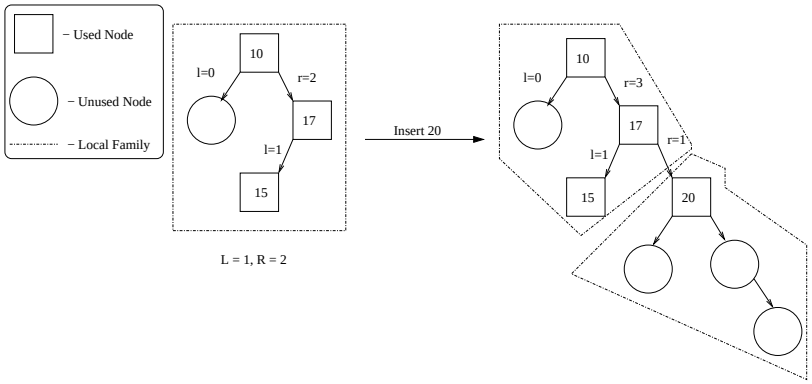

Figure 2: Insertion With Allocation Type II Aware Tree.

node encountered in a search is computed. For each search the number of unique cache line addresses, u, is recorded. For n total searches γ is computed:

$$\gamma = \frac{\sum_{i=1}^{n} u_i}{n} \tag{11}$$

The average number of accessed nodes per cache line, Φ, is simply:

$$\Phi = \frac{\lambda}{\gamma} \tag{12}$$

From Φ we derive an expression for the density of the accessed nodes in a cache line Δ:

$$\Delta = \Phi \frac{(S_N)}{S_{CL}} \tag{13}$$

Δ exists in the range $[\frac{S_N}{S_{CL}}, 1]$ and expresses the amount of available cache line space that is being utilized by nodes that were accessed.

For Cache Aware Trees the number of unused nodes in the tree, w, is calculated by a complete depth first traversal of every node (including unused) in the tree.

5.2.1 Theoretical Bounds

It is possible to bound w and Δ for Type I Aware Trees based on the topological structure of a local family. We define the number of nodes in a local family to be $\eta = \lfloor \frac{S_{CL}}{S_N} \rfloor$, and N to be the number of keys inserted into the tree.

The bounds of these values rely on the average height of a local family, h_A, defined in equation 4. These local families have identical topologies representing approximately balanced trees, hence h_A is $O(\lg \eta)$. Therefore we can expect to have the greatest number of unused nodes when only $h_A + 1$ used nodes reside in each local family. For N insertions we will allocate $\frac{N}{(h_A+1)}$ local families, each of which contains $\eta-(h_A+1)$ unused nodes. Multiplying these quantities we obtain an upper bound for w:

$$\overline{w} = \frac{N\eta}{(h_A+1)} - N = O(\frac{N\eta}{\lg \eta} - N) \tag{14}$$

To bound Δ we must realize that regardless of the key distribution, we can access at most $h_A + 1$ nodes in each local family.

$$\overline{\Delta} = \frac{(h_A+1)}{\eta} = O(\frac{\lg \eta}{\eta}) \tag{15}$$

For Type II Cache Aware Trees, we can reasonably expect our biased coin flip to predict the position of an future key half of the time. Since the root of the local family is always used we predict $\frac{\eta}{2} + 1$ used nodes, and $\frac{\eta}{2} - 1$ unused nodes per local family. The expected number of unused nodes is:

$$\overline{w} = N(\frac{\eta - 2}{\eta + 2}) = O(N) \tag{16}$$

Of course, it is difficult to obtain a strong bound for w and a bound for Δ for Type II Aware Trees because the local families in these trees are not guaranteed to have identical topologies. Regardless of this shortcoming, we can expect general performance trends depending on the key distribution. For example, if the key distribution is uniform, the topologies of the local families will be balanced, and the values of $\overline{w}$ and $\overline{\Delta}$ approach those of Type I Aware Trees. However, for nonuniform key distributions, there is the potential for highly elevated performance. Indeed $\overline{w}$ approaches 0 and $\overline{\Delta}$ approaches 1. This expectation is reflected in the results in section 6.

5.2.2 Type I Aware Tree Predictions

The average height of a local family in a Type I Aware Tree allows equation 5 to be applied to predict the values for the tree structure experiments. The predictions are based on modifications of equations 11 – 13:

$\gamma' = \frac{\lambda}{h_A+1}$, $\Phi' = \frac{\lambda}{\gamma'}$,and $\Delta' = \Phi' \frac{(S_N)}{S_{CL}}$. Hence, the predictions only rely on the direct calculation of λ from equation 10.

5.3 Key Distribution Details

Each round of experiments is repeated with two key distributions. The importance of key distributions on the structure and cache performance of binary trees was made clear in (Iancu and Acharya, 2001).

The first distribution is uniformly random. Trees are built on an input file of 1,000,000 keys of which 99,996 are unique, taken from the range [0,99999]. On a duplicate insertion the topology of the tree is not changed. Using the same distribution on the range [0, 99999] the tree is searched 5,000,000 times.

The second distribution represents a highly nonuniform key set. The search key file consists of 3,118,271 values taken from the instruction addresses of a running program. Because of their origin, this file consists of long sequences of increasing numbers. The file used to construct the tree, consisting of 174,719 keys, was created from the search key file by taking unique keys from the search file in an order preserving fashion.

6 EXPERIMENTAL RESULTS AND ANALYSIS

In the cases with a uniform key distribution, the vanilla tree and the cache aware trees[1] had maximum heights of 44, while the AVL tree had a maximum height of 20. For the nonuniform distribution, the vanilla tree and the aware trees had maximum heights of 1728, while the AVL tree had a maximum height of 21. These values have substantial consequences with respect to the cache performance of the trees. Because of space restrictions only tables of representative results are presented[2]. In the tables, C refers to cache size, and B refers to the cache data block size.

For the Vanilla tree built on the uniform key distribution, the number of misses per search decreases as cache size increases, but stays the same or slightly increases as cache line size increases with fixed cache size. This trend is due to the poor degree of spatial locality in the Vanilla tree; with 256 byte cache lines $\Phi = 1.20$. With a nonuniform key distribution however, large cache lines lead to improved cache performance. The highly predictable nature of the nonuniform distribution and naive allocation scheme results in a natural spatial locality ($\Phi = 5.91$ at 256 byte lines).

AVL trees had far fewer misses overall than their Vanilla cousins, but larger cache lines do not contribute to improved performance. See Table 1 for an example. The observed cache performance is due to the very low degree of spatial locality present in the cache aware tree. Indeed, $\Phi = 1.07$ when the key distribution is uniform, and only 1.14 when it is nonuniform.

Somewhat ironically, the AVL tree has the best cache performance of any tree analyzed when the key distribution is nonuniform. This is attributable to the average number of nodes accessed per search λ. With the nonuniform key distribution $\lambda = 17.05$ for AVL trees and $\lambda = 442.06$ for the Vanilla and Cache Aware Trees. In other words, a typical search of a Vanilla or Aware tree performs 26 times more memory accesses than the same search on an AVL tree. More memory accesses lead to more cache misses.

With the uniform key distribution, the Type I aware tree takes fewer cache misses per search as both cache size and cache line size increases. Type II Aware Trees have performance nearly equal to that of Type I Aware Trees. See Table 2 for the Type I Aware Tree's cache performance results. This performance is attributable to the good degree of spatial locality observed in the cache aware trees. Tables 3 and 4 represent the predicted and observed values of the tree structure experiments for the Type I aware trees on uniform and nonuniform distributions, respectively.

The accuracy of the predictions for the uniform case suggests that the model is a good approximation of the physical world. The larger discrepancies in the nonuniform case exist because we can only expect to traverse an average of $h_A + 1$ nodes per local family if the key distribution is uniform.

On the nonuniform key distribution, the number of misses per search for the Type I Aware Tree improves as cache size increases. However, with fixed cache size more cache misses are observed as cache line size increases from 32 to 64 bytes. Increases in line size beyond 64 bytes lead to modest decreases in cache misses. This observation is explained by the nature of the nonuniform input distribution. In this case, most new keys are inserted to the right, but the node distribution algorithm favors placing nodes to the left first. Therefore, with 64 byte cache lines, unused nodes almost never become used.

Type II Aware Trees enjoy elevated cache performance as both cache size, and cache line size increases regardless of the input distribution. See Table 5 for the results from the nonuniform distribution. The good cache performance of the Type II Aware tree, regardless of its key distribution, is due to its high level of spatial locality. The results of the tree structure experiments for the Type II Aware Tree and

[1] Only used nodes are counted in the height of the cache aware trees.

[2] For complete results please see (Puzak, 2006)

Table 1: Average Cache Misses Per Search, AVL Tree, Uniform Distribution.

B	C=1K	C=2K	C=4K	C=8K	C=16K	C=32K	C=64K	C=128	C=256K
32	21	13.98	11.8	10.51	9.24	7.96	6.69	5.44	4.18
64	23.98	16.26	12.77	11.26	9.88	8.59	7.32	6.04	4.73
128	22.72	18.8	14.42	12.12	10.41	9.02	7.71	6.42	5.09
256	20.43	19.53	17.43	12.96	10.94	9.33	7.96	6.66	5.32

Table 2: Average Cache Misses Per Search, Type I Aware Tree, Uniform Distribution.

B	C=1K	C=2K	C=4K	C=8K	C=16K	C=32K	C=64K	C=128	C=256K
32	36.7	23.19	17	14.33	12.28	10.42	8.7	7	5.3
64	28.66	18.25	11.97	10.05	8.69	7.43	6.27	5.14	4
128	18.99	13.63	8.65	7.15	6.17	5.28	4.49	3.73	2.98
256	13.6	11.42	7.38	5.47	4.74	4.06	3.51	2.97	2.43

Table 3: Predicted and Observed Type I Aware Tree Results, Uniform Distribution.

B	λ	γ	γ'	Φ	Φ'	Δ	Δ'	w
32	22.44	22.44	22.44	1.00	1.00	1.00	1.00	0
64	22.44	15.29	14.16	1.47	1.58	0.74	0.79	33410
128	22.44	10.16	9.66	2.21	2.32	0.55	0.58	77820
256	22.44	7.53	7.08	2.98	3.16	0.37	0.40	155676

Table 4: Predicted and Observed Type I Aware Tree Results, Nonuniform Distribution.

B	λ	γ	γ'	Φ	Φ'	Δ	Δ'	w
32	442.06	442.06	442.06	1.00	1.00	1.00	1.00	0
64	442.06	436.11	278.90	1.01	1.59	0.51	0.80	164411
128	442.06	220.17	190.38	2.01	2.32	0.50	0.58	174037
256	442.06	147.34	139.45	3.00	3.17	0.38	0.40	291169

Table 5: Average Cache Misses Per Search, Type II Aware Tree, Nonuniform Distribution.

B	C=1K	C=2K	C=4K	C=8K	C=16K	C=32K	C=64K	C=128	C=256K
32	1085.63	1059.21	1000.74	883.72	699.66	391.9	88.51	5.77	0.89
64	553.99	540.2	510.51	455.25	357.35	204.67	51.31	6.57	0.47
128	284.37	277.43	264.13	237.32	187.42	108.81	25.93	2	0.34
256	149.34	146.69	140.21	126.74	101.46	62.78	18.11	0.26	0.26

the nonuniform input distribution is provided in Table 6. In addition to high degrees of spatial locality regardless of the input distribution, Type II Aware Trees waste only about 12% of the nodes wasted by Type I Aware Trees. Consideration of these observations suggests that adaptive cache aware allocation schemes are superior to methods that are biased to a particular key distribution.

Table 6: Tree Structure Results, Type II Aware Tree, Nonuniform Distribution.

B	λ	γ	Φ	Δ	w
32	442.06	442.06	1.00	1.00	0
64	442.06	227.35	1.94	0.97	5023
128	442.06	118.9	3.72	0.93	15377
256	442.06	64.16	6.89	0.86	35577

7 CONCLUSIONS

It is impossible to ignore the importance of a short search depth, so a balanced binary tree should always be among the first choice of a programmer regardless of the expected key distribution. However, if a uniform key distribution is expected and the target architecture is built around large cache lines, then a cache aware tree would be an excellent choice as search depth would not be much greater than the ideal $\lg n$ and the cache performance will be highly elevated.

For nonuniform inputs, it seems much better to utilize a balanced tree because search depth becomes a major limiting factor. If the cache is made up of very many large lines, a good alternative may be a Type II Aware Tree because it will be able to attain high levels of cache performance despite the nonuniform nature of the keys.

On the uniform key distribution, the cache performance of the trees is ranked as follows: Type I Aware Tree, Type II Aware Tree, AVL Tree, Vanilla Tree.

On the nonuniform key distribution, the cache performance of the trees is ranked: AVL Tree, Type II Aware Tree, Type I Aware Tree, Vanilla Tree.

REFERENCES

Adelson-Velskii, G. and Landis, E. (1962). An algorithm for the organization of information. *Doklady Akademii Nauk SSSR. English translation by Myron J. Ricci in Soviet Math.*

Bawawy, A., Aggarwal, A., Yeung, D., and Tseng, C. (2001). Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *15th Annual Conference on Supercomputing*, page 486.

Bryant, R. and O'Hallaron, D. (2001). *Computer Systems: A Programmer's Perspective.* Prentice Hall Inc, New Jersey.

Burger, D. and Austin, T. (2001). *The SimpleScalar Tool Set, Version 2.0.* SimpleScalar LLC.

Chilimbi, T., Davidson, B., , and Larus, J. (1999a). Cache-conscious structure definition. In *SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99)*.

Chilimbi, T., Hill, M., and Larus, J. (1999b). Cache-conscious structure layout. In *SIGPLAN '99 Conference on Programming Languages Design and Implementation (PLDI 99)*.

Chilimbi, T., Hill, M. D., and Larus., J. R. (2000). Making pointer-based data structures cache conscious. *Computer Magazine*, 33(12):67.

Cormen, T., Leiserson, C., and Rivest, R. L. (1998). *Introduction to Algorithms*. The MIT Press.

Fix, J. (2003). The set-associative cache performance of search trees. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 565.

Hallberg, J., Palm, T., and Brorsson, M. (2003). Cache-conscious allocation of pointer-based data structures revisited with hw/sw prefetching. In *Second Annual Workshop on Duplicating, Deconstructing, and Debunking(WDDD)*.

Havran, V. (1997). Cache sensitive representation for bsp trees. In *Computergraphics 97, International Conference on Computer Graphics*, page 369.

Havran, V. (2000a). Analysis of cache sensitive representation for binary space partitioning trees. *Informatica*, 23(2):203.

Havran, V. (2000b). *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University.

Iancu, C. and Acharya, A. (2001). An evaluation of search tree techniques in the presence of caches. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software*.

Oksanen, K. (1995). Memory reference locality in binary search trees. Master's thesis, Helsinki University of Technology.

Puzak, T. B. (2006). The effects of spatial locality on the cache performance of binary search trees. Master's thesis, The University of Connecticut.

Sleator, D. and Tarjan, R. (1985). Self adjusting binary search trees. *Journal ACM*, 32:652.

Weiss, M. (1999). *Data Structures and Algorithm Analysis in JAVA*. Addison-Wesley Longman Inc.