Conflict Avoidance:
Data Structures in Transactional Memory

Lucia Ballard

May 3, 2006

Abstract

The Dynamic Software Transactional Memory system lets multiple threads safely access data through atomic transactions. Conflicts between transactions require one to abort, incurring a high performance cost. By managing the likelihood that actions will produce conflicts, data structure design greatly impacts the efficiency of transactional programs. I analyze a red-black tree and a binomial heap, using both the classic algorithms and delayed-rebalancing versions which enclose the rebalancing step in a separate transaction. Delaying the rebalancing step of these structures dramatically improves their performance by minimizing overlapping claims to transactional objects.

1 Introduction

As multiprocessors become common, the ability to coordinate the actions of multiple threads becomes more and more important. Traditional locking mechanisms address the issue, but can easily introduce race conditions or deadlocks into otherwise consistent code. In contrast, the Dynamic Software Transactional Memory (DSTM) system introduced by Herlihy et. al. [3] allows multiple threads to modify multiple objects atomically without explicitly locking them.

If more than one thread simultaneously attempts to modify the same object, one must abort its attempt and retry, a costly endeavor.

Consequently, the design of the data structures used in a transactional system becomes of great importance. An entirely new area of optimization is opened up. More than the running time of an algorithm, the number of conflicts it encounters determines its efficiency. Data structures can concentrate modifications in one area, inducing conflicts, or spread modifications among disparate areas, minimizing conflicts.

In this paper, I analyze a red-black tree and a binomial heap, both self-balancing data structures whose algorithms can be split into multiple transactions. These changes compromise the data structures' invariants and extend their worst-case running time. But the performance gain by avoiding conflict more than compensates for the loss in sequential efficiency.

2 Locking vs. Transactions

Both locking and transactions provide a way to safely access a data structure from multiple threads, but choosing one over the other brings various costs and benefits. First, they differ greatly in ease of use. Locking is notoriously tricky to use well. Coarse-grained locking is conceptually simple, but introduces a sequential bottleneck around the data structure in question. Fine-grained locking permits multiple threads to access the data structure, but requires great care in design to avoid introducing deadlocks or race conditions. Every change to the data structure could invalidate the locking system, making the structure hard to maintain.

In contrast, correctly using transactional memory is trivial from the programmer's standpoint. Mutable objects in the shared data structure are instantiated as *transactional objects* from a factory. Threads can read or write these objects in any order they please, as long as they bracket their actions in a transaction. When the transaction has finished, it either commits if there has been no conflict, or aborts and retries automatically if another thread has modified one of the objects involved. A separate module, the *contention manager*, chooses which of two conflicting threads to abort. The programmer may write the contention manager from scratch or choose from a variety of defaults provided by DSTM.

When it comes to speed, transactions no longer have the advantage. When threads rarely contend for the data structure, both locking and transactions incur a small fixed cost. Threads asking for mutexes must generally make a system call to acquire and release the lock. Threads committing transactions must make a private copy of the object's data and adjust the object's internal state.

When contention is high, locks generally outperform transactions. Threads that fail to acquire a lock will queue up behind it, but don't lose any progress. In contrast, threads whose transactions are aborted must discard their copies of every object they modified and re-execute the entire transaction. Also, depending on the contention manager chosen, threads may end up cyclically preempting each other as each transaction retries and reopens the same object for modification. The trading back and forth of mutual preemptions magnifies the cost of contention and presents a major hurdle to system performance.

Scherer et. al. [5] investigated the design of contention managers and the problem of finding the most efficient algorithms for choosing which threads to abort. But the cost of conflict remains high, even with well-designed contention managers. This paper examines the issue from a different perspective: how to avoid the conflicts in the first place.

3 Conflict Avoidance

3.1 Set Overlap

Each thread running a transaction in DSTM tracks the objects it has read and modified in a *read set* and *write set*, respectively. Conflicts occur only when one thread modifies an object that another thread has either read or modified.

When a thread attempts to commit a transaction, it searches through each of its sets to detect modifications in its read set or reads in its write set. Data structures that keep each thread's sets from overlapping will necessarily reduce conflict among threads.

Additionally, some data structures have objects, such as root nodes, that must be read for nearly every transaction. Modifying such an object can invalidate many transactions at once. Algorithmic changes that avoid writes to such commonly-read nodes, or that find ways to avoid opening those nodes for reading, should also improve performance. It is not simply set size that determines the amount of contention, but also the degree of mixing between reads and writes.

3.2 Delayed Rebalancing

Many popular data structures are self-balanced to improve efficiency. As items are added or deleted to trees, for example, one branch of a tree may grow much longer than another, so searches may take much longer than logarithmic time. The tree still performs correctly. But adding a rebalancing operation after each insertion or deletion improves the tree's efficiency.

The cost of maintaining balance is increased complexity for each operation, which often requires a thread to access a large number of nodes. The added complexity can constrain the number of concurrent threads that the data structure can support, creating a bottleneck. Delayed rebalancing often releases this bottleneck and improves performance in concurrent situations.

In delayed rebalancing, operations such as inserts or deletes are divided into two phases: the initial modification and the rebalance. Once a thread has completed the initial insertion or deletion, it can return and report success. Later, it or another thread can restore the data structure to a balanced state. This strategy compromises the worst case running time, since many modifications could pile up and warp the tree structure. But, as discussed by Hanke [2], it can make data structures more robust. Since changes often arrive in bursts, delayed rebalancing allows a data structure to accommodate busy periods and wait for slower periods to tidy up.

Delayed rebalancing should be even more important in a transactional memory environment. By splitting one complex operation into several simpler ones, it reduces the number of objects accessed in each transaction. Threads will be less likely to modify any particular node. However, the strategy is not without costs—the transaction machinery must run several times for each logical operation on the data structure. The following experiments were conducted to analyze the costs and benefits to overall performance of delayed rebalancing.

4 Red-Black Trees

4.1 Data Structure Overview

Red-black trees are a classic benchmark in the study of concurrency. They maintain roughly even depth among their branches, preventing the run-time degradation that slows down unbalanced search trees. They are also tricky to lock properly as operations run both up and down the tree, possibly entering a deadlock. Even in a transactional environment, where deadlock is impossible, they make for an interesting test case by allowing delayed rebalancing.

A red-black tree is a sorted binary tree with the following properties [1]:

1. Every node is colored *red* or *black*.
2. The root node is black.
3. The leaf nodes are black.
4. No red node has a red parent.
5. The number of black nodes encountered on every path from any internal node to the leaf nodes is equal.

Insertions work by finding the leaf node at the proper place in the tree, and replacing it with a new red node holding the new key and two more leaf nodes.

If the new node's parent is also red, property 4 is violated and the tree must be rebalanced. The rebalance may rotate the nodes and terminate, or it may recolor nodes and propagate the property violation up to the node's grandparent. In this way, some rebalances modify a chain of nodes from the fringe all the way up to the root. For a full description, see the appendix.

4.2 Modifications

Since the insertion operation is easily split into an initial insert and a subsequent rebalance, red-black trees are well suited to the delayed rebalancing strategy. A thread will first insert its new node at the fringe of the tree and mark it for later rebalancing. In some other transaction, it will re-examine the node and fix up the tree, restoring the red-black properties. The interval between transactions temporarily compromises the balance of the tree, so that other inserts or lookups may take longer than $O(log(n))$ time.

It can be difficult to track which nodes need rebalancing. As explored by Larsen [4], relocating the nodes that have been marked for updates is non-trivial, since nodes may have moved around the tree through other rotations. Also, other rebalancing operations may have rotated or recolored the neighborhood and rendered the requested update obsolete.

The simplest solution—lazy rebalancing—proved effective. Rather than clean up after an insert, each thread cleans up imbalances it encounters on its way to insert. As the thread traces down through the tree, should it encounter a red parent-child pair, it puts aside its pending insert and performs the rebalance. Then, in a new transaction, it starts again from the root and searches down to the fringe. If it finds the correct leaf node without encountering any imbalances, it performs the insert and finishes.

This algorithm will never clean up all the imbalances of the tree. It also extends the running time of an insert from $O(log(n))$ to $O(log^2(n))$, as a thread may trace from root to leaf repeatedly. But it eliminates the need for extra tracking mechanisms for marked nodes, which introduce new areas of contention and add significant complexity to the rebalancing operation.

This relaxed red-black tree does not prevent starvation of a single thread—for that matter, neither does the underlying library. However, each operation the thread makes, whether a rebalancing or an insert, is making progress. If the thread must stop to rebalance at a node, it has verified the correctness of the nodes above, so the rebalance operation must move the tree closer to a balanced state. This implies that rebalancing threads cooperate—the work done by one thread helps all others towards their inserts. This cooperation makes the tree obstruction-free.

4.3 Experimental Setup

The RedBlackTree benchmark inserts various integers into a standard red-black tree, using one transaction for each insertion. The RelaxedRedBlackTree inserts integers into a tree that delays rebalancing, spending several transactions on each insertion. For comparison, the UnbalancedRedBlackTree never performs a rebalancing step: it simply leaves the nodes in the order it inserted them. If rebalancing is unnecessary, it should outperform both red-black tree variants.

The test ran on a four-processor machine using the default contention manager, an exponential-backoff algorithm [5]. Each benchmark ran for 20 seconds each, doing an insertion or a lookup for a random integer with equal probability, to simulate the mix of reads and writes that a data structure might receive (varying the probability of running an insert, anywhere between 20% and 100%, had little effect on the relative performance of benchmarks, so only one set of data is presented here). The benchmarks were tested with simultaneous access by a number of threads ranging from 1 to 60.

The performance of each benchmark depends heavily on the distribution of values to insert. If the inputs are randomized, rebalancing does not bring much benefit—the tree stays balanced naturally. If they are inserted in order, an unbalanced tree will degrade to the linear running time of a linked list. Various

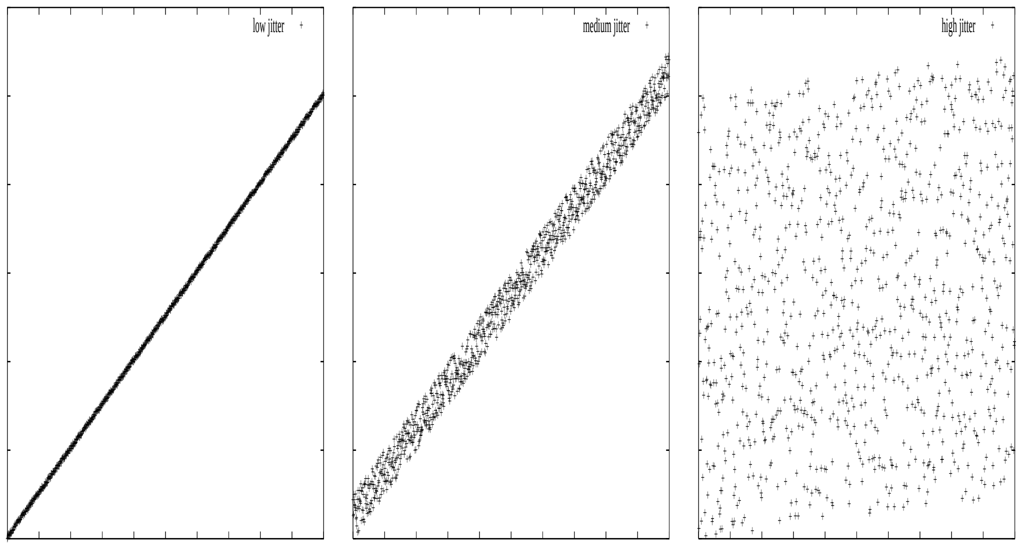

Figure 1: Low, medium, and high jitter. A higher jitter value gives a close-to-random distribution, while a low value yields in-order data.

degrees of randomness ought to bring about different results. On each loop, each thread inserted the value $k + r$, where k was the current loop increment and r a random value between 0 and some maximum, called the *jitter value* (see figure 1). Low jitter leaves k essentially in order; high jitter outweighs k and provides nearly random data. Each benchmark was tested with four jitter values: 1 (in-order), 100, 10,000, and 1,000,000 (highly randomized).

The data presented represent the average over eight full test runs. Two statistics were tracked: the percentage of transactions that committed, and the rate of successful operations. Commit percentage directly measures the ability of the data structure to avoid the cost of the abort-retry sequence. But the RelaxedRedBlackTree uses several transactions for each operation, so simply counting the number of commits overstates how many operations it performs. The number of operations per second records the overall throughput of the data structure.

4.4 Results

As conjectured, smaller transactions do lead to higher commit percentages (see figure 2). RedBlackTree, which needs up to $O(log(n))$ nodes per insert, loses more and more transactions to conflicts as concurrency rises. UnbalancedRedBlackTree only modifies one node at a time, so it can maintain almost perfect commit rates. The RelaxedRedBlackTree, by contrast, is very sensitive to randomness. Each thread modifies at most seven nodes per transaction, but the amount of randomness determines whether those nodes are also present in other thread's write sets. With more ordered input, every thread is concentrated in one small part of the tree, so each rebalance will likely conflict with the others. Contention is much lower with randomized inputs that let each thread operate in a different area of the tree.

If only commits mattered, then it would never be worthwhile to rebalance. But commit percentage does not correlate with the length of transactions or the number of transactions per operation. When overall throughput in considered, the strengths of delayed rebalancing become apparent.

The contrast between RedBlackTree and UnbalancedRedBlackTree highlights the tradeoffs of balancing. Each rebalance increases transaction size, generating conflict and slowing down the insert, but also fends off the degradation of the tree into a linked list. As expected, UnbalancedRedBlackTree outperforms the RedBlackTree with randomized input; with ordered input, the outcomes are reversed.

RelaxedRedBlackTree combines the strengths of both. It keeps transaction sizes small when the inputs are random, since most writes occur towards the tree's fringe and are not dependent on higher and more popular nodes. It also maintains a balanced tree when the inputs are ordered. The benchmark performed decently under all jitter conditions, and excelled in the mixed case, when the inputs were randomized but trended upwards.

However, there is significant overhead associated with delayed rebalancing. The tree can become lopsided, lengthening the path some threads must traverse;

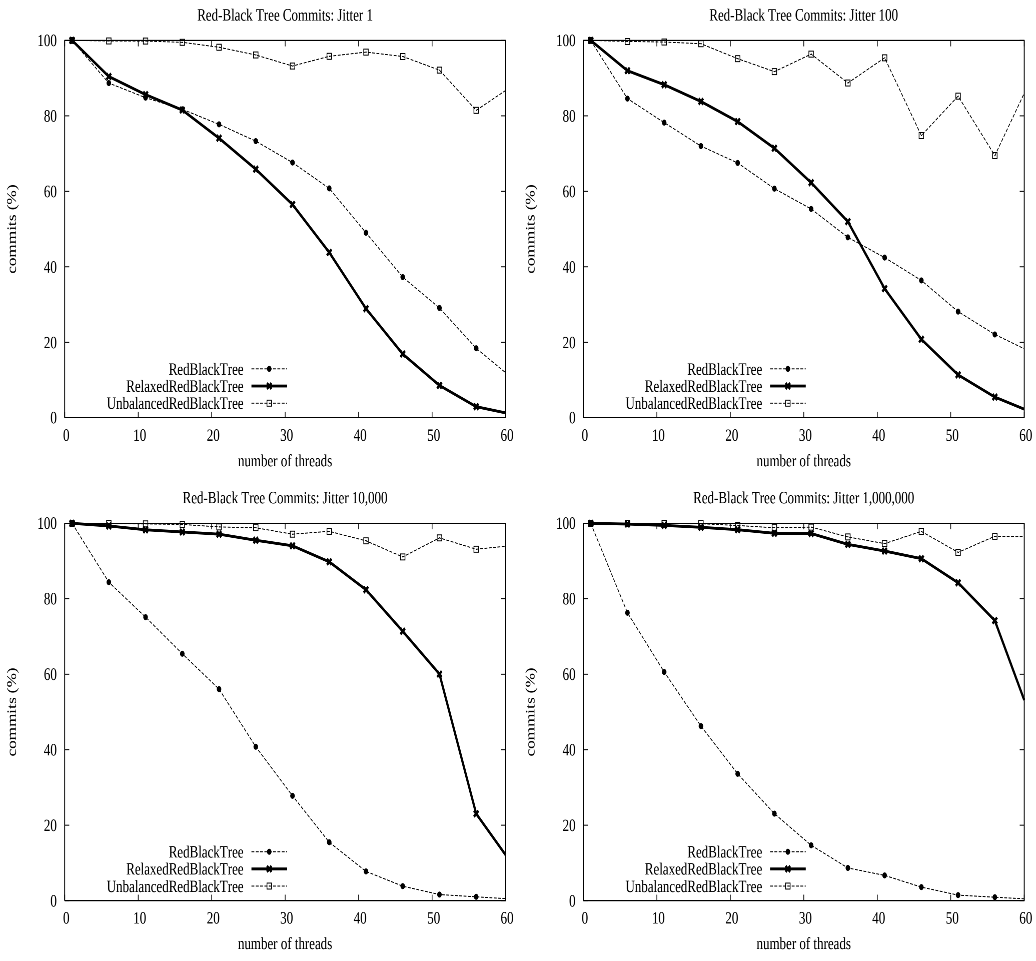

Figure 2: Percentages of red-black tree transactions that commit. For both RedBlackTree and UnbalancedRedBlackTree, commit percentage stays steady relative to the randomness of the input. But for RelaxedRedBlack-Tree, increased randomness greatly improves its commit rate.

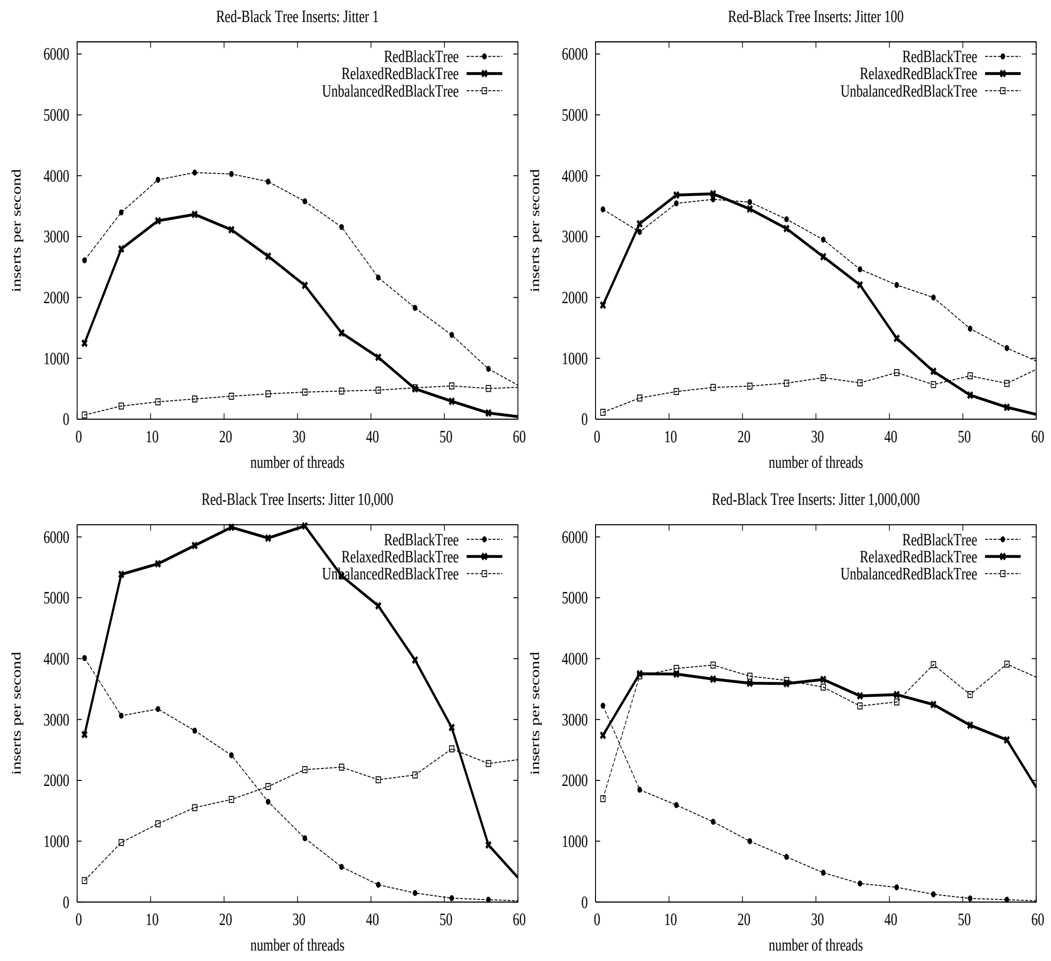

Figure 3: Red-black tree throughput for increasingly random inputs. RELAXED-REDBLACKTREE's performance does not significantly degrade for ordered input, and improves as randomness increases.

each time an imbalance is found, the thread must fix it and restart, reduplicating the traversal from the root down; and there is the cost of opening and committing more transactions. These factors keep RelaxedRedBlackTree's throughput lower than RedBlackTree under ordered input, when rebalancing is most crucial.

Even with extremely random data, the overhead of delayed rebalancing did not slow the RelaxedRedBlackTree down relative to the UnbalancedRedBlackTree. A qualitative look at the state of the RelaxedRedBlackTree may explain its surprisingly good performance. In the top layers of the tree, which are often traversed, the tree looked like a standard red-black tree. Towards the fringe, there were more red nodes than black nodes, as threads were unlikely to follow and fix up any particular path. The rebalancing operations were most effective in the area where the most threads depended on them—the top of the tree. Thus the rebalancing operations helped a great number of threads, maintaining a high level of throughput.

However random or ordered the data, if only one thread was at work, the RedBlackTree always performed best. Even if the inputs practically balanced themselves, the investment in perfect rebalancing paid off in the sequential case. But as soon as contention was introduced, the situation changed. Tree balance matters, but conflict avoidance matters more.

5 Binomial Heaps

5.1 Binomial Heap Overview

The binomial heap collects a series of keys and efficiently finds the object with the lowest-valued key. Unlike simpler heaps, which provide only insert and extract-minimum operations in logarithmic time, the binomial heap can also merge with another heap in $O(log(n))$ time. Like the red-black tree, it is a self-balancing data structure. Its algorithmic complexity makes it an interesting test case for delayed rebalancing and other conflict avoidance strategies.

The heap consists of a linked list of heap-ordered binomial trees, where each binomial tree of depth n consists of two trees of depth $n-1$ linked together. When the heap is in a balanced state, there is only one tree of each depth, or degree, in the list. An item is inserted by creating a new tree of degree zero—just a single node—and adding it to the front of the list. The heap then runs the *union* operator on the list, uniting any pair of trees of degree k into a tree of degree $k+1$. When the union completes, there is again only one tree of each degree. To extract a node, the heap is searched for the tree with the smallest key at the root node; that tree is split into its component trees; and the subtrees are merged back into the main heap. For more details on the workings of binomial trees, see the appendix.

5.2 Modifications

The binomial heap lends itself to delayed rebalancing. Both the insert and extract-minimum operations can be halted at a middle point, leaving the heap intact, but with potentially more than one tree of any particular degree. This imbalance doesn't affect the correctness of the tree or the running time of an insertion. But extractions could suffer, as many more nodes must be searched to find the tree with the lowest key. This cost, as well as the inherent cost of running more transactions, may outweigh the benefits of splitting transactions into smaller chunks. The BinomialHeap runs the standard algorithm, while the RelaxedBinomialHeap uses delayed rebalancing.

The binomial heap also suffers from a transactional "hot spot." The first tree on the heap's list is modified during every insert and almost every union. Unlike the red-black tree, whose root is far more often read than written, the first node of a binomial heap is in almost every thread's write set. Even if a union doesn't modify the first node, the node has been read, so the transaction will have to abort if the node changes. With any significant number of threads performance quickly degrades. This single node amounts to a sequential bottleneck.

To ease the pressure on this hot spot, the linked list could be modified.

ArrayHeap splits the list into several linked lists in an array, one list for each degree of tree up to some maximum. The insertion and extract-minimum operations in the ArrayHeap work analogously to the standard binomial heap, adding trees to the front of the appropriate list in the array. The union, however, can act on each list separately. If the list's length is greater than one, union combines pairs of trees and adds them to the next higher list. This way, unions can proceed independently in different parts of the array. The ArrayHeap benchmark runs each operation, including a full union, in one transaction. A delayed-rebalancing version, the RelaxedArrayHeap, has the same underlying structure as the ArrayHeap, but does each insertion, extract, and step of the union as a separate transaction.

5.3 Experimental Setup

All four versions of the binomial heap ran for twenty seconds at a time on a four-processor machine. These runs were repeated for increasing numbers of threads from 1 to 48. Each thread executed a loop that inserted a random item with probability $\frac{3}{4}$ and extracted the minimum item with probability $\frac{1}{4}$. Mixing the two operations allowed the tree to grow with time but still accounted for the performance problems with extract-minimum. These benchmarks, like the red-black tree, also gathered statistics on both commit percentage and total rate of operations. The data presented is the average of eight full tests.

5.4 Results

Relaxing the standard binomial heap had little effect on its commit percentage. Although each transaction involved fewer steps, those steps still all accessed the very front of the linked list, and so could easily conflict with each other. Both heaps' performance degraded quickly with more threads running.

The normal array heap suffered from the same problem, and its commit percentage closely tracked the BinomialHeap and RelaxedBinomialHeap. But the relaxed version maintained a much higher percentage of commits—since

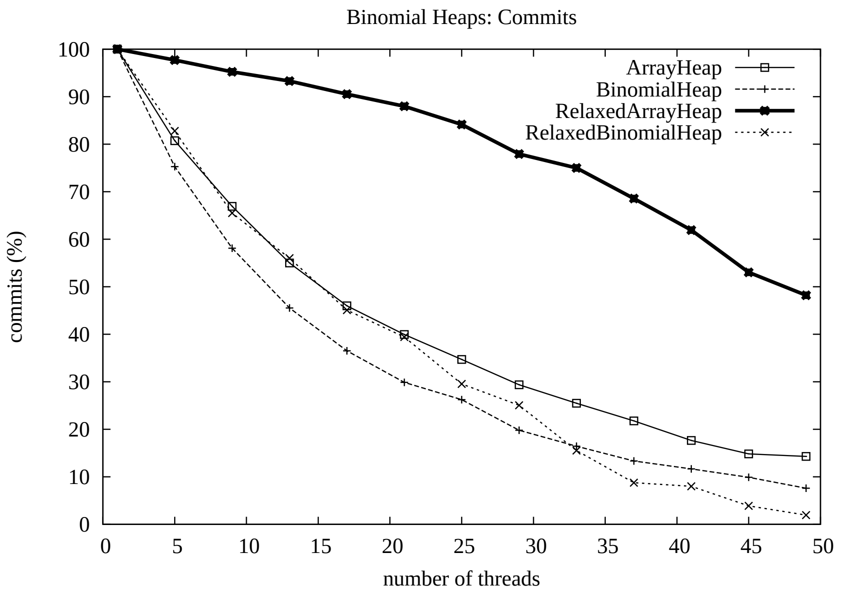

Figure 4: Commit percentage for each heap. By reducing conflicts on the degree-zero node, RELAXEDARRAYHEAP maintains a significantly higher commit ratio than any other heap variant.

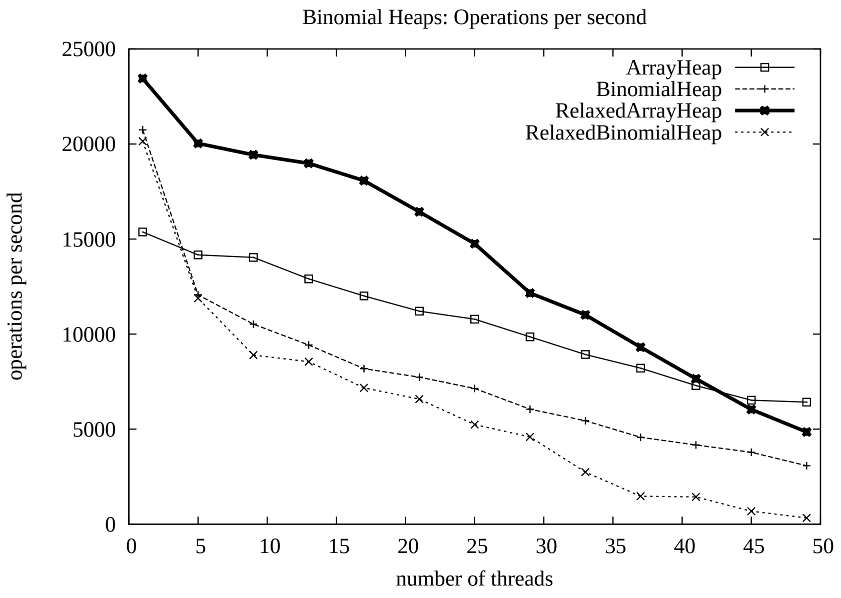

Figure 5: Rate of operations on each binomial heap. RELAXEDARRAYHEAP maintains a performance advantage, but it wanes as the level of conflict rises.

each transaction looked at a single linked list of some degree, several unions could proceed in parallel without conflicting.

Throughput resembled the commit performance. Even with the overhead of beginning and committing a new transaction for every stage of a union, the RelaxedArrayHeap maintained a steady lead over all other versions of the binomial heap in operations per second. But as concurrency rose, contention on the first list of trees was still too high to allow for much parallelism. Still, both array heaps outperformed both standard heaps.

The RelaxedBinomialHeap had the worst overall rate of operations. Delayed rebalancing added the overhead of many transactions, but didn't prevent contention on the first node. This shows that delayed rebalancing is no panacea, and it is not simply transaction length that determines performance—the strategy only works if it limits the number of objects involved in each transaction.

6 Conclusion

In addition to the questions of worst-case running time and parallelism, data structures in a transactional environment have another dimension critical to their efficiency—the degree of conflict they induce. Contention depends on which objects are modified during each operation. Smaller sets of objects, and those that tend to be dispersed throughout the structure, provide fewer chances for conflict. The sacrifices in worst-case running time and balance inherent in delayed rebalancing are well compensated by the efficiency gain through conflict avoidance. But applying delayed rebalancing indiscriminately may not give much benefit: the greatest performance boosts come from keeping highly contested nodes out of some transactions entirely.

Further research into contention management could lessen the importance of conflict avoidance by minimizing the high cost of contention. In the meantime, there is much room for improvement among the algorithms presented. The relaxed red-black tree could be extended to include deletions, and the binomial heap could be further fractured to reduce conflict on the first tree. Overall, it is clear that minimizing the number of nodes involved in each transaction should be the top priority for data structure design in a transactional system.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, first edition, 1990.

[2] Sabine Hanke. The performance of concurrent red-black tree algorithms. *Lecture Notes in Computer Science*, 1668:286–??, 1999.

[3] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[4] Kim S. Larsen. AVL trees with relaxed balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000.

[5] W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of Distributed Computing*, 2005.

APPENDIX

A Red-Black Tree Rebalancing

When a newly inserted red node n's parent is also red, a red-black tree requires rebalancing. The particular steps taken depend on the configuration of the tree.

1. If n's uncle (the parent's sibling) is also red, the nodes may be recolored. n remains red; its parent and uncle are colored black; and the grandparent is colored red. This preserves the equal-depth property from the grandparent—each path from it to the fringe encounters the same number of black nodes. Now that the grandparent is red, it may also have a red parent, and so the rebalancing algorithm must be called on it.

2. If n's uncle is black, a rotation must occur. Assume the parent is the left child of the grandparent. If n is the right child of the parent, it must be rotated left.

3. Now n is aligned with the parent. Perform a right rotation on the grandparent. This brings the red parent to the top, with the red child and black grandparent as its left and right children. Recolor the parent black and the grandparent red. Now all the red-black properties are satisfied and the rebalancing is finished.

Steps two and tree are reflected left-to-right if the parent is the right child of the grandparent.

B Binomial Heaps

A binomial heap is a linked list of binomial trees, each tree in heap order (that is, every node's value is the minimum of all the values below it). The list is sorted by the degree of each tree, and contains only one tree of each degree. Every tree of degree n consists of two linked trees of degree $n - 1$, with one

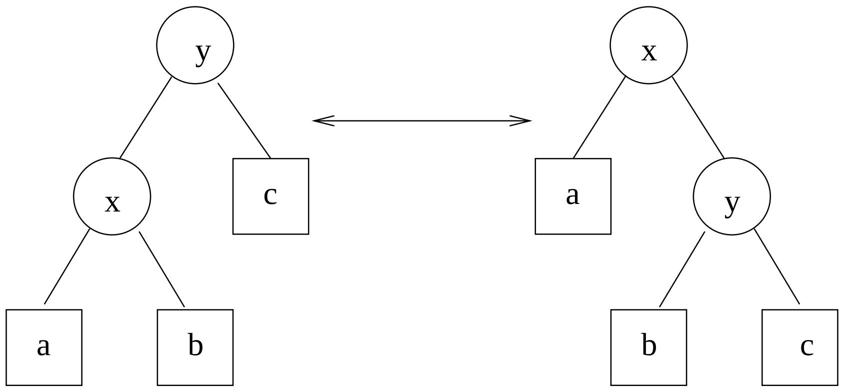

Figure 6: Left and right rotations on a red-black tree. A, b, and c are unmodified, and the left-to-right order of every node is preserved.

tree as the first child of the other. Thus, there are $\binom{n}{k}$ nodes at depth k. This property gives binomial heaps their name.

To insert, a new tree of degree zero is created containing the key to be inserted. The tree is added to the front of the list. Then a union operation is performed.

To extract the minimum key, the list is searched for the tree with the lowest key. That tree is removed, and the value at the root node is saved. Each child of the root node is also a binomial tree in heap order: these child trees are merged back into the heap's list of trees, and then a union is performed. Finally, the value at the root node is returned.

The union operation works as follows: the input is a list of trees where there are potentially more than one tree of any degree. Each tree in the list is considered sequentially. If the tree after it has the same degree, the two trees are linked: the tree whose root node has the lowest key adds the other tree as its first child. Now they form a new tree of the next higher degree, which is reinserted in the list.

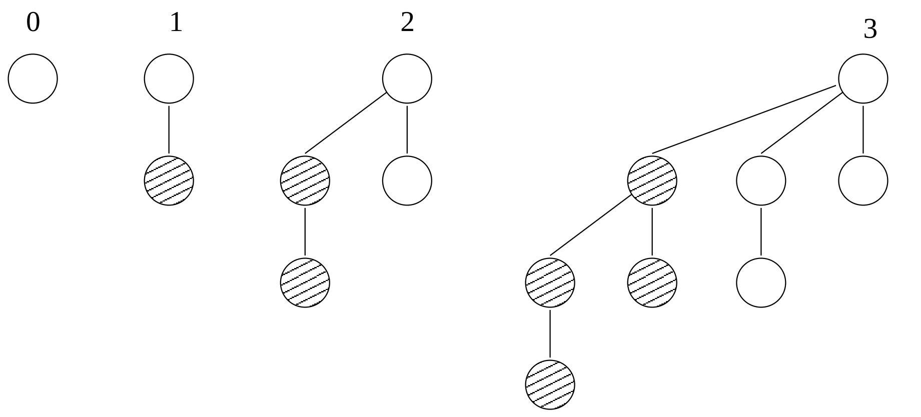

Figure 7: Binomial trees of degree 0, 1, 2, and 3. Each tree of degree k is composed of two trees of degree $k-1$, one shaded, one plain.

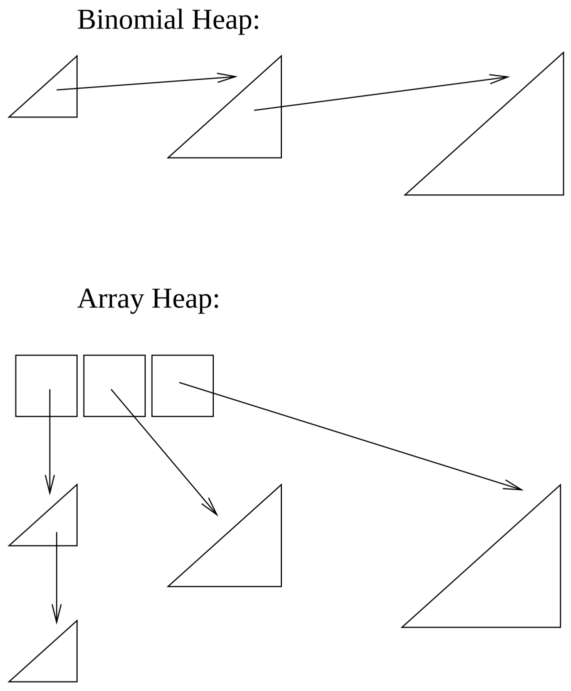

Figure 8: Differences in structure between ArrayHeap and BinomialHeap. Triangles are binomial trees; squares are array pointers.