Persistent Data Structures for Text Editors

Pramod Sathyanarayana, Ranganath Selagamsetty, Chetan Shukla, Jie Tong
Department of Computer Sciences
University of Wisconsin-Madison
{psathyanaray, selagamsetty, cshukla3, jtong36}@wisc.edu

***Abstract*— Byte-addressable, Non-Volatile Random Access Memory (NVRAM) is a new and promising technology as it provides DRAM-like performance, disk-like capacity, and persistence. These kinds of memories such as Intel Optane persistent memory, phase change memory (PCM) and memristor have opened the doors for research all across the system stack. In this work, we leverage the aforementioned unique properties of NVRAM for text editors, a user-space application that can reap benefits of this new technology. Existing text editors will gain from the usage of NVRAM as its features address the shortcomings of their current, volatile implementations. Prominent among these shortcomings are: memory-inefficiency of text editors which leads to a large memory footprint, data consistency issues on system crashes and long initial start-up times as data needs to be loaded from disk. Through this work, we port the underlying volatile implementations of the most common data structures used to build text editors to versions that reside completely in a persistent storage device. Our evaluations show that the persistent data structures perform reasonably well, providing an additional advantage of better consistency guarantees over their volatile counterparts and are a way to go forward.**

I. Introduction

With the advent of Intel's Optane memory technology, fast, byte-addressable Storage Class Memory (SCM) devices are commercially viable. However, due to the asymmetry in the latency of reads and writes [21] and other unique characteristics of these devices, there has been much research in the way of determining how to best leverage this new technology. Previous works have examined various ways to incorporate fast SCMs, exploring the replacement of DRAM altogether with this higher-density device [15], while others have proposed designs that place this technology directly in front of disk in a cache-like manner [14]. While both of these design points have merits, they are both large-scale changes that affect the entire ecosystems of computing. As such, any change of this scale will meet resistance before becoming the common practice.

To address this at a finer granularity, we present a specific user-space application that uniquely leverages the many benefits of NVRAM: text editors. Existing text editors have three main drawbacks as they currently stand:

First, their underlying data structures are optimized for performance and are often not space-efficient, incurring a large memory footprint that scales with the size of the file(s) being operated on. While the performance of the text editor itself may be efficient, the large memory footprint creates serious contention issues for memory from a system perspective. As an example, consider the rope data structure that is commonly used by Sublime [17] and Vim [13]. While this data structure guarantees O(log N) time for all operations, the space complexity is O(N), which is simply untenable (where N is the length of the rope) [20].

Second, they require save commands being called either explicitly for an initial save or at regular time intervals through "auto-saves", thus exposing them to consistency issues on crashes. From a usability standpoint, this exposes a representational gap, as every user action is inherently 'persistent', yet building implementations on volatile memory devices like DRAM lend themselves to these consistency issues. A more direct approach would be to explicitly issue a save command upon every key stroke entered to a text editor. However, due to high latency of disk accesses as well as the large granularity of these accesses, this approach is untenable for a standard computer system.

Lastly, many of the more robust text editors, like Visual Studio Code [12], suffer from exorbitantly long start up times to initially open file for access [1]. This problem becomes exacerbated when opening large files, as substantial amounts of data must be read from the slow disk driver and brought into DRAM for use.

Fast, byte-addressable SCM devices like NVRAM seem like the ideal technology to address these issues. Prior work has been done to explore the benefits of consistent and durable data structures (CDDSs), and have seen a 74%-286% increase in throughput, dependent on the data structure in question [1]. Since Phase Change Memory (PCM) and other resistive SCMs show near DRAM access latencies [2], this result is expected, and so the issue of long start-up times will naturally be resolved by shifting the data structure to exist solely in the SCM device. In this paper, we make the following contributions:

- A brief evaluation that shows that volatile implementations of the data structures used by text editors are not tractable when attempting to resolve the aforementioned issues.
- Persistent implementations of the most commonly used data structures to build text editors.
- An evaluation that displays the promising feasibility of using these implementations to build text editors not susceptible to crash consistency errors.
- A brief exposition of the shortcomings of PMDK's C++ Transactions library for persistent programming.

The contents of the remainder of this paper are organized

as follows: Section II provides background information of the data structures used in text editors and their implementations; Section III outlines the implementation details of our persistent data structures; Section IV describes the workloads and our evaluations of the persistent data structures that we implemented; Section V presents the current state of research in persistent data structures; Section VI details the conclusion of the this work; and Section VII lists a few possible points of extension that this work leaves open for future exploration.

II. Background

A. Gap Buffer

In text editors, one can insert the text instantly at any position, based on position of the cursor. Gap buffers are one of the front runners when it comes to choosing a simple, yet efficient data structure for text editors. Gap buffers are not only easy to implement, but are also especially efficient for the majority of typical editing commands, which tend to be clustered in a small area. The contents of a file are is stored in a large buffer in two contiguous segments which are separated by a gap in between them. This gap is used for inserting new text. One can visualize the gap buffer to be a pair of buffers where one buffer holds all the content (or text) before the cursor, and the other buffer holds the content after the cursor. When the cursor is moved through the buffer, the characters are shifted from one buffer to the other. This is usually done by copying the text from one side of the gap to the other, where the copy operation can sometimes be delayed until the next operation that changes the text [19]. The insert and delete operations near the cursor position (where the gap is) are very efficient, as they resolve to simple pointer manipulations.

A typical implementation of a gap buffer is based on a single large buffer, with the pre-cursor content at the beginning, the post-cursor content at the end, and the gap spanning in the middle. This implementation incurs very little overhead and can be searched and displayed very quickly, compared to other, more sophisticated data structures. However, one thing that is noteworthy, is that operations at different locations in the text and the ones that fill the gap (requiring a gap to be re-created) may require copying most of the text, which is especially inefficient for large files. The use of gap buffers is based on the assumption that such recopying of text occurs very infrequently and the amortized cost of these operations is compensated by the more common (and cheaper) operations on the text.

For any gap buffer implementation to be used in building a text editor, the following fundamental functions are required at the minimum:

i `void insert(string input, int pos);`

As the name suggests, this operation will insert the character into the text at the specified location in the buffer. Whenever the call to `insert` is made, the gap will be positioned in such a way that the left gap point (or the pointer to the end of the pre-cursor content) is at the ”to be inserted” position. At every call to `insert`, the size of the gap is checked. If the gap is empty, a call to the `grow` operation is made, which resizes the gap and the new character is inserted.

ii `void moveCursor(int pos);`

This operation helps in positioning the cursor to a desired position. In the case of `insert`, this position would be the place where the new character needs to be inserted. In case of the `delete` operation, this position would be the place where the character needs to be deleted from.

iii `void grow(int k, int pos);`

This operation “grows” the gap from the specified position. This operation is necessary when the current gap size becomes zero after an insert fills the entirety of the gap, thus requiring the buffer to be resized. The value by which the gap is increased is implementation defined. While some implementations use a constant value to increase the size of the gap [7], other implementations may use the length of the input string. One such example is illustrated in Figure 1.

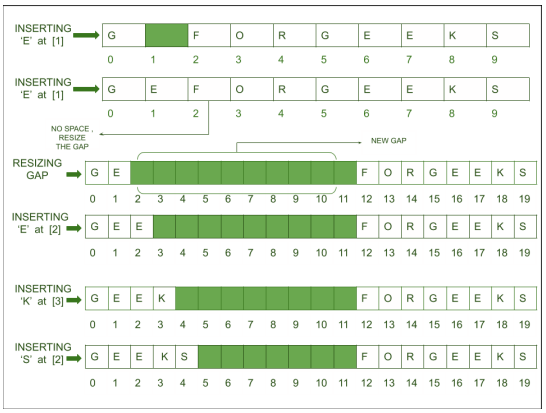

Fig. 1. Insert Triggering a Grow Operation

iv `void deleteCharacter(int pos);`

For deleting a character, three possible cases exist [6]: delete at cursor, delete at left, and delete at right.

a) **Characters to be deleted are at cursor position:** Since the cursor is already at the desired position, the character can be deleted immediately, i.e., the pointer to the start of the gap is decremented such that the character to be deleted will now reside within the gap itself, as shown below in Figure 2.

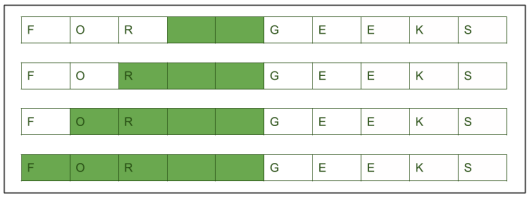

Fig. 2. Delete at Cursor

b) **Characters to be deleted are left of cursor position:** For this case, since the desired location is to the left of the cursor position, an additional step must be done to position the cursor to prepare for the delete. Once the the cursor has been moved leftwards by the appropriate amount of characters, the procedure is same as the first case. This operation can be seen below in Figure 3.

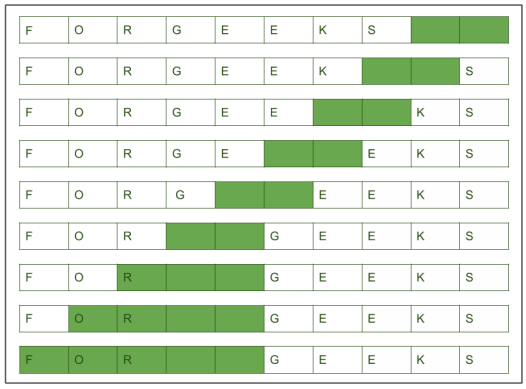

Fig. 3. Delete at Left

c) **Characters to be deleted are right of cursor position:** For this case, since the desired location is to the right of the cursor position, an additional step must be done to position the cursor to prepare for the delete. Once the the cursor has been moved rightwards by the appropriate amount of characters, the procedure is same as the first case. This operation can be seen below in Figure 4.

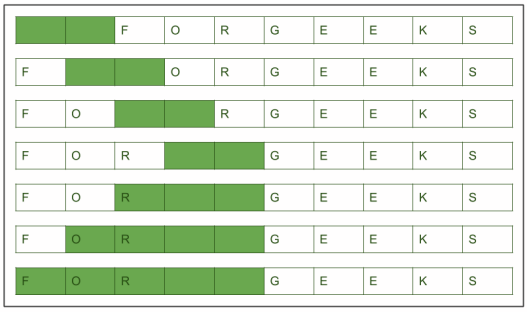

Fig. 4. Delete at Right

B. Piece Table

One problem with the gap buffer data structure, is the issue of reallocating a gap after insertions fill up the existing gap. In 1983, the Microsoft corporation developed a data structure called a Piece Table for their new text editor: Microsoft Word. The piece table is an innovative data structure that uses indirection to format even non-sequential writes into an append-only access pattern. Consider a text file that contains the text "The fox jumped over the dog." If we wanted to insert the string "entered the yard and" right after the word "fox". Traditional text editors have the issue of having to shift the text after the insert to provide space for the insertion. Let us now examine how this file would be loaded into a piece table when opened.

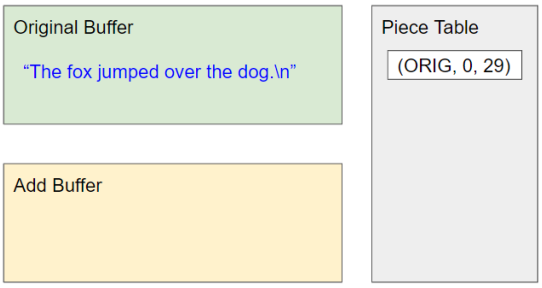

Fig. 5. Piece Table on Open

To maintain a piece table for a text file, three constructs must be created and managed. As shown in Figure 5, these three structures are the Original Buffer, the Add Buffer, and the Piece Table itself. The original buffer simply contains the entire contents of the file as a string when the file is initially opened. This buffer will remain constant until the piece table is coalesced upon a close operation. The add buffer will contain any additional strings that are meant to be inserted to the file. The piece table contains a ordered list of pieces. Each piece is identified by three pieces of information: which buffer the piece should be read from, what offset in this buffer should reading start at, and how many characters should be read from the buffer. In Figure 5, it can be seen that when a file is initially opened, there is a single piece entry in the piece table that tells the text editor to read the entire contents (all 29 characters) of the original buffer (ORIG designation) starting from the beginning (offset of 0).

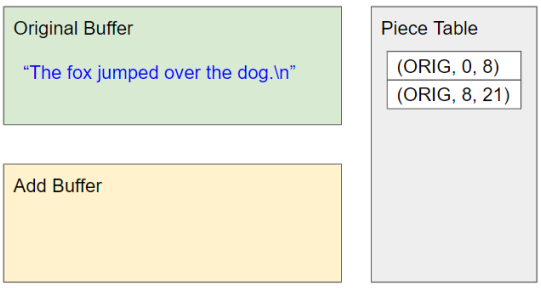

Fig. 6. Splitting a Piece in the Piece Table (Transient State)

To perform the string insertion mentioned beforehand, two discrete operations to the piece table must occur: the initial piece must be split where the insertion occurs, and a new piece must be allocated to represent the insertion. While these two steps are discrete, the transient state of the piece table after completed the first operation is never visible to the end user. I.e., the state shown in Figure 6 is merely an intermediary state. In this figure, it can be seen that the original piece has been split into two pieces where our desired insertion is supposed to occur. The first piece instructs the editor using the piece table that the first text that should be rendered to the user is the first 8 characters

from the original buffer. The next piece instructs the editor that remaining text to render should start being read from index 8 in the original buffer, and will read the remaining 21 characters.

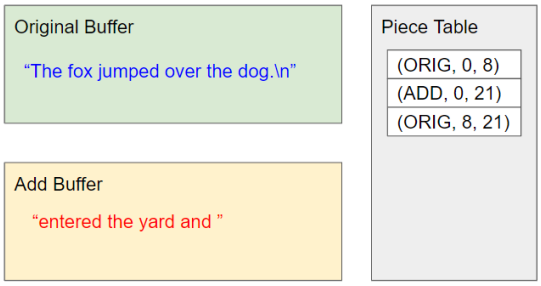

Fig. 7. Inserting to a Piece Table

In order to complete the insertion operation, the text to be inserted is added to the add buffer and a new piece is inserted into the piece table that references this text. This can be seen in Figure 7, as the second entry of the piece table references text from the add buffer (ADD designation). A text editor may render this piece table by simply reading characters in the order they are referenced by the pieces in the piece table. In this manner, performing large insertions resolve to an append-only access to the add buffer, and a small pointer switch in the piece table itself. Thus the latency overhead for managing the piece table is minimal, and reduces the any lag faced by an end user.

Thus, we define the following minimum functions that a piece table implementation must provide in order to support a text editor (in all the prototypes shown, `PT` refers to a struct type that encapsulates the piece table):

i `void open(PT *T, string source);`

This operation opens a source file in read only mode and copies the contents into the original buffer, stored in DRAM. As mentioned in the previous sections and in Figure 5, an initial piece is allocated in the piece table to indicate that before any modifications, all text should be rendered from the entirety of the original buffer.

ii `void insert(PT *T, string txt);`

This operation takes a given string and inserts it to the file at the current cursor position. Note that the cursor is a construct managed internally by the piece table. For a concrete example of how the insert operation works, please see the previous section and Figure 7.

iii `void remove(PT *T, size_t len);`

This operation modifies the entries of the piece table to hide `len` number of characters after the current cursor position. For all the efficiencies realized in the insert and open operations, the remove operation can be quite involved, and is perhaps the most problematic operation. Recall the post-insertion state shown in Figure 7. Suppose now that wish the text int the file to read as "The fox leapt over the dog." To do this, two operations need to occur:

1 Remove the unwanted text "entered the yard and jumped".
2 Insert the string "leapt".

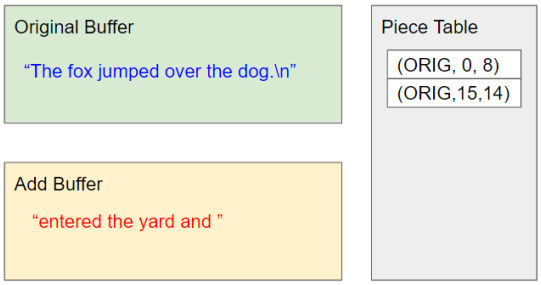

Fig. 8. Removing Text From a Piece Table

To perform the remove operation, we need only change the fields of the entries in the piece table. Figure 8 above shows the necessary changes to the pieces that must occur to enact the remove operation. Parsing the piece table, we see that an editor will render the text "The fox over the dog.\n" It is important to note that both the original and add buffers **remain unchanged**. Thus, the key pitfall of using a piece table becomes clear. When editing a text file, if the inserted text is large and then later removed, the runtime memory consumption of the editor **remains** unnecessarily large, as garbage text resides in the buffers. This is not a issue for files that are opened for a few changes (most common case), but can be quite problematic for files that are opened and have a large number of changes performed on them. Additionally, this poses a fragmentation issue as well. To complete the replacement operation, we must again insert the added text to the add buffer and allocate a piece to reference this. This is shown in Figure 9 below.

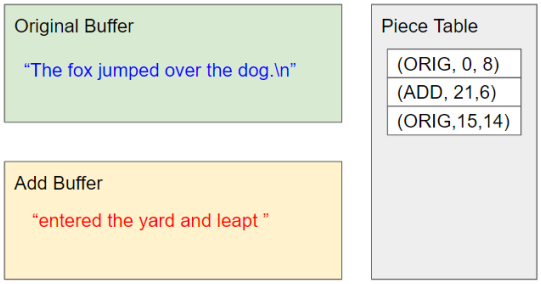

Fig. 9. Replacing Text in a Piece Table

Here we can see that for an editor to render this text completely, it must index into the original and add buffers at irregular indices. For a piece table stored in DRAM, should the buffers ever get paged out (may happen if no edits are made for a long time), these non-sequential accesses may incur large latencies to page them back in from disk.

iv `void close(PT *T, string dest);`

The final file operation that a piece table must provide is closing the file. With other file system APIs, this

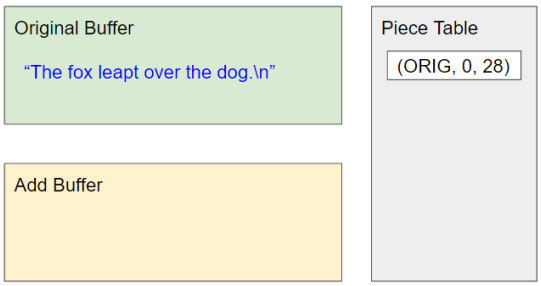

Fig. 10. Closing a Piece Table

operation is sometimes optional, however, for any file system that uses a piece table this operation is mandatory. This is because this operation performs the necessary function of **stitching** all the pieces back together. Recall that on opening a file, the piece table consists of a single piece and all the renderable text resides in the original buffer. A close operation must restore the piece table to this format, and then explicitly write out the newly created original buffer to a SCM. Thus the resulting state of the piece table is shown in Figure 10. On files with many edits, this operation may be quite costly. However, unlike the remove operation, this action may only occur on a file **once**.

v **Various Cursor Operations**
As mentioned before, any implementation of a piece table that is used by a text editor must have some internal representation of the cursor and manage it implicitly in it's fundamental operations. For our volatile implementation of the piece table, we expose the following cursor operations: `seek`, `rewind`, and `get_cur_pos`. These are merely the piece table counterparts for the commonly known file operations: `fseek`, `rewind`, and `fgetpos`, respectively. However, the management of the cursor in these functions and in the fundamental file operations mentioned above do not add latency during the use of a text editor. As such, we omit the explicit details of managing the cursor.

C. Ropes

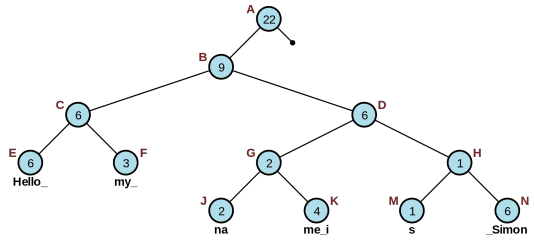

Fig. 11. Rope built on the string of "Hello_my_name_is_Simon" [20]

A final data structure that is commonly used in text editors (Sublime [17] and Vim [13]) is the Rope data structure. A rope is a unique instance of a binary tree that repeatedly partitions a long string of text so that fundamental operations like reading and writing are highly performant. The terminal (leaf) nodes in the tree contain pointers to the partitions of the initial string. Each node in the tree is weighted by sum of characters contained in a node's subtree. For a leaf node, it's weight is simply the number of characters in the partition it points to. For non-leaf nodes, their weights are simply the sum of all the weights in the **left** subtree.

While this design proves to be time-efficient, it is only so when the rope is kept balanced. Maintaining this property incurs re-balancing actions on **every** insert and remove operations. Note that, re-balancing a binary tree may need to change a pointer in every node in the tree, in the worst case. Implementing this in NVRAM is impractical as the latency faced by a user on every keystroke would precipitate lag times that would make a text editor built on this data structure unusable. For this reason, we omit an evaluation of a persistent or volatile version of a rope data structure.

III. Implementation

A. Persistent Memory Development Kit (PMDK)

Libraries such as PMDK by Intel [5], Atlas by HP [2], and others have been developed in the recent years for programming NVRAM. We used Intel's PMDK in our implementation of persistent data structures for the following reasons:

- *Extensive Documentation:* There are elaborate document/manual pages for the functionalities that we implemented. As the PMDK library is more mature compared to the other options, we had correctly expected it to support more functionalities out-of-the-box. Thus, programming the persistent data structure proceeded without many issues.
- *Active Development and Active Community:* As is evident from [5], the library is under active development and has a relatively active community. This aspect was beneficial, as we believed it would be easy to get assistance when troubleshooting unforeseen issues. This was an important consideration, as persistent programming is notorious for being difficult due to it being a relatively new research area without many subject matter experts (SMEs).
- *Synergy with Underlying Hardware:* As we planned to run our experiments on Intel's Optane memory; choosing the PMDK library (by Intel) was a natural choice. The PMDK library has been designed to integrate seamlessly with a machine with Optane.
- Other APIs, like Atlas [2], are not as optimized for performance as PMDK, requiring explicit locking in user code, and have expensive logging designs.
- Approaches like Recipe [10] cannot be used here since they only apply to limited number of indexes, where reads can tolerate inconsistent states.

B. Gap Buffer

For the implementation of the persistent version of Gap buffer, we used PMDK's libpmemobj++ library. As the volatile version used C++'s Standard Template Library

(STL), it was a logical step to use its equivalent version for the persistent memory version as well. We made use of constructs like persistent memory pointers and containers to port the corresponding primitives and STL containers that were present in the volatile version of the program.

We note that the `pmem::obj::persistent_ptr` is similar in concept and implementation to the smart pointers introduced in C++11, with one big difference – the object's life cycle is not managed. Another reason to use persistent containers is that the STL containers use algorithms that are non-optimal, from a persistent memory programming point of view. Persistent memory containers must have the properties of durability and crash consistency, while not every STL method guarantees strong exception safety.

For the different operations that were discussed in the Section II.A, we used the transaction API for the persistent memory version of the data structure. Using the transaction API provides atomicity, as well as crash consistency for the operations that we perform on the gap buffer.

C. Piece Table

Transactions provided by Intel's PMDK library guarantee consistency and hence are used to implement the following methods of the piece table index:

1) Create
2) Insert
3) Delete
4) Stitch

The members of the structures used in the implementation of the volatile piece table index were made persistent. The newly inserted blocks of text were read into a persistent string, which was then used to initialize an empty piece. The newly created piece was then pushed into a persistent vector of pieces, all within in a single transaction. To recall from Section II.B, in the insert method of our volatile implementation of the piece table data structure, the newly added string was inserted at the appropriate position in the "add" buffer of the piece table. The PMDK library has a peculiar behavior for transactions that require large amounts of data to be buffered, as in the case of the "insert" method. For such transactions, a `transaction_out_of_memory` error was generated. This appears to be due to the limited logging space in PMDK, however, documentation regarding this is limited and the exact cause of the error remains to be ascertained. In order to overcome this limitation, the "add" buffer was pre-allocated with 10^4 characters upon initially creating a piece table. We have ensured that these buffered characters do not affect other operations such as reads or write-outs to a file.

D. Verification with Pmemcheck

We have used `pmemcheck`, a Valgrind tool developed by Intel to check for memory-related bugs in our implementations of these two data structures. We did not find any bugs using `pmemcheck` for a variety of tests covering all the methods used in our implementation.

IV. Evaluation

This section evaluates the persistent gap buffer and piece table data structures, and compares them with their volatile memory counterparts. For the scope of this project, we limit ourselves to the measurements of the latency and memory usage when using these data structures.

A. Experimental Setup

Experiments were run on a server with Intel Xeon Gold 5317 at 3.00 GHz having 12 cores 24 threads and 18 MB cache, and 94GB Intel Optane NVRAM and DRAM. The server ran Ubuntu 20.04 with kernel 5.4.0. All the code was compiled using g++ 9.3.0.

B. Benchmarks

To the best of our knowledge, there aren't any standard benchmarks that we could use for the analysis of our data structures. As a result, we created two benchmarks that we believe should be able to fairly evaluate the performance of our persistent data structures. To emulate the behaviour of a regular user of text editors, we insert characters and words in a sequential manner, measuring latency for every single operation. We took into consideration the fastest typing speed of a user (120 words per second) and aimed for having delays (due to higher latency for writes) which are not noticeable by the humans. This value lies in the magnitude of milliseconds [4]. Our goal was to ensure our data structures met this criteria and minimized the DRAM memory usage. We measured the time taken by the `insert`, and `delete` operations at a character-level granularity for `n` characters to measure performance.

C. Performance of Volatile Data Structure

We began with the experiments of volatile data structure to set up a baseline and understand the scalability of gap buffers and piece tables. To make volatile data structure and persistent version comparable, we saved the data structure into a file after completing the insertion operation.

Insert Latency and Memory Usage: To evaluate the performance of insert latency and memory usage, we insert different number of characters, one at a time, in each iteration. These results are in figure 12.

	Insert Time (msec)		Heap Usage (Bytes)	
Num of characters	PT Volatile	GB Volatile	PT Volatile	GB Volatile
16	0.022511	0.028064	92,699	73,801
32	0.032363	0.052793	93,628	74,043
64	0.05181	0.103582	95,485	74,449
128	0.0914	0.189865	99,198	75,019
256	0.18342	0.389994	106,623	78,001

Fig. 12. Insert latency and memory usage of volatile data structure

Impact of Saving Frequency. To make the volatile data structures comparable with their persistent counterparts, we issued save commands to write the data structures into a file immediately after finishing the insertion operations. We tested the operation latency with various saving frequencies,

from saving the data structure on every ten character insertions (0.1) to saving on every character insertion (1.0). While it can be seen that these saving-after-inserting operations increase linearly with the saving frequency, the frequency has an upper bound of 1.0 (i.e., cannot save faster than on every character). Thus, the asymptotic complexity of these operations is **not** important. The key insight here is that reducing the frequency yields performance at the cost of making the data structure more prone to crash consistency bugs. Additionally, we see that even in the worst, our data structures only require a mere 50 milliseconds to write a character to the disk. These results can be observed in Figure 13, shown below.

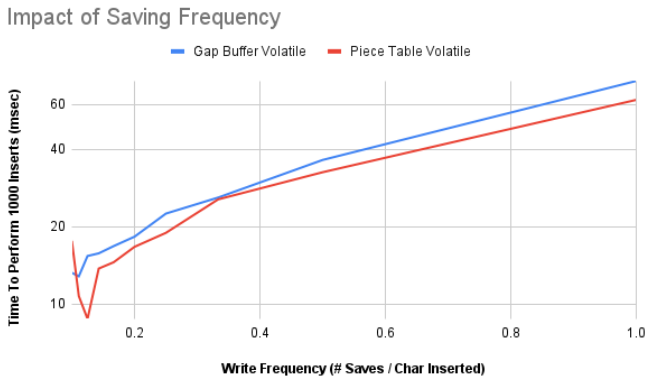

Fig. 13. Impact of Saving Frequency

D. Performance of Persistent Data Structures

Insert latency and memory usage. We tested sequential insert operations with a various number of iterations. We recorded the latency of these operations, as well as the heap usage with the help of Valgrind. The results are in figure 14. Although our persistent versions are three magnitudes slower than the volatile versions and take more than 200x memory usage, these latencies are not noticeable by the users, and the memory overhead is acceptable.

	Insert Time (msec)		Heap Usage (Bytes)	
Num of characters	PT Persistent	GB Persistent	PT Persistent	GB Persistent
16	123.697	36.2917	23,307,493	22,241,299
32	257.906	87.8567	23,602,798	22,529,059
64	485.482	110.286	24,187,630	23,104,579
128	740.439	324.291	25,357,358	24,242,259
256	1091.91	498.528	27,693,102	26,513,763

Fig. 14. Insert latency and memory usage of persistent data structure

Impact of Insert Size: For each insert operation, we sweep the input string size from 1 to 50 and record the insert latency. We observe that the insert latency for volatile gap buffer increases monotonically as the size of the input string that is being inserted goes up, whereas the insert latency for the piece table remains static. This matches our expectations, as the piece table is better suited to this type of append-only access pattern. However, we see that as we reduce the size of the inserted text, the gap buffer structure performs better. From this figure, we can infer that gap buffer would be a better choice of data structure than piece table for a (NVM based) text editor that intends to persist the data at every keystroke. However, a NVM-based text editor that intends to insert text into the data structure for every word, where a word typically consists of 6 or more characters, will benefit more from using a piece table than from a gap buffer. These results can be seen in Figure 15, shown below.

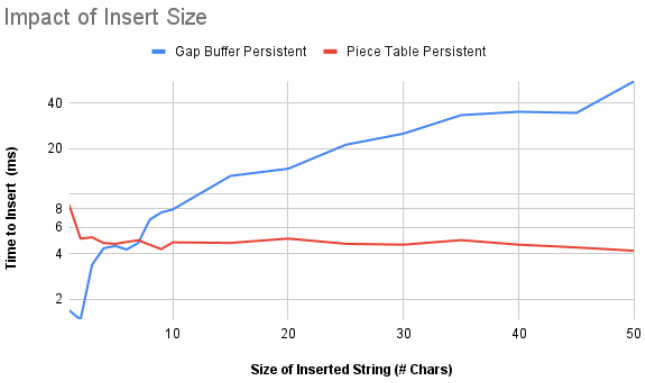

Fig. 15. Impact of Insert Size

E. Evaluation of Macro-Benchmark

We evaluate the data structures using a simulation of real-life usage of a text editor. We consider the situation where `n` units of text are inserted into the data structures on an `insert` operation and the changes are persisted by a `save` operation. We consider two units of text here, a word, comprising of 6 characters, and a character. For the volatile data structures, the save operation is performed by explicitly writing out the changed text contents to disk after each unit of text is inserted into the data structure. Traditional disk based text editors have achieved persistence of writes by explicitly issuing write-out commands at fixed frequencies, usually after a certain number of characters have been written. By considering two different units of text, a character and a word, we intend to account for two of the different frequencies at which these write-outs to disk might happen. In contrast, we need only perform a single write-out operation to the file (that the user would have initially opened) with the persistent data structures, as we leverage the inherent persistence of writes when using PMDK. We use this operation of inserting `n` units of text where each insert is followed by a save for volatile data structures and insertion of `n` units of text followed by a single save for persistent data structures as a macro-benchmark for evaluating the two data structures.

1) Volatile Data Structures: Figures 16 and 17 show the average time taken by the insert and save operations for the volatile gap buffer and piece table respectively. As we can observe, the average save time is higher than the average insert time for both units of text used and for both the data structures.

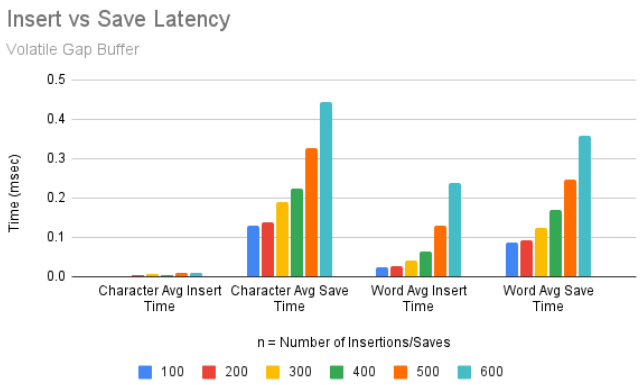

Fig. 16. Avg insert and save times by volatile gap buffer

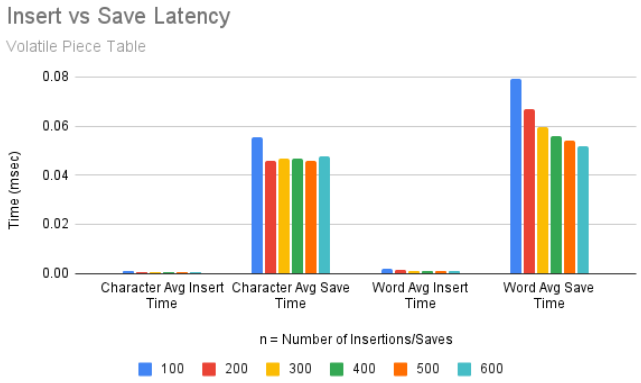

Fig. 17. Avg insert and save times taken by volatile piece table for n units of text

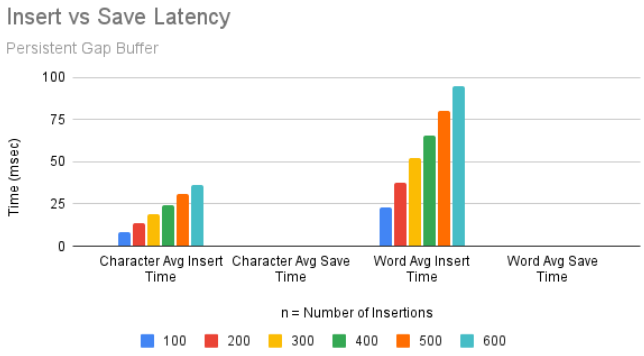

Fig. 18. Avg insert and save times taken by gap buffer for n units of text

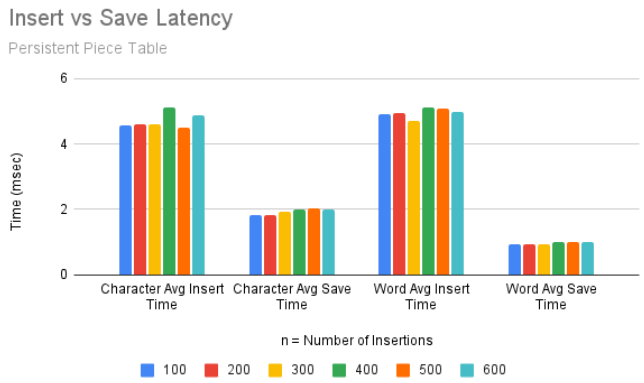

Fig. 19. Avg insert and save times taken by persistent piece table for n units of text

2) Persistent data structures: Figures 18 and 19 show the average time taken by the insert and save operations for the persistent gap buffer and piece table respectively. We can see that the average save time, in this case for a single save operation, is lower than the average insert time for both units of text used and for both the data structures. The insert operation is therefore more expensive than a write-out to file with the persistent data structures, hence the insert operation is the bottleneck here.

Further, we notice that the persistent piece table which inserts units of text in discrete pieces performs better for the insert operation than the gap buffer, which needs to shift existing text contents in order to insert a new unit of text. The insert operation for each unit of text with a gap buffer is still less than 100 ms and considering the typical human typing speed, this latency cannot be perceived by a human user. Using a piece table is however advantageous for insert heavy workloads since the insert operation has a nearly constant time of less than 6 ms even with an increasing number of insertions. However, the save operation is expensive for a piece table since the various pieces need to be stitched together before writing out the final text to an output file. With a gap buffer, save operation just involves writing the final text present in the buffer to the output file and hence the gap buffer is an ideal choice for workloads that write-out frequently to a file.

V. Related Work

CDDS [18] presents an efficient versioning-based data structure design scheme that allows for the creation of log-less systems on non-volatile memory, without any processor modifications. The authors propose a single-level Non Volatile Byte-addressable Memory hierarchy where no distinction is made between a volatile and a persistent copy of data. The CDDS design relied on background Garbage Collection for memory management and hence, suffers from numerous dead entries and dead nodes. As noted in [3], to consistently manage the NVM space and avoid memory leak, used data blocks are tracked using the persistent reference pointers (a.k.a., root pointers); CDDS ignores this issue, with the risk of causing a memory leak.

Some works like [8] and Recipe [10] have provided in-depth evaluations to determine formulaic methods for converting volatile data structures to their persistent implementations, and identify what design goals are addressed by different implementations. Similarly, [16] provides specific code snippets for implementing a hash table that is robust against crashes with a discussion of Copy-on-Write vs Versioning. While these studies provide useful insights

on how to go about designing persistent data structures, their evaluations do not look at the irregular data structures that are commonly used in text editors (Ropes, Gap Buffers, and Piece Tables).

A good summary of the work done, till date, on data structures for persistent memory is provided by [8]. The authors also provide a set of design principles for creating data structures for PM. A combination of PM and DRAM is recommended to obtain best performance while achieving persistence since dereferencing and pointer chasing on PM can be expensive. An example of this approach is presented in [9] by the authors in section "Hash Table with Transactions and Selective Persistence". An optimal size in the range of 256 B to 1 KB is suggested for nodes of data structures in PM which is a helpful guideline. Sequential access is faster in PM and DRAM based data structures will be preferred for applications involving frequent jumps between non-sequential cache lines.

Recipe [10] provides a principled approach to making some data structures persistent. The ArXiv version of their approach has updated some of the conditions to account for the data structures that did not always yield correct persistent algorithms even after meeting the prerequisites of the approach. These new formal conditions require flush and fence instructions for every read and write. While some of these instructions can be left out, the decision for doing the same is left to the user. Due to its highly specific approach for conversion of a specific class of DRAM indexes, Recipe could not be directly used to convert the volatile memory data structures used in the text editors.

WORT [9] is a single-threaded, hand-crafted write-optimal variant of Adaptive Radix Tree for persistent memory. It introduces a new recovery mechanism and modifies the node structure to be failure-atomic. Unlike the B+-tree variants, radix tree neither requires key sorting nor rebalancing of the tree structure. Also, since the structure of the radix tree is deterministic, inserting or deleting a key requires only a few 8 byte write operations. However, radix trees are sensitive to the key distribution and often suffer from poor cache utilization due to their deterministic tree structures. Moreover, radix trees are not as versatile as B+-trees as their range query performance is very poor.

FlatStore [3] is an efficient PM-based key-value storage engine. It decouples the role of a KV store into a persistent log structure for efficient storage and a volatile index for fast indexing. It introduces compact OpLog, lazy-persist allocator, and pipelined horizontal batching to achieve high throughput, low latency, and multi-core scalability. Specifically, FlatStore only stores index metadata and small KVs in the persistent log, uses a multi-class based allocation policy and persists the allocation metadata lazily during the runtime.

Dash [11] is a holistic approach to building dynamic and scalable hash tables on PM. It introduces two dynamic hashing schemes, extensible hashing and linear hashing. Dash combines techniques including fingerprinting to avoid unnecessary PM reads during record probing, optimistic locking to avoid PM writes for search operations, and bucket load balancing to postpone segment splits with improved space utilization.

To the best of our knowledge, this is the first work that is being done for conversion of the volatile data structures used in the text editors to their persistent versions. Due to very little overlap in the data structures that have been considered in the previous work, we could not re-use the already existing techniques for conversion to the persistent versions. However, we took inspiration from the common underlying ideology of working on top of the existing volatile data structures instead of "reinventing the wheel", metaphorically speaking. We believe this is a fair approach as the volatile versions of these data structures have been developed over a number of years, they are highly optimized for their respective use-cases.

VI. Conclusion

Given the impending shift to Non-Volatile Random Access Memory (NVRAM), this work presents persistent data structures for text editors. We set out to build a text editor using NVM that offers crash consistency, is fast enough for humans that the operational delay is imperceptible, and has a low memory footprint. We believe that the first two of our three major goals have been fulfilled in our work. We have been able to create data structures that are crash consistent and persist the data to the non-volatile memory within a reasonable time. Since the average user types at about 40 words per minute, with an average word length of 6 characters; we can reasonably assume that text editors need to render characters within a magnitude of milliseconds for the lag to not be noticeable. While text editors whose underlying data structures reside entirely in DRAM easily meet this requirement, they hinder usability due to the potential inconsistency issues on crashes. Therefore, we conclude that the persistent data structures that have been proposed in this work are a way to go forward especially for better consistency guarantees.

VII. Future Work

The work put forth in this paper is novel to the realm of text editors, and as such, there are many possible points of extension. One such avenue would be to explore the tradeoffs of shifting some of the workload to the recovery operation. Our implementation, as described in Section III, uses Intel's PMDK Transaction library to implement the fundamental file operations described in Section II for the various data structures. Since each operation was implemented as a single transaction, crash consistency was guaranteed implicitly, as every operation moves the data structures from one consistent state to another. However, as shown in the evaluation section, the C++ implementation of these transactions are incredibly expensive in their memory footprint. Something that is left to be explored is if there is a way to modify the volatile implementations of the Gap Buffer and Piece Table such that they can tolerate inconsistent states on accesses, and resolve these errors on recovery. In this way, tools like Recipe [10]

can be employed to more quickly derive persistent versions of these indexes.

Another point of extension from this work would be to re-evaluate what data structures should be used for text editors built upon fast, byte-addressable NVRAM devices. The data structures presented in Section II, gap buffers, piece tables and ropes, were all conceived based on the underlying assumption that memory accesses to the SCM device is prohibitive slow and operating at block-level granularity. Considering NVRAM for use in text editors begs the question whether these data structures are still relevant for future implementations. Our results show that simply converting these data structures to reside in NVRAM is promising. However, more benefits may be exposed by developing a custom data structure.

As a final consideration, this work does not consider the increase in write traffic on the memory bus. One of the biggest drawbacks of NVRAM devices like Intel's Optane, is that they have a relatively short lifespan, capable of only enduring $10^6 - 10^8$ writes [14]. This work proposes solutions that write to persistent memory on every keystroke to minimize the recovery issues on a crash. We predict that this will incur an extreme increase in write traffic, and will leave the SCM device susceptible to fast decay. It remains to be explored how much of an impact persisting every keystroke will have on the durability of the device.

VIII. Acknowledgement

The authors would like to especially thank Professor Michael Swift from the University of Wisconsin-Madison for his invaluable guidance and insight on this project.

References

[1] Casper Beyer. Why i still use vim, 2017. [Online; accessed 15-October-2021].

[2] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 433—-452, 2014.

[3] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[4] J. Deber, R. Jota, C. Forlines, and D. Wigdor. How much faster is fast enough? user perception of latency & latency improvements in direct and indirect touch. In *Proceedings of the 33rd annual acm conference on human factors in computing systems*, pages 1827–1836, 2015.

[5] Developers of the PMDK library. Pmdk: Persistent memory development kit, 2021. [Online; accessed 16-December-2021].

[6] Geeks For Geeks. Gap buffer — deletion operation, 2021. [Online; accessed 16-December-2021].

[7] Geeks For Geeks. Gap buffer data structure, 2021. [Online; accessed 16-December-2021].

[8] P. Götze, A. K. Tharanatha, and K.-U. Sattler. Data structure primitives on persistent memory: an evaluation. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3, 2020.

[9] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. {WORT}: Write optimal radix tree for persistent memory storage systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 257–270, 2017.

[10] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019.

[11] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020.

[12] Microsoft. Visual studio code: Code editing. redefined., 2021. [Online; accessed 15-October-2021].

[13] B. Moolenaar. Vim, 2021.

[14] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[15] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 24—-33, 2009.

[16] S. Scargall. Designing data structures for persistent memory. In *Programming Persistent Memory*, pages 187–206. Springer, 2020.

[17] Sublime HQ Pty Ltd. Sublime text: Text editing done right, 2021. [Online; accessed 15-October-2021].

[18] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[19] Wikipedia contributors. Gap buffer — Wikipedia, the free encyclopedia, 2021. [Online; accessed 16-December-2021].

[20] Wikipedia contributors. Rope (data structure) — Wikipedia, the free encyclopedia, 2021. [Online; accessed 15-October-2021].

[21] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, pages 169—-182, 2020.