Memory Reference Locality and Periodic Relocation in Main Memory Search Trees*

Kenneth Oksanen† Lauri Malmi‡
cessu@cs.hut.fi lma@cs.hut.fi

Laboratory of Information Processing Science,
Helsinki University of Technology,
Otakaari 1, 02150 Espoo, Finland

Abstract

Memory reference locality is widely known to have an increasingly important effect on the performance of algorithms. We demonstrate and explain why the performance of the binary tree search algorithm varies considerably depending on how the tree is laid out in the main memory. We present an algorithm which relocates a binary tree so that maximal performance can be achieved and discuss how updates degrade the locality of the tree.

1 Introduction

Binary search trees are widely used index structures in applications where the whole index can be stored in main memory. To avoid the bad worst case behaviour of these trees, various strategies have been developed to maintain them in balance. Global balancing algorithms periodically rebuild the whole tree [3, 9, 15]. Balanced trees, e.g. AVL-trees and red-black trees, perform rebalancing coupled with each update operation [1, 5, 6, 8, 11, 13].

These data structures and algorithms were developed and analyzed assuming the Random Access Memory (RAM) model [2]. This was a reasonable assumption when memory was slow and processors even slower. However, a long term trend has been that processor speed has doubled every two years and memory latencies have halved every ten years. Today, elaborate hierarchical memory and cache designs are used to feed the processor with the data it needs. In

*This paper will appear without Section 5 in the Proceedings of the 5th Hellenic Conference of Informatics, Athens, Greece, 1995.

†This work was supported by the Finnish Technology Development Centre.

‡This work was supported by the Finnish Ministry of Education.

current systems memory references with poor locality cost easily 100 times more than memory references with good locality. This factor is necessary to consider when designing algorithms and data structures.

We studied the performance of ordinary binary search trees and some common balanced trees on a wide range of current computers using the same search procedure for each data structure. Thus the variation of results on a given computer was effected only by the shape and locality of the trees. All data remained in main memory and no disk IO was involved. The results show that the memory reference locality has a dramatic effect on the performance of these trees — an optimally located version of a binary tree is 100-200 percent faster in search and scan operations than a structurally equivalent tree with poor locality.

The main factor in explaining this difference is the *translation lookahead buffer* (TLB), which is an essential part of modern processors. Poor memory reference locality causes TLB thrashing, which increases the cost of reading tree nodes from memory. We devise a linear time algorithm to relocate the nodes of the tree so that TLB thrashing is considerably reduced. Updates to the tree will eventually degrade the locality of the tree, and relocations will have to be performed periodically. We study the rate of degradation and discuss the cases where periodic relocation is feasible.

The rest of this paper is organized as follows: In Section 2 we explain some properties and trends of current hardware systems. In Section 3 we explain why binary tree operations risk thrashing the TLB and present an algorithm to relocate the tree in memory in order to reduce the number of TLB misses. Section 4 presents experiments on the effect of memory reference locality on search speed and how memory reference locality degrades when updates are made in the tree. Some means to slow down the degradation of locality are presented in Section 5. Section 6 concludes.

2 The Translation Lookaside Buffer

When a processor issues a read or write instruction from memory, it must first translate the logical memory address seen by the user program to a physical memory address understood by the memory hardware. In order to reduce the cost of this translation processors contain a fast associative memory, the *translation lookaside buffer* (TLB), of recently used address translations [14]. If the address being referred to can not be mapped by the TLB, a *TLB miss* has occurred and the mapping is resolved by referring to the data structures of the operating system. Current estimates for the cost of handling a TLB miss are between 20 and 40 instructions [10], but the overhead has been predicted to grow to 100 instructions in the future [4].

Currently TLBs contain from tens up to a few hundred *TLB entries*, each covering the mapping of one virtual memory page, typically 4 or 8 kB. A TLB can thus map at least several hundred kilobytes, often even megabytes. This is enough to cover all, or at least the most used part, of the working sets of most programs. TLB misses are therefore relatively rare and incur only a modest overhead.

There are, however, some applications for which TLB thrashing is a considerable overhead and special measures have to be taken. For instance, in some systems the frame buffer is mapped directly to the address space of the X11 server with one dedicated TLB entry. This, however, requires special operating system support and is not viable as a generic solution.

Operations on large matrices can easily thrash the TLB. Consider, for example, the following two equivalent methods for initializing a large identity matrix:

```
INIT-IDENTITY-MATRIX1(I, N)
for i := 1 to N do
   for j := 1 to N do
      if i = j then
         I[i, j] := 1
      else
         I[i, j] := 0
```

```
INIT-IDENTITY-MATRIX2(I, N)
for j := 1 to N do
   for i := 1 to N do
      if i = j then
         I[i, j] := 1
      else
         I[i, j] := 0
```

Typically either one of these routines is at least twice as fast as the other. Which one is faster depends on whether the matrix is stored in row- or column-major form in the memory. Accessing consecutive memory locations is much faster than accessing every N'th memory location. Fortunately, numerical analysts are accustomed to writing such routines in the more efficient manner [7].

Could TLB thrashing be remedied by improving the hardware and the operating system? The simplest approach would be to increase all page sizes uniformly. This, however, would lead to increased overall memory consumption, decreased performance of paging, and other problems in overall system performance.

Many current processors provide for variable size TLB entries, and indeed this could be used to alleviate TLB thrashing. Unfortunately, no common operating system we know of provides means for the user to allocate very large pages. Providing variable size pages requires invasive operating system modifications that introduce considerable overhead [16].

Increasing the number of TLB entries costs expensive on-chip silicon. Talluri et al. [16] give means to decrease the silicon of a TLB entry by introducing only modest operating system support, but they still fail to provide satisfactory solutions for applications with large working sets.

And finally, the mapping from logical to physical addresses is the basis of so many important operating system services that it is very unlikely to be eliminated, except for some special purpose systems. We are therefore prompted to seek means to avoid TLB thrashing by restructuring the algorithms and data structures.

3 Relocating Binary Trees

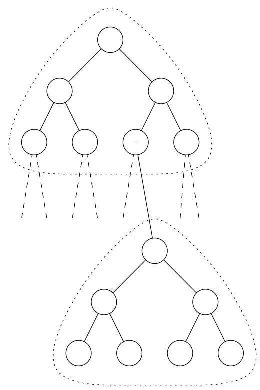
Operations on large binary trees can easily thrash the TLB. Consider the following loop which performs the standard binary tree search:

```
SEARCH(key)
p := root
while p ≠ NIL do
  switch
    case p → key > key do p := p → left
    case p → key < key do p := p → right
    case default  do return p → data
return NIL
```

Each iteration of the loop reads a node from memory and executes roughly 10 simple instructions, but if a TLB miss occurs, the iteration will last the equivalent of over 20 additional instructions effectively doubling or trebling the execution time. We confirmed this experimentally on a wide range of current computer systems. Analysis presented in [12] can be used to calculate the average number of TLB misses per search operation.

A similar slowdown appears for scan operations, but insertions and deletions have a slightly smaller slowdown because they execute more computational instructions, particularly if they perform balancing.

The figure on the right shows how to position a binary tree in memory pages so that the expected number of TLB misses on a path from root to the searched node is minimal. Areas surrounded by a dotted curve represent the contents of a virtual memory page. In the figure only 3 levels of nodes fit on a page, but in the experiments presented later we have used 16 byte nodes on a system with 4 kB pages, which allows us to allocate 8 levels of nodes on each page.



Wilson et al. call this layout of nodes the "hierarchical decomposition" of the tree [17], and depict how to hierarchically decompose a tree during garbage collection. Wilson's research was originally motivated by the need to reduce paging in large LISP systems, but the same principles apply to reducing TLB thrashing.

We give below an algorithm to build a hierarchically decomposed tree. The pseudo code uses Algol-like syntax and arrays augmented so that keywords **ref** and **addr of** correspond to C prefix operators `*` and `&` respectively: The expression **ref** *ptr* evaluates to the value pointed to by *ptr* and **addr of** *item* gives the address of *item*. The syntax $ptr \to field$ is equivalent to (**ref** x).*field*.

We assume the tree is perfectly balanced so that all leaf nodes are either on level *height* or $height - 1$. If this is not the case, some global rebalancing algorithm [3, 9, 15] should be employed prior to relocation.

We also assume we have a routine ALLOCATE which allocates memory for a given amount of nodes so that the allocated area does not cross a page boundary. After the relocation call the old tree can be freed.

```
COUNT-NODES(tree : addr of Node)
if tree = NIL then
   return 0
else
   return COUNT-NODES(tree → left) + COUNT-NODES(tree → right) + 1
```

```
RELOCATE(tree : addr of Node, height : Integer)
 1 if tree = NIL then
 2    return NIL
 3 if 2^(height−1) ≥ max_nodes_per_page then
 4    n := max_nodes_per_page
 5 else
 6    n := max(COUNT-NODES(tree), max_nodes_per_page)
 7 mem := ALLOCATE-NODES(n)
 8 mem[1] := ref tree
 9 j := 2
10 for i := 1 to n do
11    if mem[i].left ≠ NIL then
12       if j ≤ n then
13          mem[j] := ref mem[i].left
14          mem[i].left := addr of mem[j]
15          j := j + 1
16       else
17          mem[i].left := RELOCATE(mem[i].left, height − ⌊log2 i⌋ − 1)
18    if mem[i].right ≠ NIL then
19       if j ≤ n then
20          mem[j] := ref mem[i].right
21          mem[i].right := addr of mem[j]
22          j := j + 1
23       else
24          mem[i].right := RELOCATE(mem[i].right, height − ⌊log2 i⌋ − 1)
25 return addr of mem[1]
```

The algorithm works similarly to Wilson's garbage collector [17]. A call to RELOCATE allocates a large contiguous piece of memory, *mem*, into which it copies nodes from the old tree by performing a breadth first traverse. The variable i is used to traverse all binary tree nodes in *mem*. The children of the node at *mem*[i] are copied from the old tree either to a later place in *mem*, or if *mem* is full, a recursive call to RELOCATE is issued to relocate the subtree. The pointers *left* and *right* in *mem*[i] are updated to refer to the relocated nodes. The variable j tells where in *mem* to place the next relocatable node. Note that i is always less than j.

The breadth-first traversal in the function RELOCATE visits each node once and, due to the limitation on line 3 in RELOCATE, COUNT-ITEMS visits each node atmost twice. The relocation algorithm requires thus linear time.

In a practical implementation the logarithm $\lfloor \log_2 i \rfloor$ could be replaced by a variable which is incremented at the end of the loop whenever the loop counter i becomes a power of two, which in turn can be tested by simple bit operations.

4 Degradation of Locality

Updates and particularly rotations due to balancing alter the relative locations of nodes in the tree and gradually degrade the locality. We performed some experiments to see how rapidly various update policies degrade the performance. The trees used in the experiments contain $N = 2^{16} - 1 = 65535$ nodes with keys in the range from 0 to $2N$, each node used 16 bytes, and the whole tree used one megabyte.

The experiments were conducted on a 90 MHz Pentium with a 256 kB external cache and 4 kB pages. This allowed us to allocate 8 full levels of nodes on each page. Unfortunately the details of the TLB structure of the Pentium are not disclosed, but we suspect that the TLB has approximately 64 entries giving it a coverage of 256 kB. Results were similar in all other systems we tried (Alpha 21064, MIPS R4000 and R4600, SuperSPARC, HP Gecko and PowerPC 601). On the 64-bit Alpha the node size was 32 bytes and the page size 8 kB.

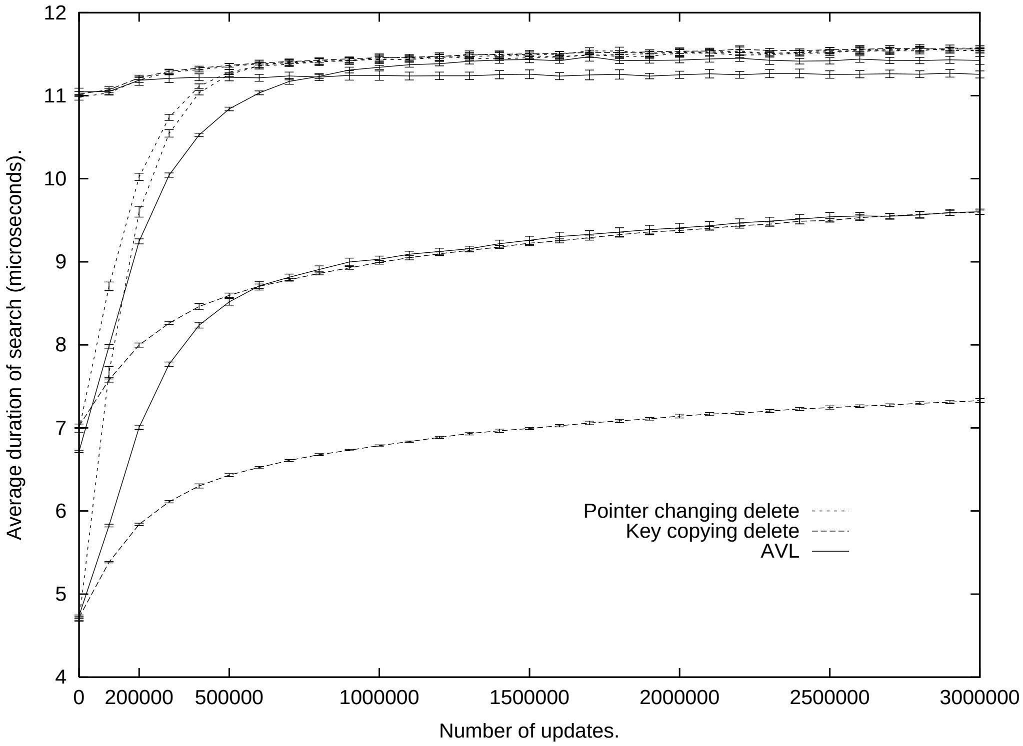


Figure 1: Search performance decrease due to locality degradation caused by updates in AVL-trees and two versions of binary tree deletion. All algorithms tested with three initial memory layouts: optimal, pre-order allocation of nodes from a linear free list, and random location of nodes. Note that update requests for AVL-trees were different from update requests for ordinary binary trees. Tree size is 65535.

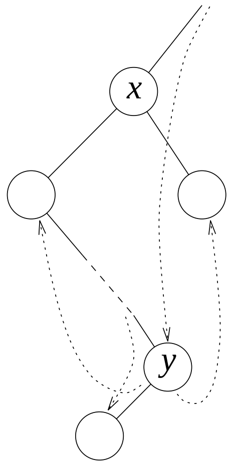


Figure 2: Deletion of an internal node x in a binary tree by changing pointers. New pointer values illustrated with dotted lines.

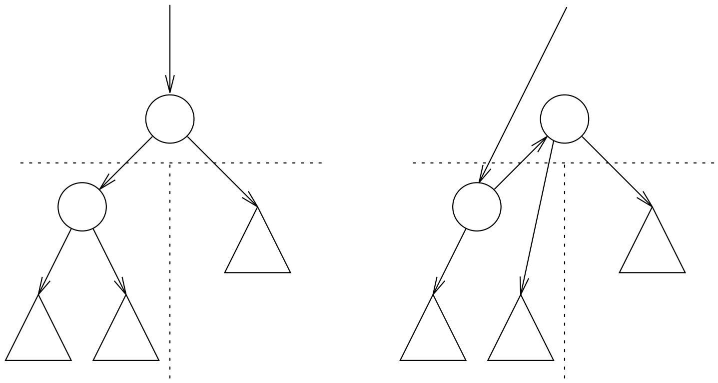


Figure 3: This illustration of a subtree near a page boundary prior to and after a rotation shows how the average number of pages a search path resides on can increase. Dotted lines represent page boundaries.

We experimented with three initial states of the tree. The first and most efficient ($\sim 4.7\mu$s per search) initial tree was the perfectly balanced binary tree laid optimally out in memory so that no search path had nodes on more than two pages.

The second initial tree was the result of constructing the tree by its default insertion algorithm allocating the nodes from a free list of successive memory locations. For ordinary binary trees the insertion was done in such an order that the resulting tree was completely balanced and a left child was allocated adjacent to its parent node. AVL trees were constructed by adding the keys sequentially and relying on the AVL balancing algorithm.

The third, slowest, initial tree (over 11 μs) was also perfectly balanced, but shuffled in memory to random order. Its performance decreased only due to increasing imbalance. Comparing this case to the previous two cases suggests locality can have a larger effect on search performance than a slight imbalance.

There are two ways to perform deletion in binary trees. The first one changes pointers to lift the rightmost node y of the left subtree to replace the deleted node x (see Figure 2), the other copies the key and data of y to x and frees y. Apparently the former method degrades locality much faster than the latter. The latter method was used also in AVL trees.

Updates for ordinary binary trees were performed so that the key to be inserted or deleted was chosen randomly from the range 0 to $2N$ according to uniform distribution. As the the size of the tree is N, approximately half of the searches succeeded. If the key to be inserted was already in the tree or the key to be deleted was not in the tree, nothing was done and another key was chosen instead. This key generation scheme preserves the balance rather well even if no explicit balancing was performed.

All searches were performed using exactly the same search algorithm. Search speeds were measured after every 100 000th successful insertion and deletion. During the updates the tree size varied from N to $N - 10000$.

All experiments were performed 100 times. The 20 slowest executions were discarded because they were slowed down by cache pollution caused by other processes on the machine. The undiscarded results were presumed to conform to normal distribution and the average values were used for the plots augmented with 95% confidence intervals.

AVL trees were updated differently because we wanted to stress the balancing in them. The key was chosen at random only with probability $\frac{1}{16}$, otherwise the key was the successor of the previously used key. The rotations performed in AVL trees clearly increase the degradation of locality. This is illustrated in Figure 3. Note that rotations rarely propagate to the top of the tree leaving it relatively untouched for rather long. Experiments we omit here for brevity suggest that the larger the variance in tree size is the faster the locality of the tree degrades.

5 Decelerating the Degradation of Locality

In the previous experiment a freed node was pushed on top of a list of free nodes, and new nodes were allocated by popping them from the same list of free nodes. We suggest a different node allocation scheme which degrades the locality of nodes slower and incurs only a modest computational and memory overhead. The main idea is to split the global list of free nodes into several local lists, each containing all the free nodes that reside on the same virtual memory page.

When a node is deleted, it is pushed to the local freelist of the page that the node resides on. When a page contains over e percentages of free nodes, it is inserted to a global pool of underused local freelists. If a page becomes empty, it is returned to the operating system.

When a new node is inserted into the tree, it is allocated from the same local list than its parent. If the parent's freelist is exhausted, the new node is allocated from the nearest ancestors's freelist. If all local freelists from the root of the tree to the insertion point of a new node are exhausted, the new node is allocated from the first local freelist in the global pool. If the ratio of free nodes drops below l, the local freelist is removed from the pool. If the global pool is empty, a new page is requested from the operating system.

This strategy of allocating nodes will try to allocate child nodes on the same page with parent nodes, thus decreasing the number of distinct pages to be visited during traversing a search path. On the other hand, since not all free nodes are visible from the global pool, the new scheme may request new pages from the operating system if some local freelists would not be empty.

We performed similar tests to the ones presented in Section 4 in order to compare the affect of local freelists on degradation and memory consumption: we constructed a perfectly balanced and optimally located AVL-tree of 65535 nodes, repetitively deleted and inserted 10000 nodes, and measured the average number of TLB misses of searches after suitable intervals.

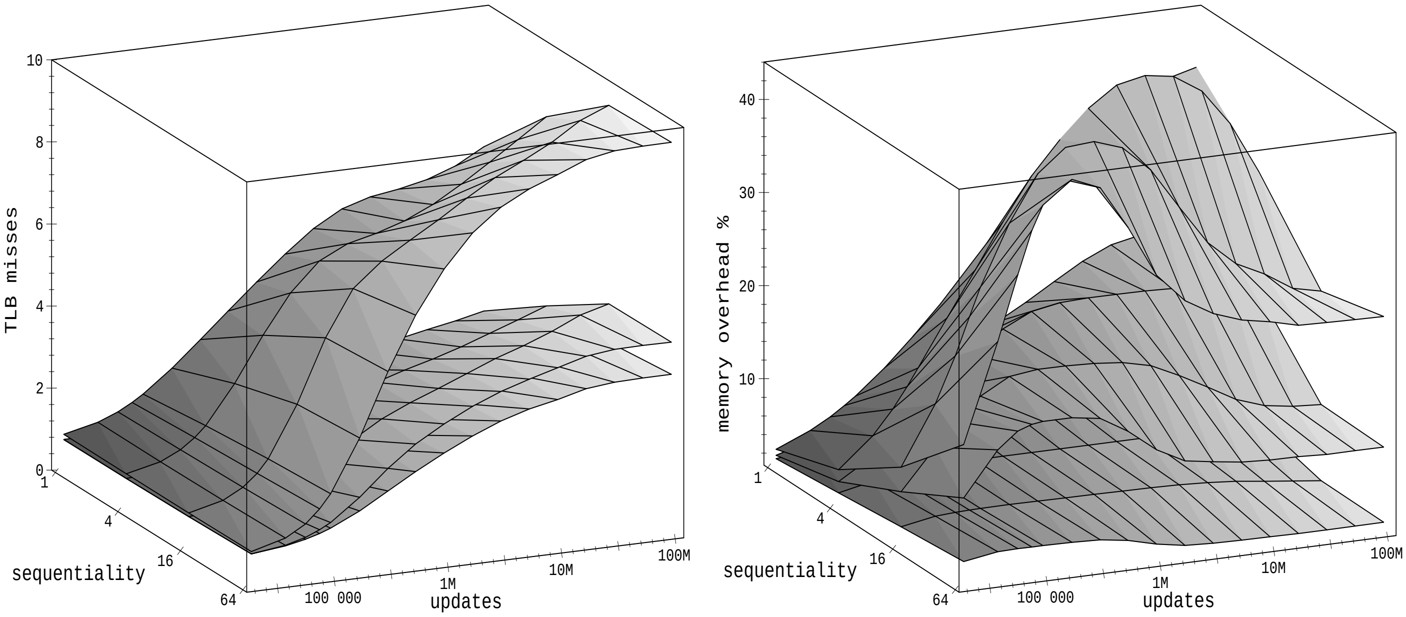


Figure 4: Left: Average number of TLB misses when searching in updated AVL-trees with a global freelist and local freelists $\langle e = 0.2, l = 0.1\rangle$ and $\langle e = 0.8, l = 0.5\rangle$. Right: Average memory overhead of AVL-trees with local freelists $\langle e = 0.8, l = 0.5\rangle$, $\langle e = 0.5, l = 0.3\rangle$, and $\langle e = 0.2, l = 0.1\rangle$. The TLB contains 64 entries, each covering 4 kB.

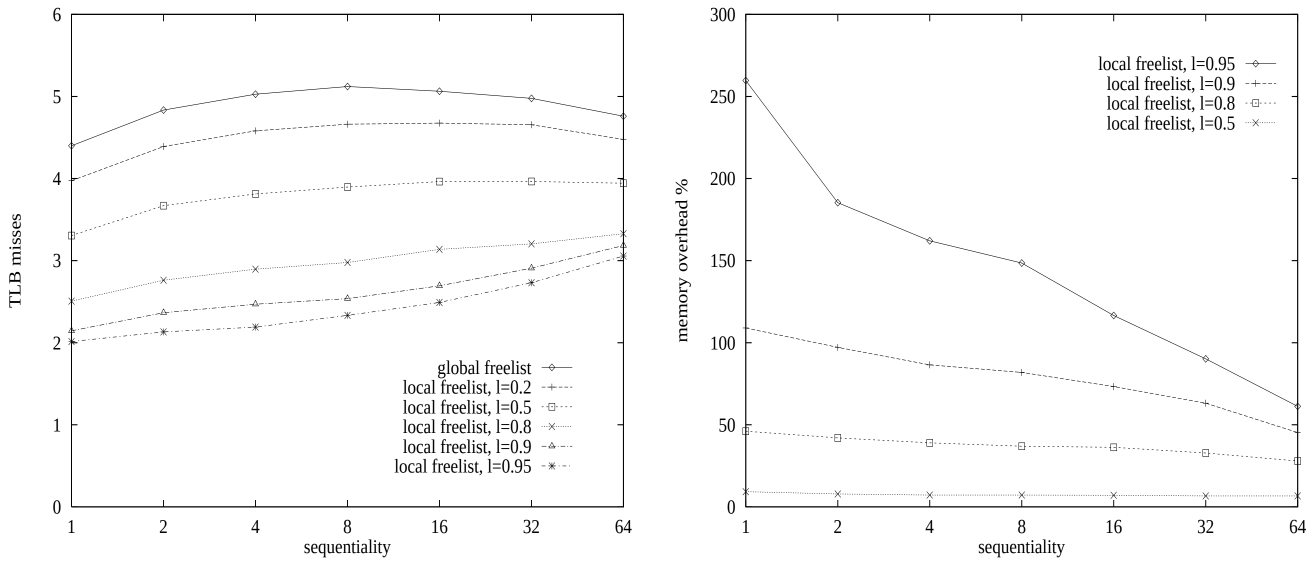


Figure 5: Left: Average number of TLB misses when searching in AVL-trees constructed from scratch. Right: Average memory overhead of the same trees.

Contrary to the experiments in Section 4, we did not measure the absolute search time, but simulated a 64-entry TLB with random replacement policy. The average number of TLB misses varied from a little over 0.75 in the optimally located tree to almost 10, which is close to an AVL-tree constructed from a shuffled freelist. We also varied the sequentiality of the updates so that the average length of sequences of successive keys to be inserted or deleted ranged from 1 to 64.

Using local freelists usually succeeds in at least halving the number of TLB misses. Increasing the limits e and l of entering and leaving the global pool of local freelists decelerates the speed of degradation, but it also increases the memory overhead of the tree. Nevertheless, as Figure 4 suggests, already a conservative choice of $l < e < 0.5$ provides for worthwhile decrease without considerable memory overhead.

We also measured the average number of TLB misses in trees constructed from scratch using either a global freelist of nodes in successive memory locations or using local freelists. This test is interesting because some applications may employ very short-lived trees where the relocation of all nodes could occupy a considerable part of the lifetime of the tree.

Contrary to the previous experiment where the size of the tree varies modestly, the locality of a rapidly growing tree improves considerably only when l is large[1]. But sufficiently large values of l also pose a considerable memory overhead. Fortunately the ratios e and l need not be constant. We suggest using large values, $l \approx 0.9$, when the tree is growing rapidly, but switching to small values, $e \approx 0.2, l \approx 0.1$, when the tree has become approximately half of its maximum size. If rotations are common, the limits e and l could be somewhat larger.

6 Conclusion

Memory reference locality is becoming an increasingly important factor in algorithm design. In this paper we have discussed the effect of memory reference locality on binary search trees and presented an algorithm to relocate a completely balanced binary tree in order to reduce the average number of TLB misses when operating in the tree. We also investigated the effect of updates on locality and presented means to manage allocation and deallocation of nodes so that updates disturb locality slower.

Applying the relocation algorithm periodically appears feasible when updates are infrequent, and small pauses can be tolerated. Where periodic relocation is infeasible, we are currently investigating hybrid algorithms which embed balanced trees, like AVL-trees, inside B-tree nodes [12]. B-trees provide for good TLB behavior, and AVL-trees provide for efficient insertion and deletion within the B-tree nodes.

Finally, we wish to propound that memory layout of binary search trees, and other data structures, must be considered with utmost care when designing fair experimental tests to compare algorithms.

[1] The value of e does not matter if we only insert.

References

[1] G.M. Adel'son-Vel'skii, Y.M. Landis, An Algorithm for the Organisation of Information. *Doklady Akademi Nauk* **146** (1962), pp. 263–266. English translation in *Soviet Math. Dokl.* **3**, pp. 1259–1263.

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[3] H. Chang, S.S. Iyengar, Efficient Algorithms to Globally Balance a Binary Search Tree. *Communications of the ACM* **27**:7 (1984), pp. 695–702.

[4] J.B. Chen, A. Borg, and N. Jouppi, A Simulation Based Study of TLB Performance, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 114-123.

[5] D. Comer, The Ubiquitous B-tree. *Computing Surveys* **11** (1979), pp. 121–137.

[6] L.J. Guibas, R. Sedgewick, A Dichromatic Framework for Balanced Trees. *Proc. of the 19th Symposium on Foundations of Computer Science* (1983), pp. 8–21.

[7] G. A. Hedayat, Numerical Linear Algebra and Computer Architecture: An Evolving Interaction, *Technical Report UMCS-93-1-5*, Department of Computer Science, University of Manchester.

[8] D.E. Knuth, The Art of Computer Programming **3**, *Sorting and Searching*, Addison-Wesley (1973).

[9] A. Moitra, S.S. Iyengar, Maximally Parallel Algorithms to Balancing Binary Search Trees. *IEEE Transactions on Computers* **34**:6 (1985), pp. 563–565.

[10] D. Nagle, R. Ulig, T. Stanley, S. Sechrest, T. Mudge, R. Brown, Design Tradeoffs for Software-Managed TLBs, *Proceedings of the 20th Annual International Symposium on Computer Architecture*, (1993), pp. 27-38.

[11] J. Nievergelt, E.M. Reingold, Binary Search Trees of Bounded Balance. *SIAM Journal of Computing* **2**:1 (1973), pp. 33–43.

[12] K. Oksanen, Memory Reference Locality in Binary Search Trees, Master's Thesis, Helsinki University of Technology, Department of Computer Science (1995).

[13] D.D. Sleator, R.E. Tarjan, Self-adjusting Binary Search Trees. *Journal of the ACM* **32** (1985), pp. 652–686.

[14] A.J. Smith, Cache Memories, *ACM Computing Surveys*, **14**:3 (1982), pp. 473–530.

[15] Q.F. Stout, B.L. Warren, Tree Rebalancing in Optimal Time and Space. *Communications of the ACM* **29**:9 (1986), pp. 902–908.

[16] M. Talluri, M. Hill, Surpassing the TLB Performance of Superpages with Less Operating Systems Support, *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, (1994), pp. 171–182.

[17] P.R. Wilson, M.S. Lam, T.G. Moher, Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, (1991), pp. 177-191.