Baskets Queue c++ Implementation

Robert Bland
UCF Computer Science
Team #4

Jacob Crandall
UCF Computer Science
Team #4

Jacob Jiskoot
UCF Computer Science
Team #4

***Abstract*—FIFO Queues have been the subject of much research. They are useful for a number of reasons in concurrent settings. They can be used to buffer data or quickly create producer / consumer systems. Because of FIFO Queues wide usage in concurrent (and everyday) programming it is important that effort is made to make them both attainable and preformant. Today one of the most popular concurrent FIFO Queues is the Michael Scott (MS) queue [1]. It is even included in the Java standard library. The MS queue is fairly preformant and easy to implement. Its main draw is that the queue provides a lock free algorithm that supports concurrent enqueues and dequeues. The main issue is that while an enqueue and dequeue can occur conccurently multiple enqueues or multiple dequeues will fail in this case.**

The baskests queue [2] proposes an implementation that allows for multiple concurrent enqueues or dequeues. Due to its implementation it is also more portable. The preformance increase (along with other benefits provided by baskets queue has been proven. In this paper we will propose a c++ implementation of the baskets queue while discussing its shortcomings, benefits and preformance. This paper is meant mostly as discussion of the results and implementation of the baskets queue at a conceptual level. It is not meant to provide the implementation itself. Instead we leave it to the reader to use this as a supplementary tool along with our provided source code to understand the baskets queue at an implementational level.

I. Introduction

A. Michael-Scott Queue

The MS queue [1] provides a lockfree FIFO queue with concurrency at the head and tail. It provides a straightforward and easy to follow implementation that makes it an easy base for both every day use and for research so it makes sense try to improve on this algorithm. In the MS queue if two concurrent enqueues (or dequeues) occur then one will succeed or one will fail (or if there are three concurrent enques one will succeed and two will fail, etc.). The failed operation will back-off and try again later.

This gives the MS queue more difficulty in highly concurrent environments. The back-off helps this concurrency issue but it introduces new issues. The back-off scheme can be difficult to deal with for its own reasons as it requires tweaking that is specific to the system running the code. Even if you find some library implementation if you want optimal preformance you will have to take time to tweak and test parameters.

B. Baskets Queue

The baskets queue [2] is based off of the MS queue but proposes changes to deal with both its concurrency issues and the back-off scheme problem. The baskets queue can be extended to other queues as well, such as the optimistic FIFO queue [3]. However, in this implementation we will be looking at an implementation that is based off of the MS queue in a linked list form.

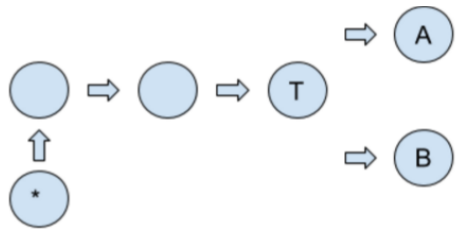

Fig. 1. Example of concurrent enqueue to an MS queue in a linked list implementation. Assume that A succeeds then B must re-traverse the list and back off before trying to insert again. In this case it must re-traverse the list as well, further hurting preformance.

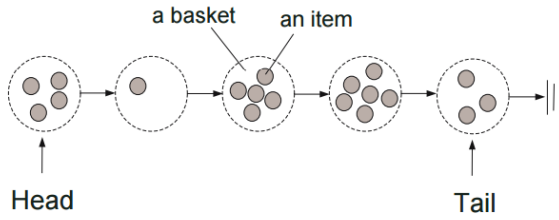

Fig. 2. Image taken from [2]. This is the abstract baskets queue based on a linked list implementaiton. Nodes are individual baskets whereas the baskets are just conceptual, not real structures.

The main principal behind the baskets queue presented in [2] is that concurrent operations can be re-ordered in any order and still be linearizable. So for example if we preform two concurent operations $enqueue(A)$ and $enqueue(B)$ then the output when dequeuing could be $A \rightarrow B$ or $B \rightarrow A$ and still be considered correct. We may retroactively set the linearization points and thus have a correct implementation regardless of actual outcome of the dequeues. In the case of no concurrent enqueues or dequeues the baskets queue based on the MS queue behaves exactly like the MS queue. It is when conflicts occur that an item is inserted into a basket and the baskets are considered a kind of ”back-off” scheme. It is in these concurrent situations that the baskets come into play. Each item that is being inserted into the basket will have happened concurrently, meaning that it failed on the

compare and swap operation that tried to move the either the head or tail of the queue.

In this case (like the MS queue) one item will succeed but the rest that failed will insert into the queue in any order. The major benefit is that the first successful enqueue becomes the new queue tail while the other inserts get linked "before" this new tail. Because of this enqueues can continue to happen past that new tail making it so that **concurrent operations can happen between different baskets**. However, concurrent operations still may not occur per basket. In [2] they show that in many cases there will only be around three items per basket (even with high concurrency) which makes it so that a back-off scheme is not necessary. If there is a highly highly concurrent environment though it may be of interest to implement a back-off scheme within the basket insert but otherwise the problem is minimized and less of concern.

The dequeue of the baskets queue is more or less the same as that in the MS queue proposed in [1]. That is when a node to remove is found then the queue will preform a compare and swap on the head, if there are issues finding some non-deleted node between head and tail then some action will be taken to correct it (often times just updating head values and trying again) or the queue may have been empty, in which case we simply return NULL. It is important to note that while enqueues will always succeed (sit in the while loop until they enqueue) that a dequeue will immediately return NULL if the queue is found to be empty, it will not spin and wait for a value to be inserted.

The baskets queue in [2] adds some auxilary features to the original MS queue presented in [1]. One of these is the ability to logically delete nodes. This is added by editing the pointer itself to store extra information. Furthermore along with the "deleted" bit a counter is also implemented to avoid the ABA problem. The use of logically deleted nodes is a popular method that allows us to essentially save hard work for later and do it all at once. That is exactly what is done in this case. When a chain of length X or more is found while removing nodes (a thread counts while traversing) then a chain will be physically deleted. This is done by resetting the tail (in an "if compare and swap" so that only one function will enter the free chain method) which cuts off all public access to this chain. The thread inside this free chain method is then free to reclaim nodes as it can.

II. Implementation

Our implementation is based heavily on the pseudo-code provided in [2]. For the most part this pseudo-code, our code and the concepts presented may be studied to gather a very detailed understanding of how the baskets queue operates. If directly running the provided code the program takes the following arguments:

- Arg1 : Number of threads : How many threads will be preforming tasks
- Arg2 : Number of jobs (Enqueues & Dequeues)
- Arg3 : Percentage of enqueues (the rest will be dequeues)
- Arg4 : Number of pre-populated nodes : 0 - Inf

After compiling with the provided make file the program may be run as an example

```
./main 4 500 .6 0
```

This will create 4 threads that together will execute 500 enqueue or dequeues, 60% of which will be enqueues and 40% of will be dequeues. The "number of pre-populated nodes" argument will be described below.

A. Supporting Implementation

We define a sort of "harness" that we use to give an example implementation of the baskets queue. There are two main parts of this. First is the sections that support simply running the baskets queue, that is a system for actually preforming enqueues and dequeues. This section may be skipped if the reader wishes as it is mostly about how we tested and utilized the queue itself. There are some concepts that apply towards concurrent programming here though that may still be of interest. Secondly we have sections of code that support the baskets queue itself. These are sections that are abstracted away by the pseudo-code in [2]. These include things like pointer manipulation for tags and counting.

1) Preforming Enqueues and Dequeues: The objects we are inserting are based around a specialized node class. If looking into using the provided code in a similar implementation it would be simple to retain all specialized node related work and simply change the payload of the nodes. For our case we are utilizing ints, this was an arbitrary decision and any sort of payload could be used here. We first take some pre-cautions to prepare our threads for a multi-threaded environment. This means that we want to seed our random jobs (percentage of enqueues / dequeues) before spinning up threads. This is because the provided c++ random implementation is not gauranteed to be thread safe. Each thread will then continually take jobs from this seeded jobs array until it is empty. Similarly to how we seed randoms we also seed nodes. That is, each thread is provided a local list of nodes. When it enqueues it pulls a new node address from this list and inserts it into the queue. When dequeing we return a pointer to such a node, the thread will reattach this pointer to its local node list. In this way we avoid ever have to allocate new memory in a concurrent environment. This is what the previous "number of pre-populated nodes" argument was for. We left it up to the user to determine a safe number of nodes to pre-allocate per thread. For example if preforming 99% dequeues then each thread may only need a small number of pre-allocated nodes but technically still with 99% dequeues we could have all enqueues. (We do not gaurantee a certain number of enqueue / dequeues as jobs are seeded randomly with bias). So the user may want to be safe and give each thread the same number of nodes as number of jobs. As a side note we did find some unreliable information that

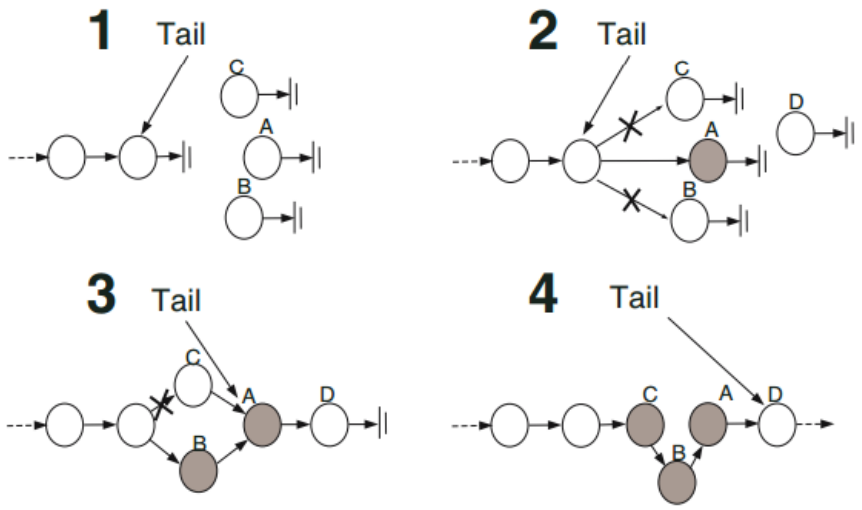

Fig. 3. Image taken from [2]. This is the abstract baskets queue based on a linked list implementaiton. Nodes are individual baskets whereas the baskets are just conceptual, not real structures.

suggests that this may not be necessary. It was suggested that if the pthreads library is linked (which it is, via our make file) then instead of the standard malloc / new memory allocators a thread safe malloc is loaded. It is worth noting however that this thread safe malloc could still have locks, hurting the runtime preformance. For those multiple reasons we leave it up to the user to determine how they would like to handle node pre-allocation. In our implementation we do not make an effort to clean the nodes either, this could be simply done once work is determined to be done however. We do make one intrusive change to the basket queue in order to support this pre-allocated node behavior. In the original baskets queue dequeue method pseudo-code provided in [2] they provide a generalized ”recycle node” function. This is meant to do exactly as we describe above. That is it makes some node pointer available to be re-used. The implementation will still work if this is re-worked to be a more widely available methodology.

2) Baskets Specific Auxilary Work: There is work outside of what is provided within the contents of the paper that this implementation is based off of [2]. Namely, the baskets paper [2] does not provides simply pseudo-code and assumes available implementations for certain tasks. The largest of these tasks is the creation data structures that can be atomically swapped that include pointers, integers and a boolean. The baskets queue relies heavily on the ability to preform atomic compare and swap operations on multiple pieces of data. It is required to be able to atomically swap a pointer, a reference counter (how many times the node has been touched) and the logical deletion flag that was mentioned earlier. This is not a behavior that is native to c++ so we must implement it manually. From our classes we learned that not all bits of a pointer are utilized. So we may utilize those ending bits without changing the valid values of the pointer. We do this by preforming bitwise operations to save values as needed.

B. Baskets Implementation

TODO: Talk about how we dont implement backoffs and other specifics to our implementation, could just walk through what our baskets.cpp does here mostly

C. Personal Experiences

One of the benefits of the baskets queue is that it is more portable than some other queues, especially those that heavily rely on back-off schemes. For that reason we were reluctant to implement the bit stealing operations as they are not portable. This is because the number of bits which must be stolen is different per system. To satisfy this portability requirement we tried to build in constants for any bit stealing operations we utilize. In the end we believe this implementation is still more portable than altering back-off scheme parameters which may require further testing; although we would like this implementation to be more portable if possible. For our specific machine implementation (Ubuntu 16.04 x86_64 architecture) the number of valid bits for stealing was found to be four. Unfortunately this means that after storing the logical deletion bit we only have three bits to store a reference counter. This introduces issues if we need four bits to store how many times a node has been referenced (which probably will happen often in any meaningful implementation). This is the biggest issue with our implementation, it does not completely eliminate the ABA problem but simply avoids it. The tags will often times not line up with references if an ABA problem arises but it is still possible. This could be fixed if we utilized a Compare and Swap two as our professor suggested in conversation. This would allow us to compare and swap with our pointer (which could still use the stolen logical deletion bit) and some

unsigned int in one atomic step. However, we were unable to find any such working implementation for c++. **TODO continue working on difficulties, etc.**

III. Benchmarks & Experimentation

A. The Experiments

Talk about machine we ran the tests on and what we benchmarked against

B. Results

Acknowledgment

We would like to thank professor Damian Dechev for teaching and some consultation concerning this project.

IV. Conclusion

The conclusion goes here.

References

[1] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, p. 126, 1998.

[2] M. Hoffman, O. Shalev, and N. Shavit, "The baskets queue," *Lecture Notes in Computer Science Principles of Distributed Systems*, p. 401414.

[3] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues," *Distributed Computing*, vol. 20, no. 5, p. 323341, 2007.