Survey of Optimized Functional Data Structures

Mermer Dupree

*As computers become highly parallelized and memory becomes faster and cheaper, we must begin to consider new programming paradigms, like functional programming, to tackle the challenges that arise from these systems. Functional programming has for a long time been a programming paradigm of interest to computer scientists. It offers many benefits like safe multithreading, less error prone code and even the possibility of automatic parallelism. One of the challenges of functional programming is that it uses high amounts of memory. Using functional data structures that are optimized can reduce the memory usage significantly. In our research, we created four optimized functional data structures and compared their efficiency to similar non-optimized functional data structures and non-function* *data structures.[[1]](#footnote-1)*

**I. Introduction**

Moore's law states that chip performance doubles roughly every two years. While this has held true for the past 50 years, the semiconductor industry now believes this is coming to an end. In order to keep improving the performance of computers, we will have to instead increase the number of cores. Instead of relying on multiple layers of cache, computers could be optimized for highly parallelized computing. This, along with improvements in memory speed, changes the bottleneck of a program from the memory to the throughput. This puts the burden on programmers to create efficient parallel programs. As a result, code can become susceptible to bugs and race conditions.

Functional programming provides many benefits that allow programmers to tackle the complexity involved in parallel programming. Creating programs that are stateless results in code that is easier to maintain, easier to reason about and less bug prone, and eliminates most of the issues that result from multi-threaded code [1]. In fact, it is possible to automate the parallelism in a functional program. For example, Swift-T is a scripting language for high performance computing that can automatically distribute tasks across hundreds of thousands of nodes without any explicitly paralyzed code [2].

One of the challenges of functional programming is that is can use a great deal of memory. If you cannot change a data structure's state, programmers must make a copy of the data structure every time they wish to make a change. Functional programmers have created many optimized versions of functional data structures that significantly reduce the memory usage by sharing parts of a data structure that are unchanged. From the perspective of the client, it looks like they created a completely new instance of the data structure, but in reality the copy shares much of the same memory. Since the data structures are cannot be changed, we are guaranteed to avoid any side effects that would normally arise from the data structures sharing memory. We will look at four optimized functional data structures and compare them to their un-optimized versions as well as their non-functional counterparts. The four data structure are the linked list, the banker's queue, the red black tree and the bit vector trie.

**II. Linked List**

A functional linked list is one of the most basic optimized functional data structures. It is a LIFO container and has very memory efficient push and pop operations. Whenever an object is pushed, instead of returning a copy of the list with the object added, a new head node, containing the element and pointing to the original list, is returned. Likewise, when the top element is removed, the next node on the list is returned as if it was a completely new list without the top element. Notice that neither operation alters the state of the original data structure, yet the "new" list returned looks exactly as the client would expect. The resulting data structure is one that does not use any additional memory as compared to a non-functional stack, and maintains constant time push and pop. Alternatively, a non-optimized version would have to create a new copy of the list for both push and pop. This would result in not only a lot of memory usage, but also relatively slow push and pop.

**III. Banker's Queue**

The banker's queue is a FIFO container that is simple and efficient. It consists of two functional linked lists: an enqueue list and a dequeue list. The enqueue list is prepended to when an object in enqueued. The top of the dequeue list is popped off when an object is dequeued. When the enqueue list is larger than the dequeue list, the enqueue list is reversed and appended to the dequeue list. This operation is expensive, requiring us to copy the entire list, but since it happens exponentially infrequently, it has an amortized constant time [3]. This results in a data structure that has a performance that is very close to that of a non-functional queue. Figure 1 shows the memory used for inserts in a banker's queue versus that of the C++ standard library non-functional queue. The memory usage is nearly identical. Figure 2 shows the memory usage if we did not use any optimizations and instead copied the entire data structure for each insert. The improvement in memory performance is dramatic. Maintaining the banker's queue has a time cost though. In Figure 3, we compare the speed to the non-functional queue. While the banker's queue is slower, it is not dramatically different. Figure 4 shows the memory performance of deletes. The memory cost of deletes is relatively flat, suggesting that deletes have little impact on memory.

FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Used%20Inserting%20Objects%20onto%20a%20Queue.pdf FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Cost%20of%20Inserting%20Objects%20Into%20a%20Functional%20Queue.pdf

*Figure 1.*

*Figure 2.*

FunctionalDS/FunctionalDS/Results_graphs/time/inserts/Time%20Cost%20of%20Inserting%20Objects%20into%20a%20Queue.pdf FunctionalDS/FunctionalDS/Results_graphs/mem/deletes/Memory%20Used%20Deleting%20Objects%20From%20a%20Queue.pdf

*Figure 4.*

*Figure 3.*

**IV. Functional Red Black Tree**

The functional red black tree is a self-balancing binary search tree that can be used as an associative data structure. Just as in a non-functional red black tree, each node in the tree is either red or black. To maintain a balanced tree, there are two conditions that must be met when inserting and removing nodes. First, each node must have the same number of black nodes on its left side as on its right side. And no red node can have a red sibling (note that a terminating null node is considered to be black). When a new node is inserted it is initially red. If a violation occurs, there are procedures to rebalance the tree by swapping a few nodes or recoloring nodes. What makes this efficient as a functional data structure is that, for insert and deletes, most of the tree does not need to be copied. For inserts, only the nodes that are along the path from the root to the parent of the node being inserted must be copied. For deletes, only the nodes that are along the path from the root to the node being deleted must be copied. For times when the tree must be rebalanced, there is only slightly more overhead. In figure 5, we compare the memory performance of the functional red black tree to that of the C++ standard library non-functional map. Both scale linearly. Figure 6 shows the memory performance of a non-optimized red black tree which copies the tree for each insert. Clearly, the optimized version saves a lot of memory. In figure 7, we compare the speed of the functional and non-functional versions. The results show some overhead for maintaining the functional data structure, but the difference is not dramatic. Figure 8 show the memory performance of deletes as compared to a non-functional map. As shown, the increase in memory usage per delete is very gradual.

FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Used%20Inserting%20into%20an%20Associative%20Data%20Structure.pdf FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Cost%20of%20Inserting%20Objects%20Into%20a%20Fuctional%20Red%20Black%20Tree.pdf

*Figure 6.*

*Figure 5.*

FunctionalDS/FunctionalDS/Results_graphs/time/inserts/Time%20Cost%20of%20Inserting%20Objects%20into%20an%20Associative%20Data%20Structure.pdf FunctionalDS/FunctionalDS/Results_graphs/mem/deletes/Memory%20Used%20Deleting%20Objects%20from%20an%20Associative%20Data%20Structure.pdf

*Figure 7.*

*Figure 8.*

**V. Bit Vector Trie**

The bit vector trie is a functional data structure that is optimized for operations that you might perform on a vector like random access/update of elements and the append function. The bit vector trie starts off as an array of elements that is copied every time an element is inserted. This is repeated up until 32 elements. Upon inserting the 33rd element, an array of pointers is created and referred to as the root array. The first pointer points to the array of the first 32 elements that we inserted. The second second pointer points to a new array that contains only the 33rd element. With each additional insert the second array is copied as well as the root array to adjust the pointer. Once the second array is filled with 32 elements, a third array is created and pointed to by the third pointer in the root array. Once all 32 pointers of the root array are full, that is to say elements have been inserted, a new root array is formed. This time, it is an array of pointers to arrays of pointer. The result is a tree with a branching factor of 32 where the leaves are the inserted objects. This is shown in Figure 9. For every update, only the arrays that lay along the path from the root to the target index are copied. Therefore, the efficiency of the update is determined by the height of the tree. For arrays that will reasonably fit in memory, the height is less than or equal to six. Therefore, we can think of the memory usage as being bounded by a constant even though technically it is logarithmic [4]. Figure 10 shows comparable memory performance of the append operation between the bit vector trie and the C++ standard library non-functional vector. Figure 11 shows the performance of a non-optimized append. Like the other optimized data structures, the amount of memory saved is large. In figure 12, we see that there is a time cost for the functional data structure much like the other data structures we've looked at.

../../../Desktop/bitvectri.pdf FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Used%20Appending%20Objects%20onto%20a%20Vector.pdf

*Figure 9.*

*Figure 10.*

FunctionalDS/FunctionalDS/Results_graphs/mem/inserts/Memory%20Used%20Appending%20Objects%20onto%20a%20Functional%20Vector.pdf FunctionalDS/FunctionalDS/Results_graphs/time/inserts/Time%20Cost%20of%20Appending%20Objects%20onto%20a%20Vector%20copy.pdf

*Figure 11.*

*Figure 12.*

**VI. Conclusion**

Our results show a dramatic improvement in memory performance when these optimizations are applied. Optimizations like these are essential for functional programming to be efficient. Reducing the parts of a data structure that has to be copied improves the memory performance and the speed of the data structure operations. Our results also suggest that the data structure used is important in functional programing. Functional data structures are optimized for particular operations. This limits their flexibility, but dramatically improves performance. To study these issue further, future work in this area could include performing tests with caching disabled and performing tests in parallel.

**VII. References**

[1] Martin, Robert. “Functional Programming; What? Why? When?” NDC Oslo 2014, June 2014, Keynote Address.

[2] J. Wozniak, T. Armstrong, M. Wilde, D. Katz, E. Lusk, and I. Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, pages 95–102, May 2013.

[3] Okasaki, Chris. Purely functional data structures. Cambridge University Press, 1999.

[4] Spiewak, Daniel. "Extreme Cleverness: Functional Data Structures in Scala." Clojure/Conj 2011, 12 November 2011, Sheraton Raleigh Hotel, Raleigh, NC. Keynote Address.

1. The source code for this project can be found at www.github.com/mdupre2/Functional\_DSgithub.com/mdupre2/Functional\_DS [↑](#footnote-ref-1)