Performance Analysis of BSTs in System Software

[Extended Abstract]

Ben Pfaff
Stanford University Department of Computer Science
blp@cs.stanford.edu

Categories and Subject Descriptors: E.1 [Data Structures]: trees; E.2 [Data Storage Representations]: linked representations; C.4 [Performance of Systems]: performance attributes

General Terms: algorithms, performance

Keywords: binary search tree, BST, threaded tree, AVL tree, red-black tree, splay tree

1. INTRODUCTION

OS kernels often use binary search tree (BST) based data structures. Choosing the right tree and node representation can significantly impact the performance of code that uses these data structures. Surprisingly, there has been little empirical study of the relationship between the algorithms used for managing BST-based data structures and performance characteristics in real systems [6, 2, 10]. This extended abstract outlines performance results for 20 total BST variants: four data structure variants each with five different node representations.

2. DATA STRUCTURES

In an ordinary BST, insertion of data items in a pathological order, such as sorted order, causes performance to drop from $O(\lg n)$ to $O(n)$ per operation in a n-node tree [7]. One solution is a balanced tree such as an *AVL tree* or *red-black tree*, which limit the height of an n-node tree to no more than $1.4405 \lg(n+2) - 0.3277$ and $2 \lg(n+1)$, respectively [1, 5]. Another popular solution is a *splay tree*, which rotates or "splays" each node to the root at time of access [9].

A minimal BST node contains a data item and left and right child pointers, but efficient in-order traversal in such an *ordinary BST* requires maintaining a stack, which must usually be rebuilt if the tree is modified during traversal. One way to eliminate the stack is to add a parent pointer to the node, yielding a *tree with parent pointers*. Adding predecessor and successor pointers to each node, yielding a *linked list tree*, is another solution. Alternatively, we can use right-child pointers that would otherwise be null to store a pointer to the node's successor, yielding a *right-threaded tree*. If we also use left-child pointers that would otherwise be null to store predecessor nodes, the result is simply a *threaded tree*.

SIGMETRICS/Performance'04, June 12–16, 2004, New York, NY, USA.
ACM 1-58113-664-1/04/0006.

3. EXPERIMENTS

We ran three experiments to test the 20 variations on binary search trees described in the previous section.

3.1 VMAs

Our first experiment measures the speed of managing a set of virtual memory areas (VMAs), which represent sections of memory in a Unix process. A normal process has at least three VMAs, for its code, data, and stack segments, but processes can create an arbitrary number with the `mmap` system call. Because VMAs vary in size from 4 kB to over 1 GB, hash tables are not an appropriate representation. Most Unix-like OSes use a BST variant to keep track of VMAs, as does Windows NT for its VMA equivalents [4].

We recorded VMA activity in three real programs: Mozilla 1.0, VMware GSX Server 2.0.1, and squid running under User-Mode Linux 2.4.18.48. The three test sets display very different VMA behavior, but they all display strong locality. We used these as test sets in a simulation of VMA activity. We ran the simulation for each test set with each of our 20 BST variants.

Splay trees were the big winner for all three data sets, bettering the best of the competition by 23% to 40% each time, due to the splay tree's ability to optimize the tree for locality of reference. AVL trees were consistently faster than red-black trees, by up to 20%. AVL trees also made up to 32% fewer comparisons, suggesting that the stricter AVL balancing rule is responsible. However, the average internal path length for the red-black trees was only about 3% longer than for the AVL trees. We conclude that AVL and red-black trees globally balanced the trees about as well, but the AVL balancing rule leads to better balancing in the important places for test sets like these with strong locality.

Ordinary BSTs were the slowest node representation in all cases but two. Within the balanced tree implementations, plain nodes were slowest, dragged down by rebuilding stacks after tree modifications, and parent pointers were fastest, beating out threaded representations because of their extra overhead in distinguishing child pointers from predecessors or successors.

3.2 Internet Peer Cache

RFC 791 [8] requires that each IP packet sent by an internet host be stamped with a 16-bit identification field that "must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system." The Linux kernel ensures this by maintaining an AVL tree of per-host counters, incrementing a host's counter

once for every packet transmitted to it. A hash table would be faster in the average case, but an attacker could cause hash collisions that would degrade its performance. Universal hashing [3] could make such attacks more difficult, but they are impossible in a balanced tree.

The peer AVL tree is used as a second-level cache. The first-level cache is the route cache, each entry in which contains a pointer to the corresponding peer AVL tree node. When an entry in the route cache is dropped, its corresponding node in the peer AVL tree remains for some amount of time before it is dropped. On the other hand, if a fresh route cache entry is added for the IP address during that time, the corresponding peer AVL tree node will be retained.

We wrote a simulation that attempts to reproduce heuristically the most important features of the route cache. We experimented with its behaviors in two scenarios, trying all 20 BST variants in each. The first scenario was a web server under "normal" load, in which hosts with random IP addresses make on average 9 requests at intervals of 30 s on average. For this data set, typically parent pointer and threaded nodes were fastest, reflecting that these types of nodes allow for $O(1)$ deletion. Within node representations, speed was consistently, from fastest to slowest, ordered as unbalanced trees, red-black trees, AVL trees, and splay trees, reflecting the extra work that each kind of tree performed trying to balance already random data.

The second scenario was an "attack" in which consecutively numbered IP addresses make a new request every 10 ms. For this case, AVL trees performed better than red-black trees because the stricter AVL balancing rule does a better job of keeping the tree height at a minimum at the important point, the point of the next insertion.

3.3 Cross-Reference Collator

The first two experiments dealt with problems that were best solved with BSTs. However, in practice problems that are better solved with hash tables are sometimes solved with BSTs anyhow. For example, the C++98 standard library provides (typically) BST-based "set" and "map" template classes, but no hash-based template classes, encouraging C++ programmers to use the wrong tool.

With this in mind, we wrote a "cross-reference collator" tool that sorts and merges a set of identifiers extracted from program source code using BSTs as an index. We ran the tool with all 20 BST variants on the set of files in include/linux from Linux 2.4.18. To test worst- and best-case orders, we also ran it on sorted and randomly shuffled collections of identifiers from the same test set.

For the normal case, splay trees were consistent winners, taking advantage of locality, resulting from the tendency of identifiers to recur within a few lines of first mention. Red-black trees are consistently slightly faster than AVL trees, reflecting the more relaxed red-black balancing rule that results in less rebalancing. Finally, the times for unbalanced trees are only at most 11% greater than for AVL trees, demonstrating that the natural identifier ordering tends toward randomness.

Splay trees were also the big winners for the sorted case, again because of their advantages for locality. AVL trees performed better than red-black trees because the stricter balancing rule maintains a shorter maximum path length.

For the shuffled case, trees that did less extra work were again the winners. However, the shuffled results were consistently slower than the corresponding "normal" results. We found that lack of locality in the shuffled results, combined with a processor cache of limited size, was the culprit.

4. CONCLUSION

This extended abstract summarized our empirical comparison of the performance of 20 BST variants in three different real-world scenarios, and demonstrated that choice of data structure can significantly impact performance. Our results show that BST data structures and node representations should be chosen based on expected patterns in the input and the mix of operations to be performed.

We found that in selecting data structures, unbalanced BSTs are best when randomly ordered input can be relied upon; if random ordering is the norm but occasional runs of sorted order are expected, then red-black trees should be chosen. On the other hand, if insertions often occur in a sorted order, AVL trees excel when later accesses tend to be random, and splay trees perform best when later accesses are sequential or clustered.

For node representation, we found that parent pointers are generally fastest, so they should be preferred as long as the cost of an additional pointer field per node is not important. If space is at a premium, threaded representations conserve memory and lag only slightly behind parent pointers in speed. A plain BST has fewer fields to update, but combining traversal and modification requires extra searches. Maintain a linked list of in-order nodes is a good choice when traversal is very common, but exacts a high memory cost. Finally, right-threaded representations fared poorly in all of our experiments.

5. ACKNOWLEDGEMENTS

A Stanford Graduate Fellowship supported this work.

6. FULL PAPER

Visit `http://benpfaff.org/papers` for the full paper summarized by this extended abstract.

7. REFERENCES

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.

[2] J.-L. Baer and B. Schwab. A comparison of tree-balancing algorithms. *Communications of the ACM*, 20(5):322–330, 1977.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, section 12.3.3, pages 229–232. MIT Press, 1990.

[4] H. Custer, editor. *Inside Windows NT*, page 200. Microsoft Press, 1993.

[5] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[6] P. L. Karlton, S. H. Fuller, R. E. Scroggs, and E. B. Kaehler. Performance of height-balanced trees. *Communications of the ACM*, 19(1):23–28, 1976.

[7] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*, section 6.2.2, pages 430–31. Addison-Wesley, Reading, Massachusetts, second edition, 1997.

[8] J. Postel. RFC 791: Internet Protocol, Sept. 1981. Status: STANDARD.

[9] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[10] W. E. Wright. An empirical evaluation of algorithms for dynamically maintaining binary search trees. In *Proceedings of the ACM 1980 annual conference*, pages 505–515, 1980.