Optimizing Performance of Persistent and Concurrent Data Structures Implementation

by Ariel Kazula, Saar Bar-Oz and Menucha Benisti
with guidance of prof. Hagit Atia

The primary objective of this project was to investigate the performance of two distinct algorithms for a persistent and concurrent stack, namely DFC stack[1] and PBcomb stack[2], with the aim of identifying factors that can be leveraged to improve the DFC stack efficiency. To achieve this, we analyzed the implementation details of both algorithms.

Algorithms:

First we will present a brief explanation on the two algorithms, their implementations and discuss the differences between them.

The DFC Algorithm:

The DFC algorithm involves each process announcing its operation by adding it to an announcement array. Each process attempts to obtain a global lock and become a combiner. If a process is unable to acquire the lock, it waits for the combiner to apply its operation. The combiner collects all the announced operations and applies them to the linked list representation of the stack, subsequently writing the response to the announcement array. Additionally, the combiner utilizes an elimination technique to pair concurrent Push and Pop operations.

The Operation Request:

During each operation, a thread communicates its intention to either push or pop an item onto or from the stack by writing the operation type and argument into a shared announcement array. The thread then sets a flag indicating that its announcement is ready to be collected by a combiner thread. Following the announcement, the thread aims to become the combiner by acquiring a shared lock. Only one thread can execute the combiner code at a given time.

The Combiner:

Initially, the combiner code calls the Reduce procedure. The Reduce procedure is responsible for combining all announced operations and pairing up Push and Pop operations in a thread-safe and fault-tolerant manner. It collects all announcements that are ready, updates the epoch in the announcement structure to be the current global epoch number, and inserts operations into pop and push operation lists. Then, it performs the actual reduction/elimination by eliminating couples of Push and Pop operations without accessing the linked list stack representation, reducing the number of PWB instructions. Finally, it returns the surplus push or pop operations and applies them to the stack. The combiner persists all modified variables, including the combined announcement structures and the updated top entry, using PWB instructions, followed by a single Pfence instruction.

Then, if there are surplus push operations, the combiner allocates new nodes for each push operation. If there are surplus pop operations, the combiner removes nodes from the stack, uses their key as the response value, and deallocates them. After applying all operations, the combiner updates the top entry and persists all modified variables using PWB instructions. Finally, the combiner releases the combiner lock, and returns its response.

The PBComb algorithm:

The PBComb algorithm operates by maintaining two copies of the object's state in an array called MemState. Each record in MemState consists of three fields: st (the current state of the object), ReturnVal (an array that stores the response value for each thread), and Deactivate (a bit vector that indicates whether a thread's request has been served or not).

To reduce synchronization overhead, the algorithm uses an integer shared variable called Lock to implement the lock. An odd value of Lock indicates that the lock is taken, while an even value indicates that the lock is free. When a thread wants to acquire the lock, it executes a CAS operation on Lock. If the CAS operation succeeds, the thread becomes the combiner and executes the combiner code.

The combiner chooses one of the two StateRec records in MemState that is not currently indexed by MIndex to serve requests. It then copies the current state of the object into the chosen record and executes a simulation phase, where it processes each active request by applying the request using the chosen record, recording the response value in ReturnVal, and setting the corresponding Deactivate bit to indicate that the request has been served. Once the simulation phase is complete, the combiner updates MIndex to index the chosen record and releases the lock, giving up its role as combiner.

When a non-combiner thread wants to execute an operation, it first records its request and its *activate* bit in Request. It then checks whether the lock is free. If it is not, the thread busy-waits until the current combiner releases the lock. Once the lock is free, the thread checks whether its request has been served by checking whether its activate and deactivate bits match. If they do, the thread returns the response value stored in ReturnVal. Otherwise, the thread contends again for the lock.

Theoretical Differences between the algorithms:

There are several theoretical differences between the DFC and PBComb algorithms:

1. The DFC algorithm uses the Reduce procedure in order to eliminate concurrent push and pop operations, in order to avoid accessing the stack itself. The PBComb algorithm uses no such technique and performs every thread request on the stack.

2. Due to the addition of the Reduce procedure in the DFC algorithm, there is a need for an extra array of size N called *vcoll* to be constantly updated, in order to indicate which requests have been collected for the procedure. The PBComb algorithm obviously requires no such array.

3. The DFC algorithm stores in the nvram a variable called *cepoch*, whose main purpose is to track which request has been served. The PBComb algorithm uses an *activate* bit for each request to indicate if it has been served.

4. The PBComb algorithm maintains two copies of the global memory state and swaps between them after persisting the current copy. Conversely, the DFC algorithm stores copies in lower resolution, including two copies of the stack and two copies per announcement request. This enables better granularity for recovery and avoids the need to copy data for unhandled requests. However, it can negatively impact performance due to the need for additional write-back instructions to persist these copies.

5. The DFC algorithm persists the object that stores the threads' requests to the non-volatile random-access memory (NVRAM), while the PBComb algorithm only persists the results of the operations. This distinction reflects a tradeoff between persisting the requests themselves to keep them available after a crash, and not persisting them, which results in better performance but a lower level of crash recovery.

Practical Differences between the algorithms:

1. The DFC algorithm uses the common Posix functions in order to handle work with threads, while the PBComb algorithm uses its own mechanism to handle threads and their scheduling.

2. PBComb algorithm takes into account NUMAs - groups of cores inside the processor that have shared caches. The algorithm tries to split threads to the same NUMAs, so operations they perform will be faster thanks to cache considerations. The DFC algorithm does not consider NUMAs at all.

3. The DFC algorithm uses a pool of pre-allocated nodes for the stack, in order to avoid memory allocation for every push. The PBComb algorithm doesn’t use such a pool.

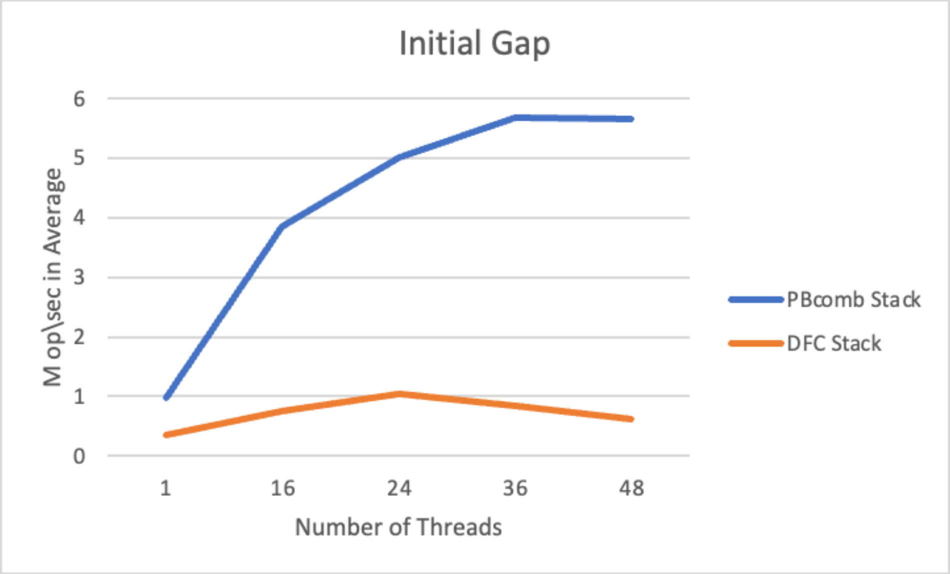
4. The DFC algorithm uses transactional persistent memory allocation, while the PBComb algorithm uses direct persistent memory allocation.

The initial gap in performance between the implementations of the 2 algorithms:

To evaluate the performance of the DFC and PBComb algorithms, we used a push-pop benchmark, in which we executed 1M couples of push and pop operations that were distributed equally between the threads. We ran the benchmarks using a varying number of threads in order to test algorithms' throughput and scalability. Each experiment was repeated 10 times, we calculated the median results and obtained the following results:

No. threads	**1**	**16**	**24**	**36**	**48**
DFC Stack	0.34	0.76	1.04	0.83	0.612
PBcomb Stack	0.97	3.86	5.01	5.68	5.65

Measured by Average throughput (million ops/sec)



As we can see, the PBcomb stack outperformed the DFC stack by a large margin.

Optimizations experiments

In our project, we tried to understand and identify the reasons why the PBcomb stack algorithm implementation had significantly better performance than the DFC stack algorithm implementation. Our goal was to understand what attributes had more effect than others on the performance of the algorithms. Through our research, we explored various ideas and hypotheses regarding the strengths and weaknesses of each algorithm and devised a comprehensive testing plan to validate these assumptions. In this report, we will present our findings and discuss the attributes that contributed the most to the superior performance of the PBcomb stack algorithm, and present guidelines for writing and designing efficient persistent memory programs.

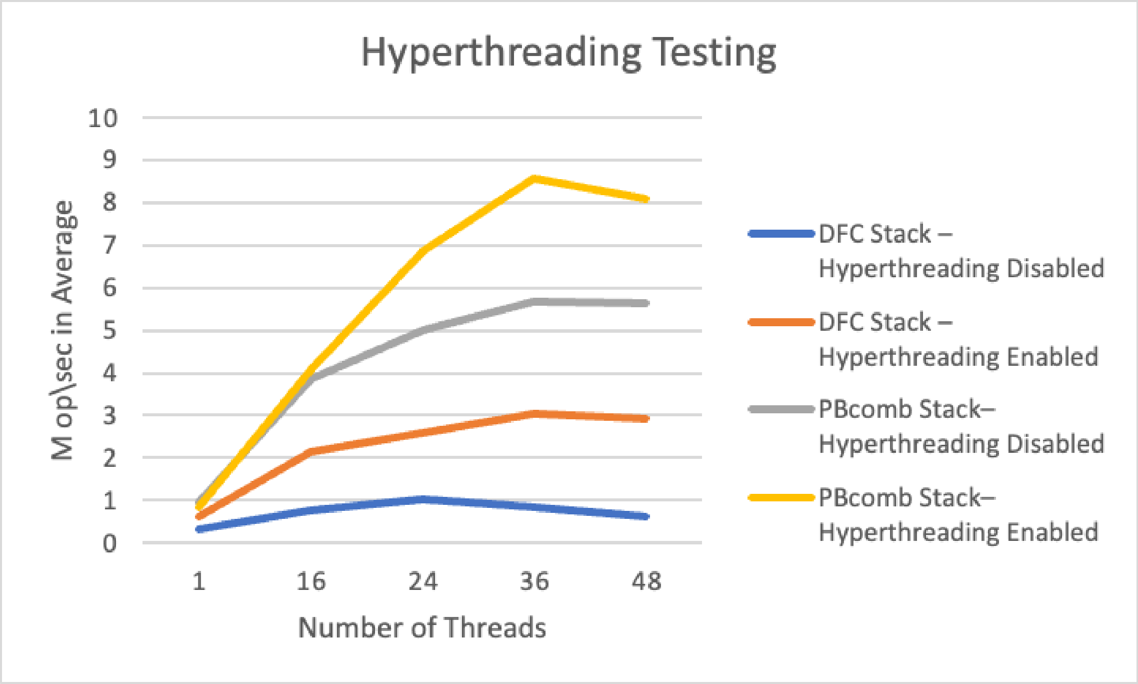
Testing the effect of hyperthreading mode on the algorithms:

Hyperthreading, also known as simultaneous multithreading , is a technology that allows a single physical CPU core to execute multiple threads simultaneously. This means that a processor with hyperthreading enabled appears as two or more logical processors to the operating system, allowing multiple threads to be executed concurrently. In a concurrent program, hyperthreading can improve performance by allowing multiple threads to be scheduled and executed more efficiently, especially when one thread is waiting for another to complete a task. With hyperthreading, a CPU core can switch between executing different threads more quickly, reducing the amount of idle time and improving overall throughput.

With that being said, we had a good reason to believe that both the algorithms, which are of course multi-threaded, would achieve better results when running on a machine with the hyperthreading technology enabled. To test this we run the benchmark test with hyper threading both enabled and disabled.

The results obtained are presented in table below:

No. threads	**1**	**16**	**24**	**36**	**48**
DFC Stack – Hyperthreading Disabled	0.34	0.76	1.04	0.83	0.61
DFC Stack – Hyperthreading Enabled	0.62	2.16	2.58	3.03	2.92
PBcomb Stack– Hyperthreading Disabled	0.97	3.86	5.01	5.68	5.65
PBcomb Stack– Hyperthreading Enabled	0.86	4.08	6.88	8.58	8.1



From the test results above it is clear to see that hyperthreading indeed increases performance,for both the algorithms as assumed, additionally we can see that this change did not affect the gap between the algorithms.

All the test results that appear in this report from this point forward are when running on a machine where hyperthreading is enabled.

After the theories we raised from looking at the algorithms presented in the articles in question, we turned to the code and tried to find differences in the implementations that might have an impact on the performance of the algorithms.

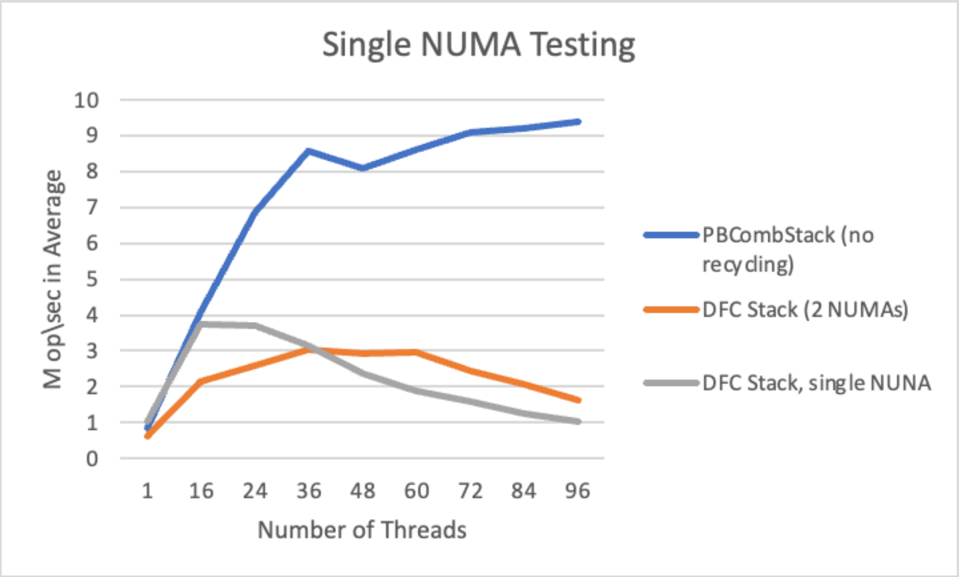
NUMA Nodes Affect On Performance:

We noticed that in the PBcomb Stack implementation there are some arguments controlling the system configuration for each benchmark run. One of these arguments was NUMA_NODE_SUPPORT, which when enabled, allows the user to control the number of Numa nodes used by the running process.

The use of multiple Non-Uniform Memory Access (NUMA) nodes can have a significant impact on the performance of synchronizing operations. When threads running on different processors access memory in different NUMA nodes, there can be a significant delay in accessing that memory, which can result in poor synchronization performance. In order to minimize the impact of NUMA on synchronizing operations, it is important to ensure that threads accessing shared data are pinned to the same NUMA node. This can be achieved by using appropriate scheduling techniques that take into account the NUMA topology of the system.

The server we were working on has 2 NUMA nodes, so we wanted to test the performance of our code on a single NUMA and 2 NUMA nodes, which is the default configuration. The results can be seen below:

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack Single NUMA	1.02	3.72	3.70	3.15	2.38	1.88	1.59	1.25	1.01
DFC Stack 2 NUMAs	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64



Our experiments revealed that running the program on a single NUMA node resulted in better performance when the number of threads was smaller than the number of CPU cores in the system. Specifically, on our test machine with 40 cores (20 base cores that act as 40 due to hyperthreading), running the program on a single NUMA node was optimal when the number of threads was less than 40. However, when the number of threads exceeded the number of system cores, using multiple NUMA nodes resulted in improved performance.

The reason for this behavior can be explained by the fact that when there are more threads than cores that never yield, the processor needs to constantly switch between threads to allow them to make progress, and this can result in cache thrashing on the Numa level. Using multiple NUMAs can help alleviate this issue by providing additional memory banks that can be accessed by certain cores without interfering with others.

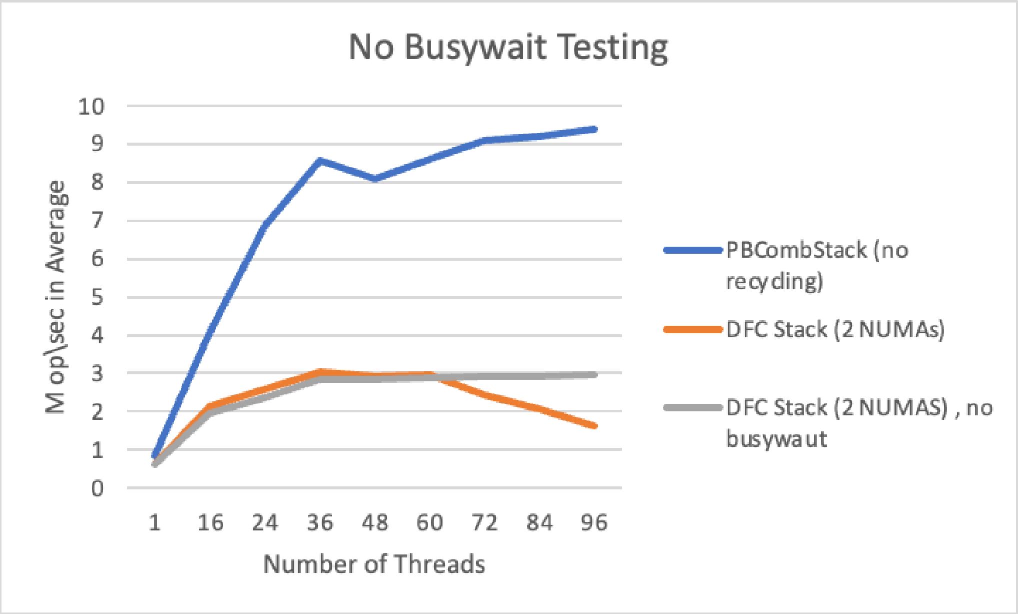
However, when the number of threads is lower than the number of available cores, using multiple NUMAs can result in increased overhead in coordinating between the different memory banks, which can lead to reduced performance.

Avoiding Busy Waiting:

Following the previous results we conducted a test by adding a yield command when waiting for the combiner instead of busywaiting.

Multiple Numas

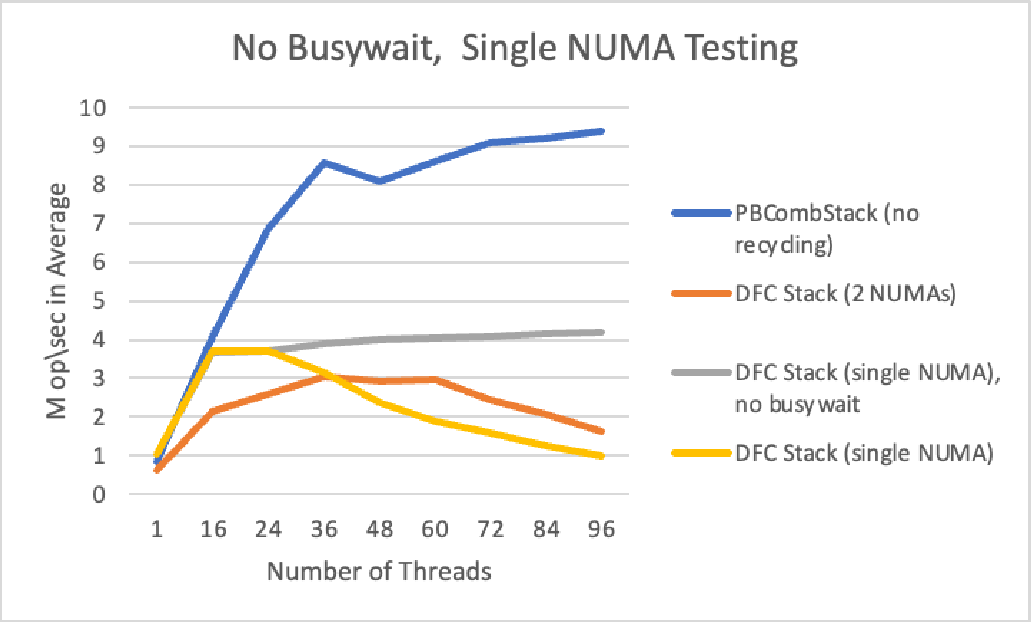
No. threads	1	16	24	36	48	60	72	84	96
DFC Stack Using 2 Numas	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
DFC Stack without Busywaiting Using 2 Numas	0.61	1.95	2.36	2.84	2.86	2.90	2.93	2.94	2.98



As expected, when the number of threads is big enough, yielding provides better performance rather than busywaiting, as there are no unnecessary context switches

Single Numa:

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
DFC Stack Single NUMA	1.02	3.72	3.70	3.15	2.38	1.88	1.59	1.25	1.01
DFC Stack, without Busywaiting single NUMA	1.02	3.67	3.70	3.91	4.01	4.05	4.09	4.15	4.18



Notably, our experiments showed that avoiding busy waiting when the number of threads exceeded the number of system cores removed the theorized problem of cache thrashing while maintaining the reduced overhead of synchronization between different NUMA cores. This allowed the increase in performance to continue even after passing the number of system cores.

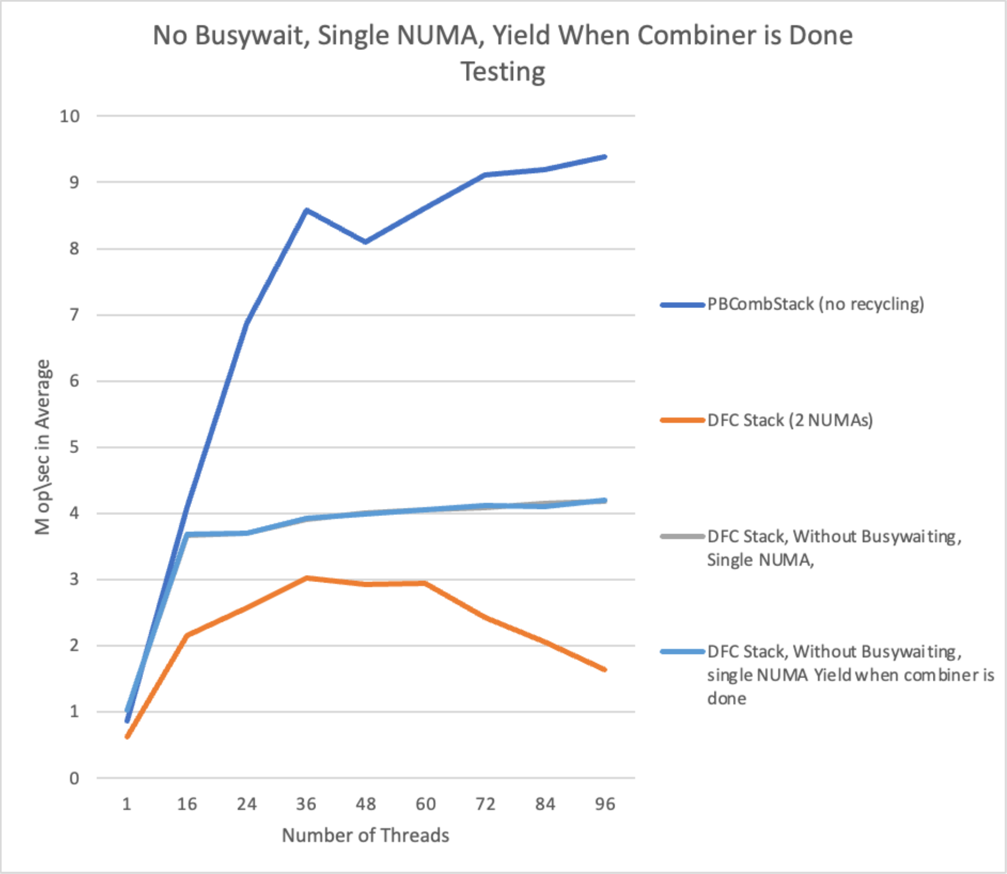
Yielding at the end of combiner phase:

During our testing, we observed that the combiner tended to execute a relatively large number of iterations using the same thread when utilizing yielding.

We hypothesized that this phenomenon could be caused by the waiting threads not awakening before the combiner enters a new epoch. This can cause the program to underutilize the epoch and handle only a small subset of the threads' requests each epoch, as some threads may not have managed to process their results and announce new requests before the epoch starts. This can lead to inefficient use of system resources and lower overall program performance.

To address this issue, we experimented with adding a yield in the end of the combiner function to allow other threads to process their results and announce new requests before the start of the next epoch.

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
DFC Stack, Without Busywaiting Single NUMA	1.02	3.67	3.70	3.91	4.01	4.05	4.09	4.15	4.18
DFC Stack, Without Busywaiting Single NUMA Yield when combiner is done	1.02	3.69	3.70	3.93	3.99	4.06	4.12	4.10	4.20



Our experiments showed that this approach only had a minor impact on the overall performance of the algorithms in most cases.

Afterwards, we turned to another factor we wanted to test:

Pinning Threads:

The process of pinning threads to specific CPU cores can have a significant impact on the performance of synchronizing operations in parallel computing. When threads are not pinned to specific CPU cores, the operating system can move them between different cores in order to balance the workload. This can result in unpredictable delays and contention for shared resources, which can negatively impact synchronization performance.

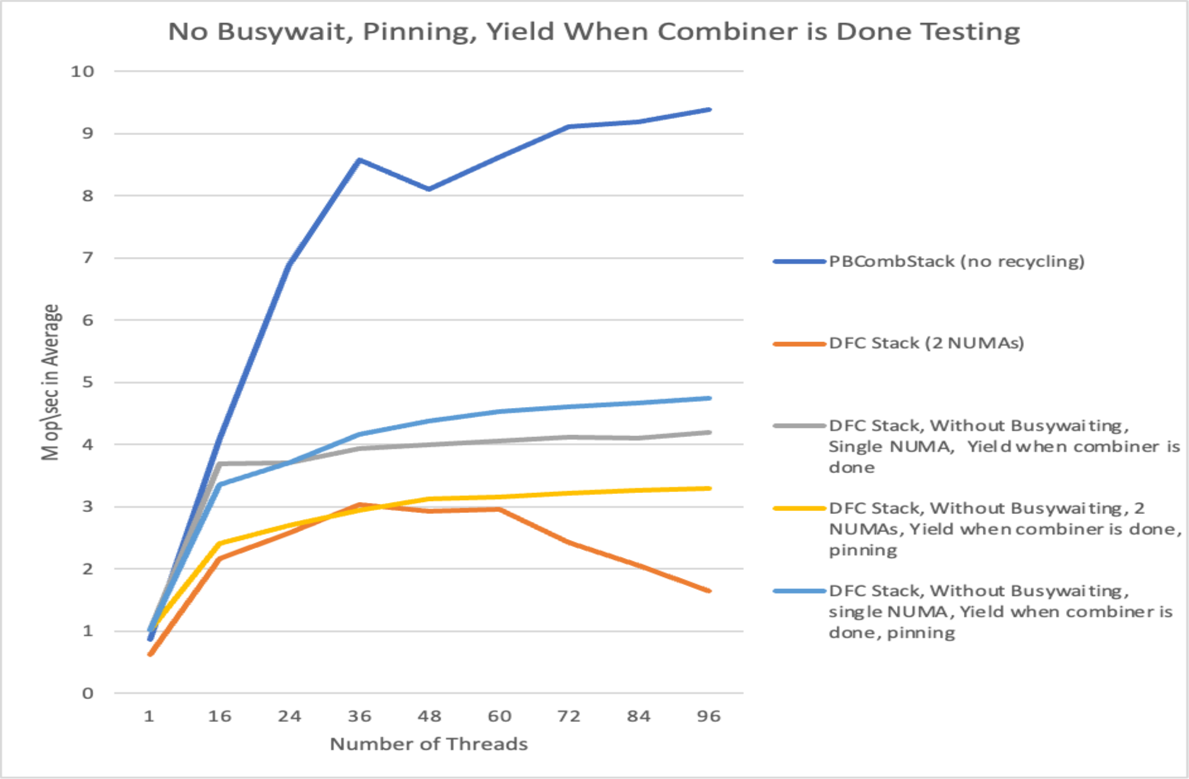
By pinning threads to specific CPU cores, we can ensure that they remain on the same core throughout their execution. This can help to minimize the overhead associated with thread migration and contention for shared resources, which can in turn improve synchronization performance.

However, it is important to note that pinning threads to specific CPU cores can also have negative effects on performance. This change can cause cores to be underutilized in case of imbalance of the workload, additionally if we pin too many threads to a single CPU core, we may introduce contention for shared resources on that core, which can result in poor performance.

Overall, the process of pinning threads to specific CPU cores can have a significant impact on synchronization performance in parallel computing. By carefully considering the number of threads that are pinned to each CPU core and taking into account the NUMA topology of the system, we figured we might further improve the program's performance.

We pinned only the process to a single NUMA using round-robin and received the following results:

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
DFC Stack, Without Busywaiting Single NUMA Yield when combiner is done	1.02	3.69	3.70	3.93	3.99	4.06	4.12	4.10	4.20
DFC Stack, Without Busywaiting Single NUMA,Yield when combiner is done, Pinning	1.02	3.36	3.7	4.16	4.38	4.53	4.61	4.67	4.75
DFC Stack, Without Busywaiting 2 NUMAs,Yield when combiner is done, Pinning	1.02	2.40	2.69	2.94	3.12	3.15	3.22	3.27	3.30



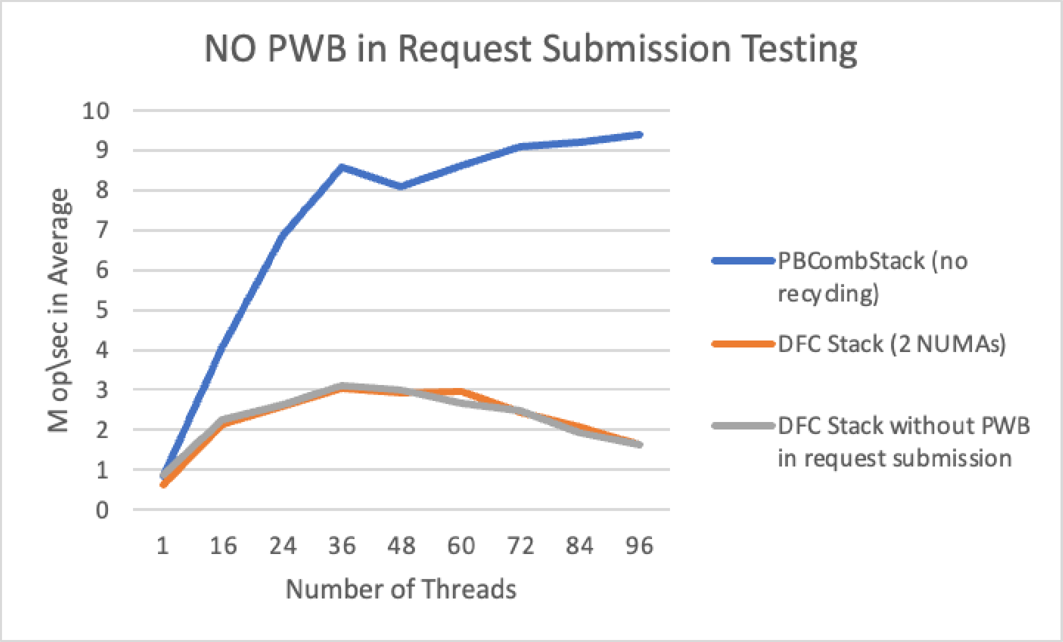
Removing the explicit persisting operation (PWB) of the request data:

Next, we suspected that a significant factor contributing to the differences in results might be the difference in the request submission process between the DFC and PBComb stacks. In the DFC algorithm, the *Op* function is responsible for submitting a new request and involves two write operations to NVM, followed by PWB and PFENCE commands. The first write operation updates the *announce* structure with the new request, while the second updates the *valid* field in the *transactional_announce* structure, allowing the combiner to access the new request.

In contrast, the PBComb stack only executes one fence command during the request submission phase. This call to *syncFullFence()* is not explicitly specified in the paper describing the algorithm. We questioned whether backing up each thread's request in NVM was necessary since in case of a system crash, each thread could restore its state and resend any uncompleted requests. We discovered that the DFC algorithm was inefficient in comparison to the PBComb stack in this regard.

To investigate this further, we attempted to remove the necessary PWB commands under the assumption that they were the most expensive call. We ran the test and compared the results to the previous code in the same configuration. The results are presented below:

No. threads	**1**	**16**	**24**	**36**	**48**	**60**	**72**	**84**	**96**
DFC Stack	0.62	2.16	2.58	3.0	2.92	2.95	2.43	2.05	1.64
DFC Stack without PWB in request submission	0.87	2.27	2.64	3.12	2.99	2.65	2.49	1.91	1.61



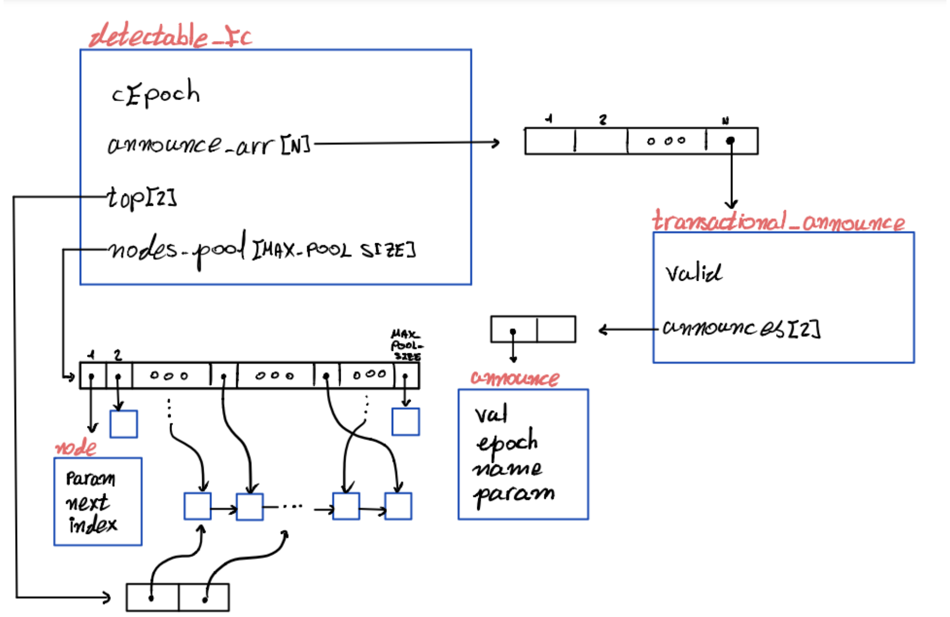
Based on the above results, we can conclude that the significant decrease in the total number of PWB operations did not have a significant impact on performance. This was a surprising finding as we initially believed that PWB operations were very expensive in terms of performance. We gained a better understanding of the reasons behind this change later on, as detailed in the PMEM Research section.

Changing the DFC Data Structures:

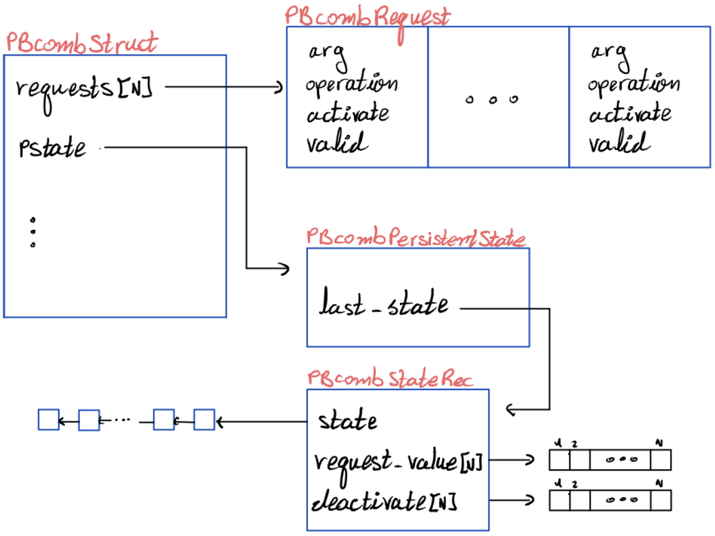
We delved into the differences between the data structures used by the DFC and PBcomb , and analyzed them.

For convenience, we bring here schemes of both data structures:

DFC Stack Data Structures:



PBcomb Stack Data Structures:



We focused on the following differences that we suspected that had the most impact.

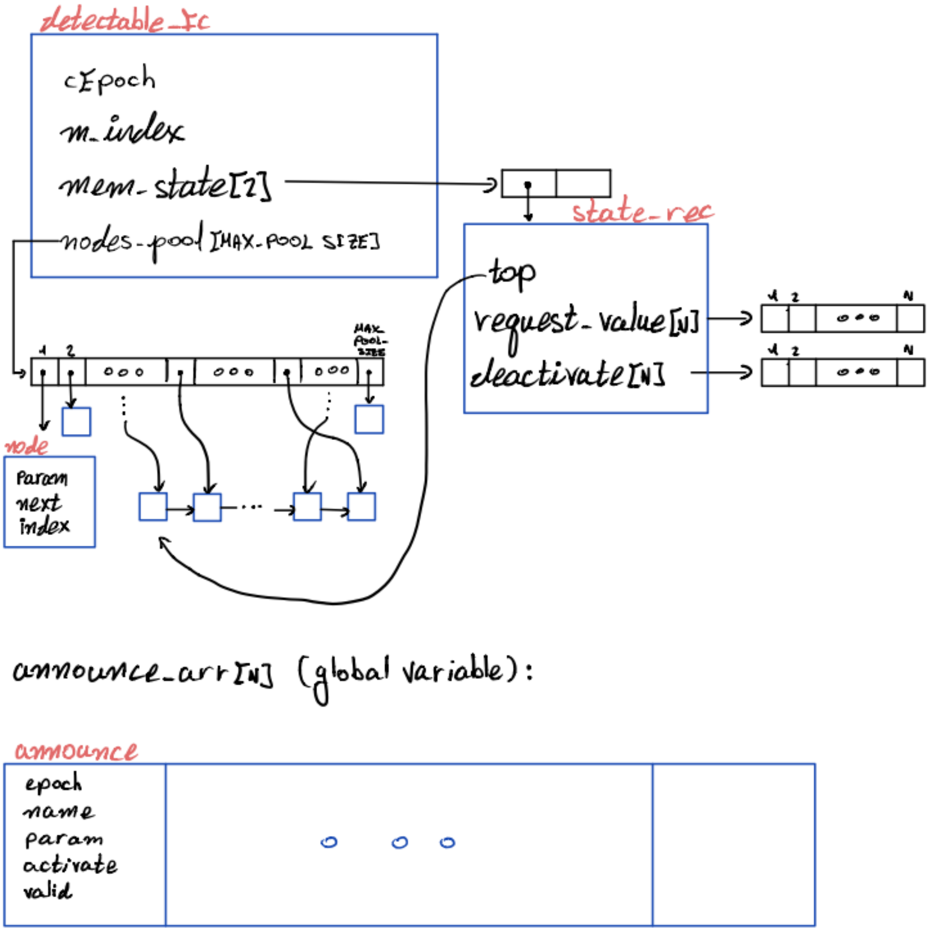
1. In the DFC stack, the structure *announce* contains both the request parameters: *name, param*, and the result parameter: *val* all of which are persistent. When on the other hand, in the BPcomb stack, the results of the request written by the combiner are all in the structure *BPcombStateRec* which is a persistent structure (as it must be), and the request data is organized in a separate data structure called *PBcombRequestRec* which is volatile. Which means that the PBcomb stack persists less data.

2. In the DFC stack there is an excessive use of pointers, where in the BPcomb stack there are somewhat less pointers.

   Using pointers here is wasteful for 2 reasons

   1. Accessing a field using a pointer can require two I/O operations, one to access the pointer and another to access the actual data. This can result in additional overhead and reduced performance.
   2. The use of pointers requires more persistent data, as it is necessary to persist both the pointer and the data it points to. This can result in higher storage requirements and slower write time.

We have made changes to the data structures of the DFC stack in an effort to mitigate the problems. The changes we have made are outlined in the following scheme:



We separated the structure of the requests to which all the threads write, and the combiner reads from, from the structure containing the results of the requests to which only the combiner writes, and the rest of the threads read from.

We moved the requests array from permanent memory to volatile memory in order to avoid unnecessary IO calls. By making this change, we were able to arrange the results in contiguous memory, which should have helped with IO efficiency when persisting the data in bulk.

Finally we run the benchmark on the modified DFC stack code. The following table shows the results achieved by this test:

No. threads	**1**	**16**	**24**	**36**	**48**	**60**	**72**
DFC Stack	0.62	2.16	2.58	3.0	2.92	2.95	2.43
Modified DFC Stack	0.45	1.53	1.8	1.99	2.05	2.08	2.11

For technical reasons this test was run on basic configurations.

As can be seen, in contrary to our expectations the results we got for the modified DFC stack were actually worse than the base DFC stack performance.

This degradation could maybe be explained by these two characteristics of the modified DFC stack:

1. In the beginning of every combiner phase, the entire *state_rec* structure is being copied, that is so in case of system crash in the middle of the combiner phase the old copy of *state_rec* will contain a valid state that can be recovered from the persistent memory. This copy may be costly in terms of performance compared to the base DFC stack. That is since before only the *announce* structure of the thread collected was copied, and now we copy 2 arrays of length N.
2. The library we were working with utilizes pointers to store objects, which meant we couldn't remove their usage.
3. At the time, we still believed that the PWB (persistent write buffer) operations were the most costly calls and didn't change the actual read/write to the objects .

So, we decided to take a step back and do some more research.

PMEM Research:

After conducting some research and further consideration, we realized that memory devices use internal logic to maximize performance, which means that memory can be persisted without explicitly performing PWB operations. In such cases, the PWB operations themselves do not have any effect, which could explain our findings.

Furthermore, our readings have highlighted that every access to the non-volatile memory, including normal read/write calls, can incur significant costs and negatively impact program performance. This is due to the device's input/output (IO) calls for most of the access to the data, which can result in substantial delays. As a result, it is critical to minimize the number of IO calls by efficiently organizing and accessing data in memory.

In light of the performance costs associated with accessing persistent data structures, it is critical to minimize the total number of read and write operations to these structures, rather than focusing solely on optimizing PWB and PFENCE operations.

Following the findings we went and researched the implementation of PMEM devices, we managed to find a transcript of Performance Guidelines for Intel® Optane™ Persistent Memory[4], from the transcript We have discovered that the hardware operates on 256-byte ECC blocks. This implies that applications seeking to fully utilize the available bandwidth must operate on 256-byte blocks. The current generation of Intel Optane DC Persistent Memory can be configured with an interleave size of 4 kilobytes. Optimized applications can take advantage of this by being aware of the interleave size and avoiding constant writes to the same physical DIMM. This approach can significantly improve overall throughput

In addition we found a paper ["The Performance Power of Software Combining in Persistence"] that explains in detail the importance of using consecutive memory.

Using Direct Memory

The earlier version of the DFC stack was implemented using the libpmemobj-cpp library. This library acts as a wrapper around libpmemobj and offers persistent pointers, as well as scoped and closure transactions to make memory management less prone to errors. Upon further examination of the libpmemobj-cpp library, it became clear that the tools it provides do not cater to our requirements for two reasons.

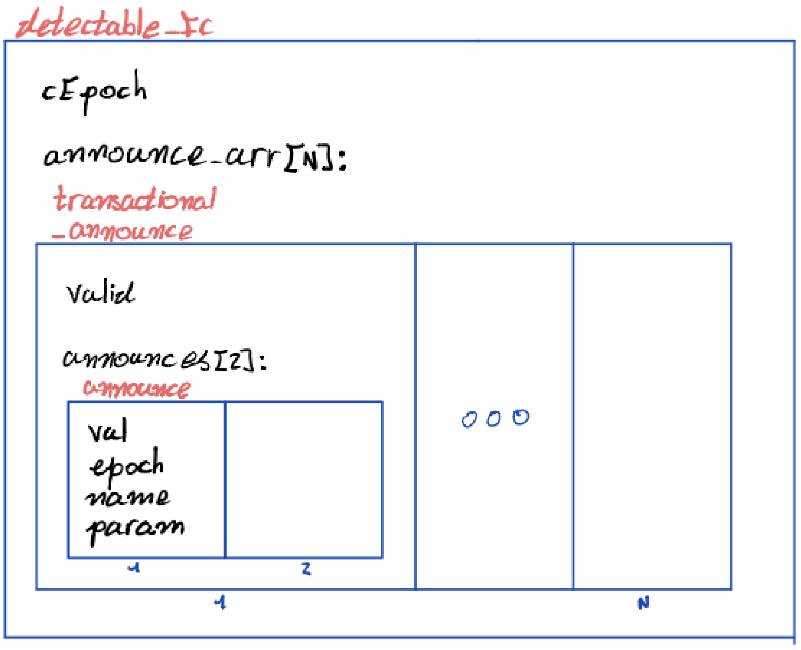
1. The use of transactional memory has no contribution in our case, as we are already responsible for ensuring data validity.

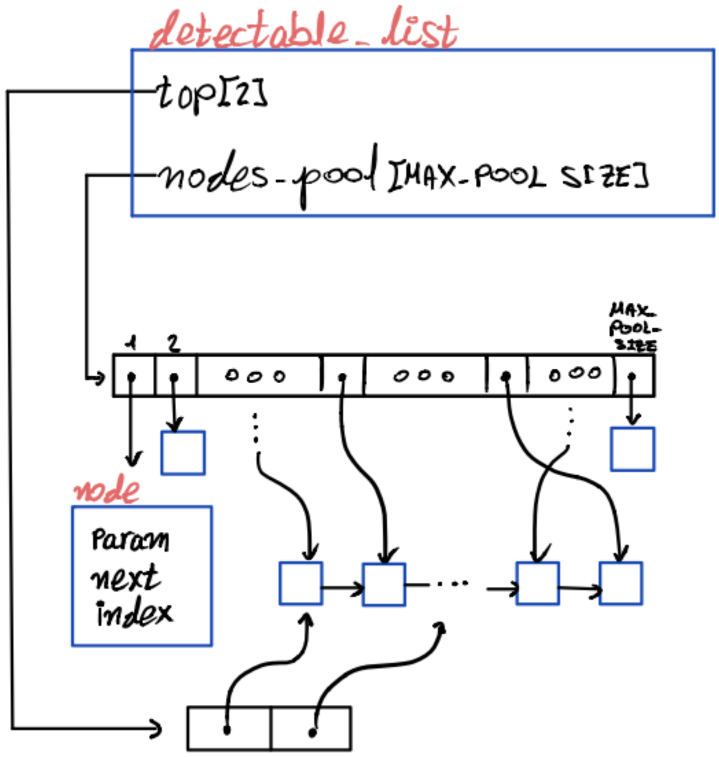
2. The implementation of members as persistent pointers has a negative impact on performance, as previously explained.

Therefore, we have decided to utilize Direct Memory instead and avoid the use of pointers, which aligns better with our needs.

We have divided the original DFC stack structure into two distinct parts: the stack Implementation, which we have maintained as-is, and the rest of the data structure. We have rewritten the latter part as a single flat object, utilizing in-place arrays instead of pointers. This was feasible due to the fixed size of the data structure.

The following scheme outlines the new structures:





To create the object in persistent memory, we have utilized the base libpmemobj library and mapped it directly to memory addresses.

This change provided multiple advantages:

1. The ability to read/write data in bulks

   As we know, the speed of reading and writing data is largely independent of its size within certain fixed ranges. Utilizing flat objects with in-place data allows us to read or write a significant amount of data in a single IO operation. This is because with flat objects, the data is stored contiguously in the memory, allowing us to access large chunks of data without requiring multiple IO operations.

2. The ability to avoid unnecessary IO’s

   To stress this impact we give the following example:

   Let us compare two options for reading a field from an object in an array

   - Using Pointer Structure:

     First, we must access the pointer to the array, followed by the pointer to the object, then the pointer to the field, and finally access the actual data. Each jump may require an additional IO operation.

   - Using Flat Objects:

     We can access the data based on its offset using a single input/output operation.

   The impact of using flat objects with in-place data becomes more significant as the depth of the nesting increases. When the data structure is deeply nested, using pointers to access data requires multiple jumps between memory addresses, which can result in a significant performance penalty due to the additional input/output operations required.

   We run the DFC stack with Flat-Obj modification as explained above, when changing the system configuration according to our previous findings. The performance achieved by the Flat-Obj modification appears in the table in the Appendix.

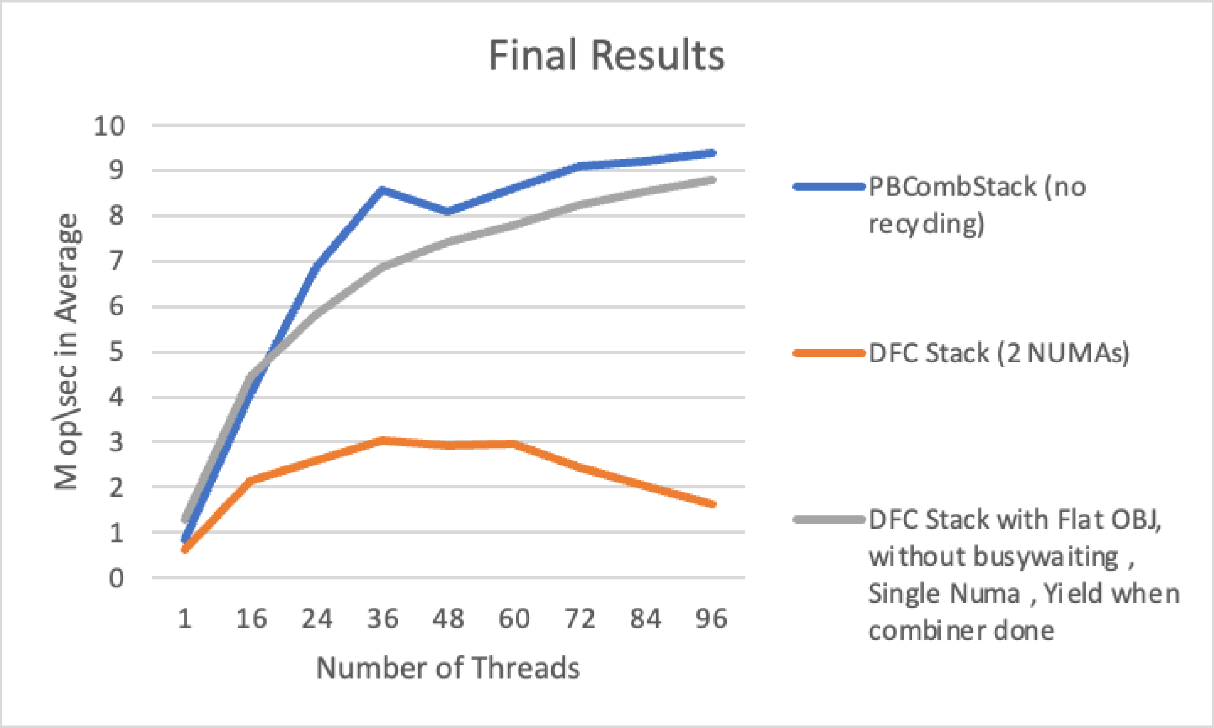
   The Best results were achieved at the following configuration:

No. threads	**1**	**16**	**24**	**36**	**48**	**60**	**72**	**84**	**96**
DFC Stack with Flat OBJ, without busywaiting , Single Numa , Yield when combiner done	1.28	4.45	5.84	6.88	7.41	7.79	8.24	8.52	8.78

[To ensure that nothing was broken due to the library change, we verified that the recovery test passed as expected.]

Final Results

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
PBcomb Stack	0.86	4.08	6.88	8.58	8.1	8.62	9.11	9.19	9.38
DFC Stack with Flat OBJ, without busywaiting , Single Numa , Yield when combiner done	1.28	4.45	5.84	6.88	7.41	7.79	8.24	8.52	8.78



As we can see, we managed to achieve performance levels very similar to PBcomb while still persisting the requests.

Our findings suggest that in order to maximize performance we should:

- Use Direct Memory and flat objects for efficient reading and writing of large amounts of data in a single IO operation and accessing data by its offset through a single input/output operation.
- Run on a single Numa to avoid synchronization costs when accessing memory.
- Avoid busy waiting to prevent unnecessary context switches and cache thrashing.
- Include a yield at the end of the combiner procedure to allow other threads to process their results and announce new requests before the start of the next epoch.

Additional Experiments:

Other things that were tested and resulted in insignificant or unhelpful results:

1. We conducted a test in which only threads waiting for the combiner on the same processor as the combiner would perform a yield, while the rest would engage in busy waiting. However, we observed a decrease in performance. Our hypothesis is that while this configuration may seem advantageous at first, it is possible that the processor on which the combiner is running is handling a smaller number of threads compared to other processors. As a result, all threads that start their epoch on the combiner's processor give up their quantum time, prompting the operating system to assign each subsequent thread to this processor. Consequently, the performance actually decreases as the other processors have a greater number of threads performing busy waiting at the same time.

2. We experimented with changing the lock mechanism by having threads perform a conditional wait for the combiner, such that at the end of the combiner run, it would wake up all the waiting threads. This approach was intended to address the issue of threads waking up before the combiner had completed its work, which can happen when the operating system chooses to. However, we observed a decrease in performance. Our analysis suggests that this may be due to the overhead of communication between threads, which takes more time than the potential savings from having threads wake up before the combiner completes its work.

   In conclusion, the use of data structures with minimal pointers is essential in developing algorithms for persistent memory. By minimizing the number of pointers, we can reduce the amount of memory that needs to be persisted, which is critical in optimizing the performance of these algorithms. Additionally, when implementing such algorithms, it is vital to consider the system configuration, including the number of NUMA nodes, CPU pinning, and avoiding busy waiting. Taking into account these factors allows for further optimization of algorithm performance and efficient use of system resources. In summary, it is crucial to carefully consider both the design of data structures and the system configuration when implementing algorithms for persistent memory in order to achieve optimal performance.

Ideas For Future Improvements:

1. Use the Flat Object structure while avoiding persisting the requests.

2. Use the same principles used to create the flat dfc object to improve the actual queue object.

   One idea we had was implementing the stack using an array instead of a linked list: Since the maximum possible number of nodes for the stack is allocated before the benchmark runs, we can use a continuous array. The two top pointers will point to different indexes in the array. Every push to the stack will write the value to the head of the array and update the top index, while a pop will reduce the top index. This approach utilizes continuous memory, which allows us to write the data to a buffer in memory and copy it to the persistent memory in a single call using pmemcpy.

3. Test the improvements suggested in this paper on other persistent and concurrent data structures. While the majority of our experiments and work focused on the stack data structure, the improvements we identified are not limited to stack implementations. These enhancements could be easily applied to other data structure implementations, and we believe they will have a significant positive impact on performance. Therefore, we encourage further testing and application of our proposed improvements in a broader range of data structures.

We believe that by utilizing option 1 and 2 the performance of the algorithm will suppress the results of the PBcomp.

References

[1] Flat-Combining-Based Persistent Data Structures for Non-Volatile Memory

[2] The Performance Power of Software Combining in Persistence

[3] Our GitHub Project

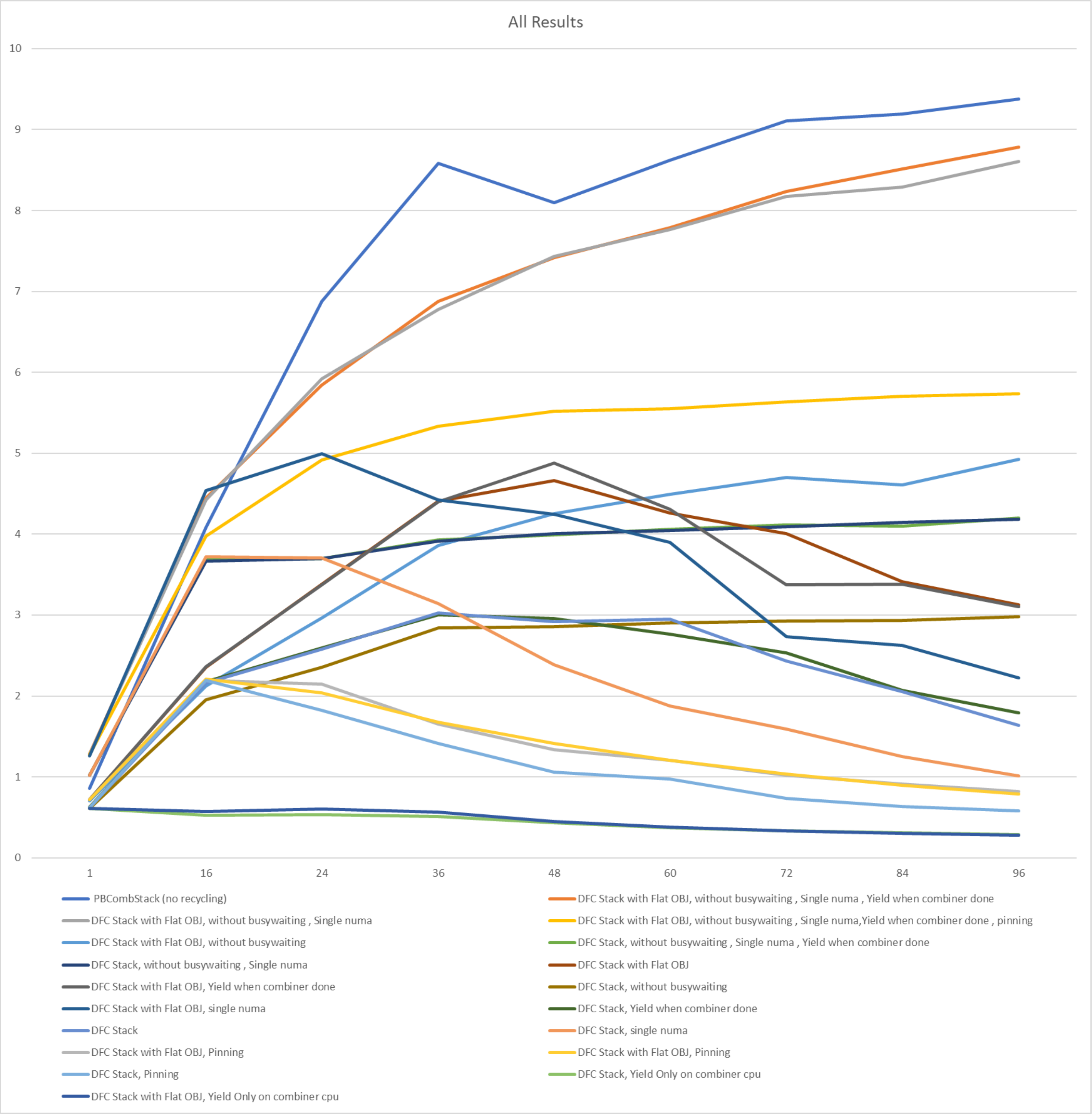
[4] Performance Guidelines for Intel® Optane™ Persistent Memory

Appendix

All Results:

Number of threads	1	16	24	36	48	60	72	84	96
PBCombStack (no recycling)	0.86	4.08	6.88	8.58	8.1	8.62	9.11	9.19	9.38
DFC Stack with Flat OBJ, without busywaiting , Single numa , Yield when combiner done	1.28	4.45	5.84	6.88	7.41	7.79	8.24	8.52	8.78
DFC Stack with Flat OBJ, without busywaiting , Single numa	1.27	4.42	5.92	6.78	7.43	7.76	8.17	8.29	8.61
DFC Stack with Flat OBJ, without busywaiting , Single numa,Yield when combiner done , pinning	1.28	3.97	4.92	5.34	5.52	5.55	5.64	5.71	5.73
DFC Stack with Flat OBJ, without busywaiting	0.70	2.12	2.97	3.86	4.25	4.49	4.70	4.61	4.93
DFC Stack, without busywaiting , Single numa , Yield when combiner done	1.02	3.69	3.70	3.93	3.99	4.06	4.12	4.10	4.20
DFC Stack, without busywaiting , Single numa	1.02	3.67	3.70	3.91	4.01	4.05	4.09	4.15	4.18
DFC Stack with Flat OBJ	0.72	2.36	3.38	4.41	4.66	4.26	4.01	3.41	3.13
DFC Stack with Flat OBJ, Yield when combiner done	0.72	2.36	3.38	4.40	4.88	4.31	3.37	3.38	3.10
DFC Stack, without busywaiting	0.61	1.95	2.36	2.84	2.86	2.90	2.93	2.94	2.98
DFC Stack with Flat OBJ, single numa	1.26	4.54	4.99	4.42	4.25	3.90	2.73	2.62	2.22
DFC Stack, Yield when combiner done	0.63	2.17	2.59	3.01	2.96	2.76	2.53	2.07	1.79
DFC Stack	0.62	2.16	2.58	3.03	2.92	2.95	2.43	2.05	1.64
DFC Stack, single numa	1.02	3.72	3.70	3.15	2.38	1.88	1.59	1.25	1.01
DFC Stack with Flat OBJ, Pinning	0.72	2.19	2.14	1.65	1.33	1.20	1.02	0.91	0.82
DFC Stack with Flat OBJ, Pinning	0.72	2.21	2.04	1.68	1.42	1.20	1.04	0.90	0.79
DFC Stack, Pinning	0.62	2.19	1.82	1.41	1.06	0.97	0.74	0.63	0.58
DFC Stack, Yield Only on combiner cpu	0.61	0.52	0.54	0.51	0.43	0.37	0.34	0.31	0.29
DFC Stack with Flat OBJ, Yield Only on combiner cpu	0.61	0.57	0.60	0.56	0.45	0.38	0.33	0.30	0.28

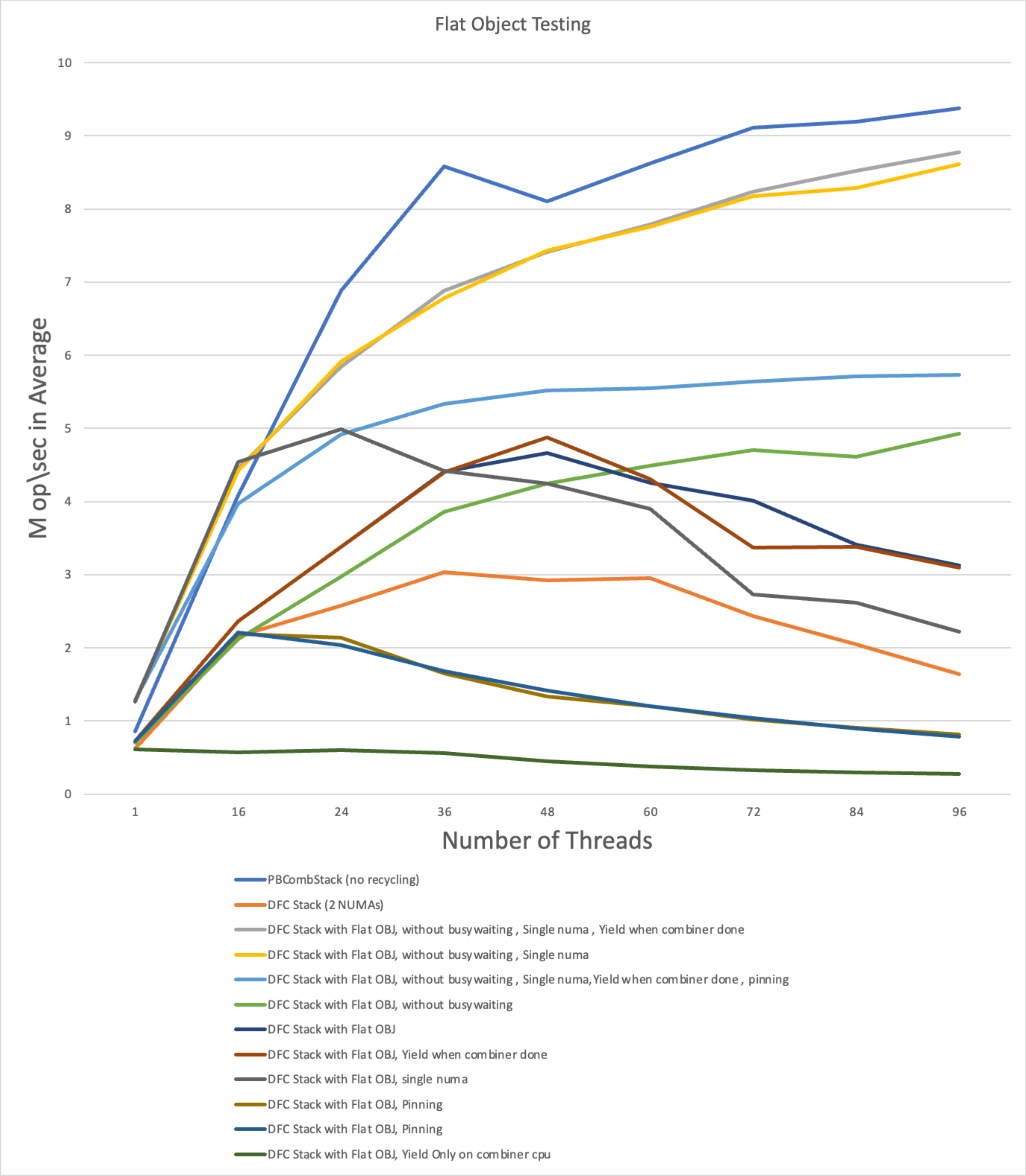
All Results Graph:



Flat Object Results:

No. threads	1	16	24	36	48	60	72	84	96
DFC Stack with Flat OBJ, without busywaiting , Single Numa , Yield when combiner done	1.28	4.45	5.84	6.88	7.41	7.79	8.24	8.52	8.78
DFC Stack with Flat OBJ, without busywaiting , Single Numa	1.27	4.42	5.92	6.78	7.43	7.76	8.17	8.29	8.61
DFC Stack with Flat OBJ, without busywaiting , Single Numa,Yield when combiner done , pinning	1.28	3.97	4.92	5.34	5.52	5.55	5.64	5.71	5.73
DFC Stack with Flat OBJ, without busywaiting	0.70	2.12	2.97	3.86	4.25	4.49	4.70	4.61	4.93
DFC Stack with Flat OBJ	0.72	2.36	3.38	4.41	4.66	4.26	4.01	3.41	3.13
DFC Stack with Flat OBJ, Yield when combiner done	0.72	2.36	3.38	4.40	4.88	4.31	3.37	3.38	3.10
DFC Stack with Flat OBJ, single Numa	1.26	4.54	4.99	4.42	4.25	3.90	2.73	2.62	2.22
DFC Stack with Flat OBJ, Pinning	0.72	2.19	2.14	1.65	1.33	1.20	1.02	0.91	0.82
DFC Stack with Flat OBJ, Pinning	0.72	2.21	2.04	1.68	1.42	1.20	1.04	0.90	0.79
DFC Stack with Flat OBJ, Yield Only on combiner cpu	0.61	0.57	0.60	0.56	0.45	0.38	0.33	0.30	0.28

Flat Object Results Graph:


Flat Object Testing
M op\sec in Average
Number of Threads
PBCombStack (no recycling)
DFC Stack (2 NUMAs)
DFC Stack with Flat OBJ, without busywaiting , Single numa , Yield when combiner done
DFC Stack with Flat OBJ, without busywaiting , Single numa
DFC Stack with Flat OBJ, without busywaiting , Single numa,Yield when combiner done , pinning
DFC Stack with Flat OBJ, without busywaiting
DFC Stack with Flat OBJ
DFC Stack with Flat OBJ, Yield when combiner done
DFC Stack with Flat OBJ, single numa
DFC Stack with Flat OBJ, Pinning
DFC Stack with Flat OBJ, Pinning
DFC Stack with Flat OBJ, Yield Only on combiner cpu