Brown University

Construction of Buffered Durable Linearizable Systems

by Christina Stepin

Undergraduate Honors Thesis in Computer Science

Approved by the Examining Committee:

Prof. Maurice Herlihy, Prof. Ugur Cetintemel
Thesis Advisors

Saturday 19th April, 2025

Contents

Chapter 1

Introduction

With the prevalence of large computing clusters, fault tolerant system have never been more important. Some work on fault tolerant systems uses the safety property of buffered durable linearizability [7]. Informally, this is a guarantee that in case of a crash, the state reconstructed from non-volatile memory reflects a consistent prefix of the interrupted computation.

However, the majority of work focuses on buffered durable linearizability for individual data structures, and not buffered durable linearizability for the entire system; this is difficult to guarantee because the property is not composable. But one paper that does construct a buffered durable linearizable system is Montage [14]. This thesis focuses on the design of Montage and its limitations – including how certain components of Montage are hard to generalize and may be difficult to use for the broader computer science community. We propose several adaptations to Montage, which makes it easier to use while ensuring the runtime remains comparable to the original Montage design.

The biggest limitation of Montage comes in its copying mechanism – it uses the same mechanism for all data structures. We discuss why this is an issue and suggest alternatives to improve runtime and adaptability. We also offer some further thoughts on next steps in investigating buffered durable linearizable systems.

Chapter 2

Background

It is now common for companies to rely on the computation power of thousands and even hundreds of thousands of machines. This means that a crash is not only inevitable, but increasingly common. Companies need to plan for failure of their machines. Once this failure occurs computer scientists need to know how to recover the state of the computer, redo any lost operations, and then continue running the application. To this end, we define several correctness properties of crash tolerant systems.

Non-Volatile (Persistent) Memory (NVM) survives crashes. Updating the non-volatile memory is an expensive operation.

Linearizability ensures operations on objects appear to happen atomically between when they are called and when they return. For example, if a queue contains a "3", a "3" must have been pushed at some point. If such an instruction exists, then we can say that the state is linearizable. Otherwise, if no such instruction exists, then the memory state is not linearizable, and is thus not correct in our definition.

Durability is the property of memory that survives crashes.

Durable linearizability guarantees that all of a user's changes are reflected in NVM in the case of a crash.

Durable linearizability is an attractive property in that it ensures that crash recovery is trivial; in case of a crash, one can just use the memory state in NVM. Durable linearizability is also composable [7], meaning that two objects that are each durable linearizable are also together durable linearizable. Guaranteeing this property is expensive, as all changes would have to be pushed to NVM immediately after they are enacted. Since even a single push to NVM is quite expensive, continuously pushing changes is impractical. Instead, a slightly weaker and much cheaper property is used.

Buffered Durable Linearizability (BDL) guarantees that in the event of a crash, a linearizable prefix of operations is reflected in NVM. This is cheaper because operations can be batched before being pushed to NVM. However, there is a tradeoff. BDL guarantees lower latency while the system is running, but higher latency to recover the system in case of a crash. Users potentially have to redo some operations before the crash [7].

BDL is not composable. For example examine two queues that are individually BDL. Say we pop the top element from one queue and push this element into the other queue atomically. If there is a crash after this operation, the element could appear in neither of the queues. One queue would have its constant prefix of history after the element was popped, and the other would have its constant prefix before the element was pushed in. The popped element would effectively disappear. So, while the queues individually maintain the property, the system as a whole would be in an inconsistent state, so would not be BDL.

While there have been papers published previously on ensuring that individual objects are BDL like lock-free queues [4], hashtables [12], and radix trees [10] there is not much work on creating a universal construction to ensure given system is BDL besides Montage [14].

Copy-on-Write (CoW) is a method typically used within operating systems to create efficient copies of data structures. The efficiency is derived from ensuring that the data structure is only copied when it is modified. Some CoW methods also specify that only the regions that are modified need to be copied. For example, if a hashtable employs copy-on-write and one key-value pair is updated, only this updated key-value pair needs to have a copy generated. The new key-value pair can instead be inserted in memory to the location of the previous key-value pair. Before modification, if multiple users (or threads, processes) request access to the same resource the system can instead give them all pointers to the same instance of an object. If all accesses are reads, this saves a significant amount of memory for the computer.

Overall, CoW boosts performance and memory-usage, by ensuring that reads can occur faster, and delaying unnecessary copies until a write in the system occurs. Both Montage (the model discussed in the paper) as well as our system rely on instances of CoW.

Chapter 3

Montage

Montage is a system guaranteeing BDL. To describe how Montage works, we will describe the important elements within the system, before discussing its limitations.

3.1 System Design

Montage utilizes a snapshot design to ensure BDL. What this means is that the NVM always contains a snapshot of the system with a consistent prefix of the system's operations. So when threads update objects within the system, they operate on copies of the objects. Each of these copies corresponds to a certain "epoch" within the system running time. At a designated point in time (such as after an instruction is read or after a certain amount of time), the epoch of the system is incremented, and the volatile memory copies persisted to NVM. In case of a crash, the current copy of the system is recovered from NVM, and it becomes a starting point for new volatile copies. Memory used to create copies before the crash is freed.

3.2 Elements

3.2.1 Epochs and the Epoch-Advancing Mechanism

Montage defines **epochs** as a cut within execution order, where the boundaries of epochs represent states of Montage that are consistent. Snapshots can be defined by epochs: each snapshot looks like the system at the end of an epoch, and the persistent snapshot is updated to look like an epoch. Practically, the system tries to ensure each epoch lasts between 10 and 100ms. Montage achieves consistency by allowing the

user to specify the beginning and end of a transaction with *BEGIN_OP* and *END_OP*. There is a system guarantee that a transaction will never span multiple epochs, and epochs can instead contain multiple complete transactions.

Montage guarantees there are at most 2 active epochs at any given point in time, and achieves this by persisting everything in epoch e when the clock ticks from $e + 1$ to $e + 2$. More concretely, at the end of a given epoch, it ensures that all persisting operations are complete before incrementing the epoch clock.

3.2.2 Payloads

Montage utilizes persistent blocks of memory (payloads) to store the state of various objects. To reduce memory usage, each payload only stores necessary information. For example, only a set's elements and not its lookup method (like a tree structure or a skiplist) are stored. Each item in a queue would belong to a different payload, and its ordering might be stored by labeling the payloads with the indices of the elements.

Because there are at most two active epochs at any given moment, when the clock ticks over from epoch $e+1$ to epoch $e+2$, all payloads created in epoch e are immediately persisted. If Montage discovers payloads from epochs that are two or more behind the current epoch, it deletes these older payloads to satisfy the property of Montage only ever maintaining information from two epochs at a time.

3.2.3 Anti-payloads

In case of a crash, the system must know which temporary copies were created in volatile memory, so that the memory can be freed. In the case that an epoch was in the middle of promotion during a crash (ie, Montage started persisting some payloads from a specific epoch), it is impossible to tell which payloads were or were not already persisted. So all payloads from this epoch would be unhelpful to the system and are freed. Montage keeps track of the payloads allocated in a specific epoch by ensuring that whenever a payload is created, an **anti-payload** associated with the payload's epoch is also generated. An anti-payload is only created when a payload is deleted in an epoch after the payload was created (otherwise, the payload could just be deleted right away).

In case of a crash, the system deletes all anti-payloads in epochs following the epoch of the persisted copy. Anti-payloads are reclaimed an epoch after their corresponding payloads are reclaimed. For example, payload from epoch e must be reclaimed when the epoch clock ticks from $e+1$ to $e+2$, and the anti-payload is reclaimed when the epoch clock goes from $e+2$ to $e+3$.

3.2.4 Copying Mechanism

When a write is issued, Montage copies the payload where the applicable data is contained. If the payload has already been copied within the epoch, the data can be modified in place. Otherwise, the entire payload is copied and then modified. Upon a copy being promoted to NVM, every payload replaces its instance in the previous NVM snapshot.

3.2.5 Copy Promotion

We were not able to deduce exactly how Montage promotes and rolls back payloads into NVM when it wants to update the epoch. A cursory glance through the code also did not demystify their exact methodology.

The Montage paper describes that at epoch boundaries, payloads payloads are pushed to NVM at once. It is unclear what happens if a computer crashes in the middle of such an update. Because the act of pushing payloads into NVM involves replacing the corresponding payloads with ones just modified in volatile memory, it seems that a back-up copy of the previous payloads must be stored somewhere. This is to ensure that in case a crash, the system knows which payloads to keep and which ones to rollback.

Because Montage guarantees BDL, they must have a way of handling crashes within updates. This includes a rollback mechanism to preserve BDL. We assume that Montage then stores back-up copies of payloads that they free after a successful epoch update.

3.3 Limitations

Montage is innovative and adaptable to many different types of data structures. Certain aspects of its design, like the creation of anti-payloads for every epoch, were very effective. Despite these strengths, we would claim its weaknesses come from its lack of adaptability.

3.3.1 Payloads

We believe the payload structure is burdensome, because it requires the programmer to figure out the "minimum" amount of information to be stored in a payload. This requires in-depth knowledge of any data structure to be persisted. It is difficult to pre-

sume that all users know how to reduce any data structure into a payload-like structure. Moreover, whenever a data structure is slightly tweaked, the payload representation also needs to be adapted.

We propose storing the entire data structure in NVM. We believe that this does not incur too great of a space cost, and achieves important benefits. First, storing the entire data structure may be more accessible to users, as they are not required to possess in-depth knowledge of the construction of data structures. Second, the Montage paper notes that upon a crash there is overhead to reconstruct a data structure based on the payloads. With our proposal, a user could directly reference the objects as they are in NVM. Finally, we believe that the transient data that Montage leaves out of payloads could potentially speed up copying operations.

3.3.2 Epochs and the Epoch-Advancing Mechanism

Montage guarantees that at any moment in time, the snapshot in NVM is at most 2 epochs behind the copies in the volatile memory. We claim that this may result in pushing to NVM at an unnecessarily frequent rate. Instead, we allow the user to define a parameter for how often they may want a copy to be updated, and then try to create epoch boundaries based on this user-defined notion.

The advantage of this approach is tunability – users choose the parameter that works best for their application. Also, because we assume that crashes are infrequent, average run-time speeds up.

However, we acknowledge that this poses an issue of inadequate space in memory. There may be a potentially unbounded number of copies of the system. Since each copy takes up memory, there may be an issue if more memory is required than the computer can provide. To avoid this issue, we added another parameter to determine how often snapshot copies are promoted to the snapshot in persistent memory. If the system notices that too much memory has been used, it can stall until an appropriate number of epochs have been promoted to be in NVM and their volatile memory is relinquished.

To add to this point, the Montage papers discuss how the fences that ensure there are never more than two active epochs constitute a bottleneck. We propose to fine-tune this design by implementing work-stealing among the threads. Specifically, we initialize two groups of threads; one set of "working" threads and one set of "persisting" threads. The working threads copy payloads and make changes to them, defining new epochs as necessary. Persisting threads concurrently persist changes within an epoch to a copy. At the end of the epoch, they wait at a memory barrier. When all persisting threads rejoin on the barrier, the system knows that all changes within an

epoch have been made to the temporary copy. This copy is promoted to the new NVM snapshot.

If persisting threads are slower than the working threads, we allow the working threads to help persist payloads from earlier epochs to NVM. The system can tune the number of working and persisting threads depending on which operation is lagging. This ensures that there is a balanced approach to enacting both steps of this operation. This provides a speed-up over Montage, because threads are guaranteed to constantly either be at a memory barrier or persisting to ensure that the memory barrier can be crossed.

3.3.3 Copying Mechanism

Montage specifies that each payload is copied into volatile memory whenever that payload is modified. This payload copy is tagged with the current epoch number. If the same payload needs to be updated multiple times within the same epoch, the change can be done in place (the same copy of a payload can be updated).

We agree that if a data structure is a simple set or hash-table, Montage's solution holds water. However, more complex data structures suffer from this solution because of the cost of reconstruction necessary in case of a crash. Our argument is that we can have the same runtime as Montage when the system is running correctly, and we ensure that the system can recover faster than Montage can.

The main idea is that there can be significant speed-ups in copying if a user specifies how they want a certain data structure to be copied. These speed-ups are significant in that Montage's main bottleneck comes creating copies of the payloads. If one is able to speed up this copying portion, then the whole system becomes faster.

We explore this more in the chapter on copying (section 5).

3.3.4 Copy Promotion

We assume that we have a temporary snapshot and a trusted snapshot in NVM. Both are snapshots of the system at some point in time. However, we ensure that when payloads are persisted and replace previous payloads, they replace the payloads in the temporary copy. If all payloads are successfully persisted, the temporary copy becomes the trusted copy. The original trusted copy is discarded (its state is out-of-date), and a temporary copy is created based on the new trusted copy.

If the system crashes in the middle of an update, the temporary snapshot may not be reliable, so will be discarded. Users only need to reference on the trusted snapshot.

This system requires no rollbacks in case of a crash, as only the trusted snapshot needs to be referenced to preserve the BDL property. We agree that there may be more overhead from maintaining multiple snapshots in NVM and promoting the temporary copy to the trusted copy. However, this trade-off is acceptable – either way, the main bottleneck is persisting payloads. Using more memory to ensure a faster recovery is a reasonable tradeoff.

Chapter 4

nbMontage

The Montage group released a paper a year later with improvements for non-blocking data structures. They called this system **nbMontage** [1].

The main differences between Montage and nbMontage were how they dealt with keeping a transaction within a single epoch. In Montage, users could specify a "begin_op" and "end_op" operation. The epoch would be verified with a "CAS_verify" operation. In nbMontage, all three of these operations are exchanged out for a "lin_CAS" operation – a linearizable compare-and-swap which performs a double-compare-and-swap [5] with the epoch number and expected value.

nbMontage is outside the scope of this project, though our model should generally work for non-blocking data structures as well.

Chapter 5

Improvements to Copying

The central improvement in this thesis is Montage's copying mechanism.

As a brief reminder, Montage implements a payload system where only the core parts of a data structure's information are copied over. These are stored in blocks of memory called payloads and any time there is any modification to a payload, the entire payload is copied over. This method depends on the user specifying parts of the abstract state of the data structure to persist. This may be difficult for users who are not experts in the data structure. We believe that something more reasonable is for users to specify a copy-on-write function for the object.

This method would not change how Montage implements copying on data structures that don't rely on an underlying order (a set or hashmap) as a copy-on-write would still consist of copying individual elements or key-value pairs. However, this would affect data structures that rely on a lookup method. For example, a B+-tree [13] or skiplist [9].

Montage persists these objects by persisting the individual items (like the leaf nodes in a B+-tree or elements in a skiplist). Then, in case of a crash, it reconstructs the object based on the persisted payloads. This becomes an expensive operation – not only is the system expected to have underlying knowledge of how the specific tree or list stores elements, but in case there are transient data structures added on-top of these objects, Montage has extra work of adding these structures back.

This is why we claim our method of storing the entire object in NVM is especially powerful. First, objects may have transient data structures that allow them to have CoWs occur with lower latency [9]. Second, our recovery function is much faster than the recovery of Montage. For all data structures in Montage – including those like sets and hashmaps, which may not have an underlying look-up method – there is still the overhead of recreating what the object looks like in case of a crash. In our model,

recovery just looks like pulling the persistent version of the object from NVM. This makes the recovery time constant. To restart the system, one simply needs to copy the system into volatile memory and continue operations from there.

Another strength of our model is that users can specify different CoW functions for their objects, and choose the tradeoffs they want to accept [2, 17]. This ensures data structures with an underlying lookup method like B+-trees and skiplists to have speed-up with more specialized CoWs.

Besides CoWs for specific objects, we also claim that with improvements to general CoWs, users can choose the tradeoffs that work better for their purposes. If users want to maximize cache sharing and minimize memory [15], batch CoW updates [17], increase transparency and debuggability [6] they are able to do so through our model.

Chapter 6

Proofs of Correctness

We structure our proofs of correctness similarly to how the Montage papers structure theirs. Specifically, we need to guarantee that our design is still linearizable, BDL, and satisfies liveness. We will reference a majority of their work for our own proofs of correctness.

6.1 Linearizable

We rely on the proof of linearizability used in Montage.

They specify that the point of linearizability is always guaranteed to be between any two epochs. This means that in case of a crash, we know that the data present in NVM is linearizable, because the data is reflective of the state of a system after the end of an epoch. This means that our change to the epoch clock will not affect the linearizability of the object, as we do not change the points of linearization – we simply say that more epochs (and so more points) might exist as a result.

Within the epochs themselves, we also need to state that the update to the NVM snapshot copy will preserve linearizability. As specified in Montage, we guarantee that any individual objects that are newly persisted are done so in a safe way; all dangling pointers are updated. We also specify that our model has a temporary snapshot on which persistence is done. Only at the ends of epochs is this temporary snapshot updated to be the main NVM snapshot. This guarantees that in case of a crash, only a snapshot that has been fully persisted (so much look like the state at the end of an epoch) is relied upon.

6.2 Buffered Durable Linearizable

If there is a crash, we have several situations. If the epoch number reflected by the NVM is 0, then we know that there have been no snapshots fully persisted, so we safely restore the system to its initial state.

Otherwise, all active payloads and anti-payloads are discarded, and the state of the system is recovered from the persisted snapshot of the system in NVM. A copy can be made on this snapshot in volatile memory, and operations continue as usual on this volatile copy.

By the definitions of epochs, we know that what is reflected in NVM is the state of the system at the end of an epoch. Since an epoch includes some amount of consecutive operations, the state will then represent what the system looks like after exactly these operations have finished. So, the copy in NVM would then be all operations that linearized and finished execution up to the end of the epoch boundary. This must then be a consistent prefix of operations that returned and linearized prior to the crash and is then BDL.

6.3 Liveness

The Montage paper specifies that their system only has one loop. At the end of each loop, the epoch clock must have advanced, and is only able to do so if some operations were previously completed. Thus, Montage proves liveness, because they guarantee the epoch increments on each iteration of the loop, and this itself implies that at least one operation was successfully completed.

Since our changes to Montage's design do not affect the coding structure, we can similarly claim liveness. As we also guarantee that the global epoch timer is only advanced from epoch e to $e + 1$ when at least one operation was completed in epoch e, one iteration of the loop similarly implies progress in our model. Thus, our model still retains latency.

6.4 Latency

We claim that the latency of our system should be at least as fast as Montage when it is running, and is faster on recovery.

While the systems are running, our system should have at least an equal runtime

as Montage. The main differentiator in our system is the frequency of persistence, and the copying methods of the data structures.

Because our model persists by a user-specified amount, if this amount is less often than Montage we claim a speed-up. One of the most costly operations will be the persistence to NVM, so if our model does this less often, we expect a speed-up.

Second, based on the user-defined copy functions of CoW, we expect at least similar runtime, if not a faster latency. Either, we claim, our methods of CoW will be the same as that of Montage. Or, if a user inputs a specific CoW function for their system, we expect a speed-up as guaranteed by the specified CoW.

In case of a crash, Montage is forced to rebuild their data structures, including adding back transient data structures. As we store the entire data structure in NVM, recovery simply involves pulling the data structure from NVM. So, since our model does less operations than Montage, we know that recovery will be faster.

Thus, our model should perform at least as quickly as Montage.

Chapter 7

Experimental Results

Since our model changes individual components of the Montage model, we conducted smaller tests based on individual components. We evaluated the results based on their latency. Montage also specifies that they perform their tests on a queue and hashmap filled with 0.5 million items. We conduct a similar test, and also provide a qualitative analysis of how we expect our model to compare to theirs on other data structures.

Upon recovery, Montage mentions these data structures may be represented as payloads in NVM that upon a crash are reconstructed.

To simulate this, we represented a hashtable as a queue with CoW on every entry, where each of these entries stores a pair. We compared this against what our model would generally look like – a hashtable with CoW for every key-value pair. Similarly, each of these had 0.5 million entries and had a series of reads and writes performed. We also tested on queues. For Montage, we had a queue with CoW on every entry, and to simulate recovery, constructed a new queue from the CoW queue. For our model, we had a queue with CoW on every entry and to simulate recovery, simply returned this same queue.

We simulated having a general runtime by starting a mutation phase where 100,000 get and set operations were performed. We simulated the recovery process (the extraction phase) by simply implementing what each system would do – ours just returns the object from NVM, while Montage recreates the object.

We averaged 50 runs for these results.

Unsurprisingly, in the queue example, our queue seemed to do significantly better than the Montage queue. The runtime phase timed to be 5.8ms for both queues. The recovery phase for our model timed to be 8.28 ms, while for Montage timed to be 65.42 ms – an 8x slowdown.

For the maps, we found that our runtime timed to be 149.22ms and Montage's runtime was 80.86ms. However, our recovery was 6ms, and theirs was 142.24ms – a 23x slowdown.

We show this example to demonstrate that our potential slowdown during runtime would not be too great. The potential speed-up for recovery, however, massively outperforms Montage. If a user also specifies a more efficient CoW specific to hashmaps, we expect to have an even greater result.

We also created an integration test, where we tested both the effects of persisting less often coupled with our more specific CoW during runtime and recovery. Although we were unable to create an exact copy of Montage and our model, we felt that these tests were representative of what the actual results might be like.

We simulated persistence with msync, memcpy, and mmap where we created a temporary file to act in place of NVM. At the beginning of the program execution we mmap-ed a file to simulate the location of the NVM. When actually pushing changes, we memcpy-ed the changes into the file, and then msync-ed to represent flushing data into NVM.

Montage mentions various epoch lengths that they used when testing their system. To simulate a rarer persistence, we persisted our system 10 times as rarely as the Montage set-up.

We ran experiments on the two queues, as described above. With our model, our queue averaged to 99.94ms during the mutation phase, and 8.38ms extraction phase. Montage's model averaged to 838.22ms with the mutation phase and 62.3ms for the extraction phase. Our results are as expected – with less persistence, we achieve a faster result. With our more specialized CoW and method of storing the object in memory, we have a faster recovery time.

We repeated the experiment on the two models of the hashmap. We got that our map model had a mutation phase that averaged to 26.86ms, and an extraction phase that average to 6.22ms. Montage's mutation phase averaged to 102.64ms and extraction was 75.46ms. Here, with the delayed persistance, we see a slowdown of both the mutation and extraction phases with Montage's model.

Finally, we wanted to point out that papers with specialized CoW structures claim they receive a non-trivial speed-up when copying. While we were not able to formally evaluate it in this paper, we would imagine that more specialized data structures would achieve a great speed-up. Several specialized CoWs that we examined could potentially speed-up probabilistically balanced tree [16] and B+-trees [8].

These results indicate that tunability on the part of the copy function and the amount of persistence surpasses Montage in latency on (potentially) both the standard

execution and recovery phases. There are potentially more speed-ups to be gained from specialized CoW depending on the system and usage. We take this to indicate that a user specifying a copy-function would perhaps be more beneficial, than the user instead specifying how a system should be stored in memory.

Chapter 8

Potential Extensions

We bring up possible continuations of the project.

8.0.1 User Interface

This paper was a discussion of how Montage – a generalized model for adding the BDL – could be improved. This thesis mostly discussed well-known data structures. However, we acknowledge that user-defined data structures are also an important system to discuss.

We propose that any user-defined data structure could be BDL if the user specifies a copy-on-write and persist function. The two most important steps in Montage and our model are how to define the payloads and how to copy them over. If a user specifies them for any structure, then our model could simply use the user-specified functions to persist the regions they modify as specified by their "persist" function, and copy it the way they specify to copy it.

This would be part of an interface wrapping around any user-defined structures. We did not have enough time to properly investigate what this interface would look like, or any other attributes it would need to have. However, we believe these two functions are sufficient to integrate it into our model.

8.0.2 Extending Log-Free Ideas

We also read several constructions that turn a specific data structure (as opposed to a general system) into BDL that use attributes of the structure itself to hide away the information necessary to recover the model upon a crash [3]. We believe that there could be extensions in our project where the user could also specify a recovery

function. From papers in the literature, it seems that this would also decrease the time necessary for recovery.

8.0.3 Restarting After Crash

We also discussed how to begin operation again after a crash; the majority of papers in the literature only specify how to recover state. However, there are some general papers also discussing how to restart and continue on operation [4]. This would require either storing each thread's state at epoch boundaries, or dynamically finding epoch boundaries in such a way that the threads would have little to no state to remember. There would also be a question of tying a given epoch number to an instruction in the code (essentially an epoch boundary) so that the computer would know where to continue operation after a crash.

8.0.4 Epoch-Clock and Snapshotting

We examined several methods of improving the efficiency of individual components. One such paper did investigation on the most efficient methods of updating the epoch clock in a multi-threaded system [18]. This also helped ensure that threads began writing to new epochs together, and would have relatively up-to-date payloads. Given more time, we would think about the best way to ensure consistency between threads in terms of the epoch-clock. There is also research based on the Montage model, arguing that the snapshot mechanism could be made more efficient [11].

8.0.5 Testing

Finally, we believe there are several places in which our model would outperform Montage. It would be instructive to conduct more experiments to verify this claim.

Overall, this research area is full of interesting open questions. Efficiency and adaptability could be improved in all or almost all of the domains.

Chapter 9

Conclusion

Guaranteeing a consistent and reliable method of restoring state after a crash has never been more important. Solving this problem is of utmost importance, because all computers have the potential to crash. An efficient restoration ensures that important data and systems can continue running quickly after a system crash.

As discussed in this paper, a majority of the literature in this domain has investigated how to make individual data structures BDL. The primary model that investigates a generalized solution to this problem is Montage. However, Montage relies on users specifying the minimum underlying component of a data structure. We believe that having users specify a copying method, and instead keeping the entire data structure in NVM provides better tradeoffs. Like discussed above, this guarantees that copying and recovery are, on average, achieved with lower latency. We also provide better tunability to the system by letting users specify how often they want to persist up to a threshold determined by memory capacity.

This thesis demonstrates that smaller changes can make a big difference to both performance and usability.

Bibliography

[1] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael Scott. "Fast Nonblocking Persistence for Concurrent Data Structures". In: May 2021. DOI: 10.48550/arXiv.2105.09508.

[2] Shimin Chen and Qin Jin. "Persistent B + -trees in non-volatile main memory". In: *Proceedings of the VLDB Endowment* 8 (Feb. 2015), pp. 786–797. DOI: 10.14778/2752939.2752947.

[3] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. "Log-free concurrent data structures". In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 373–385. ISBN: 9781931971447.

[4] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. "A persistent lock-free queue for non-volatile memory". In: *ACM SIGPLAN Notices* 53 (Feb. 2018), pp. 28–40. DOI: 10.1145/3200691.3178490.

[5] Timothy Harris, Keir Fraser, and Ian Pratt. "A Practical Multi-Word Compare-And-Swap Operation". In: vol. 2508. May 2003. ISBN: 978-3-540-00073-0. DOI: 10.1007/3-540-36108-1_18.

[6] David Hildenbrand, Martin Schulz, and Nadav Amit. "Copy-on-Pin: The Missing Piece for Correct Copy-on-Write". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 176–191. ISBN: 9781450399166. DOI: 10.1145/3575693.3575716. URL: https://doi.org/10.1145/3575693.3575716.

[7] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model". In: vol. 9888. Sept. 2016, pp. 313–327. ISBN: 978-3-662-53425-0. DOI: 10.1007/978-3-662-53426-7_23.

[8] Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, Sami Ben-Romdhane, Mohiuddin Qader, and Jung-Sang Anh. "Jungle: Towards Dynamically Adjustable

Key-Value Store by Combining LSM-Tree and Copy-On-Write B + -Tree". In: June 2019.

[9] Tadeusz Kobus, Maciej Kokociński, and Paweł Wojciechowski. "Jiffy: a lock-free skip list with batch updates and snapshots". In: Apr. 2022, pp. 400–415. DOI: 10.1145/3503221.3508437.

[10] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. "WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 257–270. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

[11] Mohammad Moridi and Wojciech Golab. "Snapshotting Mechanisms for Persistent Memory-Mapped Files". In: June 2024, pp. 1–9. DOI: 10.1145/3663338.3665832.

[12] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael L. Scott. "Dalí: A Periodically Persistent Hash Map". In: *International Symposium on Distributed Computing*. 2017. URL: https://api.semanticscholar.org/CorpusID:1404820.

[13] Ohad Rodeh. "B-trees, shadowing, and clones". In: *TOS* 3 (Feb. 2008). DOI: 10.1145/1326542.1326544.

[14] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael Scott. "Montage: A General System for Buffered Durably Linearizable Data Structures". In: Sept. 2020. DOI: 10.48550/arXiv.2009.13701.

[15] Xingbo Wu, Wenguang Wang, and Song Jiang. "TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers". In: *Proceedings of the 6th Asia-Pacific Workshop on Systems*. APSys '15. Tokyo, Japan: Association for Computing Machinery, 2015. ISBN: 9781450335546. DOI: 10.1145/2797022.2797024. URL: https://doi.org/10.1145/2797022.2797024.

[16] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Ooi, Sheng Wang, and Xiaokui Xiao. "Analysis of Indexing Structures for Immutable Data". In: June 2020, pp. 925–935. DOI: 10.1145/3318464.3389773.

[17] Yang Zhan, Alex Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald Porter, and Jun Yuan. "Copy-on-Abundant-Write for Nimble File System Clones". In: *ACM Transactions on Storage* 17 (Jan. 2021), pp. 1–27. DOI: 10.1145/3423495.

[18] Xiong Zheng and Vijay Garg. "An Optimal Vector Clock Algorithm for Multithreaded Systems". In: July 2019, pp. 2188–2194. DOI: 10.1109/ICDCS.2019.00215.