INVESTIGATION OF VIRTUAL MEMORY AWARE DATA STRUCTURES

An Undergraduate Research Scholars Thesis

by

CONNOR NICHOLLS

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Roger Pearce

May 2024

Major: Computer Science

Copyright © 2024. Connor Nicholls.

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Connor Nicholls, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Investigation of Virtual Memory Aware Data Structures

Connor Nicholls
Department of Computer Science & Engineering
Texas A&M University

Faculty Research Advisor: Dr. Roger Pearce
Department of Computer Science & Engineering
Texas A&M University

Virtual Memory Aware toolkit is a library of data structures with specific purposes, allowing for significant speedups to be found through the elimination of assumptions. This toolkit is designed to be flexible within its specific purposes, and allows for complete customizability of parameters for each project. While the library will be continuously updated as more structures are added, the primary focus is that of the bag and the stable index set. Bags are containers that do not guarantee sequence ordering, and as such are enabled to provide faster removals in cases where the order does not matter. Stable index sets are linear probed hash tables that perform resize operations by allocating levels of ever-increasing user-defined size to add new information to, allowing for quicker resize functions and stable access of members via a returned handle. This is supported by a fingerprinting algorithm, by which a selection of the hash is kept alongside each inserted object in order to allow for more efficient duplicate checking. While these algorithmic improvements allow for stronger use cases, they are not solely responsible for this.

Following the theme of elimination of assumptions, instead of using *malloc* for memory allocation, VMA toolkit utilizes *mmap*, handling the memory allocation directly. By skipping over *malloc*, which contains logic to allow for generalized use cases, VMA toolkit is designed for high performance and large data set use cases, meaning that the memory allocation can be optimized

for the specific use cases and create quicker allocation times throughout. Notably, handling allocation directly also means that the program can direct the kernel on how to handle the pages it has received, once again allowing for significant speedups compared to the standard implementation.

Our experiments indicate that our VMA implementation of *bag* performs faster in all use cases than bag, demonstrating its utility for general purpose storage of data. Our data also shows that our stable hash table performs much faster than its current alternative, the naive index set, and performs comparably to, although not faster than, the standard hash table in all but specific delete cases, where the stable traits of the structure allows for gains over the standard hash table.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Roger Pearce, for his guidance and support throughout the course of this research.

Thanks also go to my colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Thanks also to John & Hank Green, Tommy Smeltzer, Troy Nakamura, Jim Crossley, Clay Wispell, and all others who, throughout my life, drew out the passion and creativity in me, and helped me explore it in many different ways.

Finally, thanks to my friends for their encouragement in good times and comfort in rough times, and to my parents for their patience and love through everything.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work was accomplished without any funding required.

NOMENCLATURE

STD	Standard
VMA	Virtual Memory Aware
XM	X Million
XB	X Billion

1. INTRODUCTION

High-performance computing is one of the fields pushing innovation within computer science, with scientists looking to push the limits of programs and eek out every bit of performance possible at every turn. Much time has been spent on optimizing the performance of hardware components, with each integrated circuit being pushed to the limits of size and computing power. But as roadblocks build up, and it grows increasingly harder to get more performance physically, the research community must turn to improvements that can be made to the very systems that are utilized on that hardware. Across the enormous quantities of data being processed at the highest levels, even a slight speedup can have large consequences on what is able to be accomplished. The Virtual Memory Aware toolkit looks to accomplish one step in this direction, speeding up the resident data structures commonly used for storage and access of this data.

Data is one of the most fundamental building blocks of computer science, as at its core, the purpose of a computer is to execute a set of instructions on data, and as such, the ease of access to the data lends itself to more efficient systems. VMA finds speedups in many of the fundamental areas essential to data structures as they were innovated in the early ages of computing and operating systems. Between virtual memory manipulation, cache alignment, restructuring of structure properties, and narrow scopes of intended use, VMA libraries are able to gain a significant speedup over their competitors, creating the building blocks for faster data access and faster computing. The goal of this paper is to not only provide the tools for faster computing, but also to encourage the development of more similar and better tools to push the top class of computing hardware to its theoretical limits.

Pushing the limits of computing speed is important for the advancement of the human race. From gene sequencing to trace our very DNA, to perusing the depths of space at an incredible level of detail, to searching the body for solutions to cancer, to improved processing abilities for artificial intelligence and virtual reality, computing has a presence in many of the important issues and technologies of today. Currently, many resources are being dedicated to solving these problems, and considering the enormous scope of these projects, slight speedups could result in potentially days, weeks, even months worth of time saved in processing, not even counting the savings in electricity costs and computing center maintenance and rent.

2. BACKGROUND

This body of research is built upon the shoulders of those who came before. The intention of this section is to provide an understanding of the underlying theories at play within the thesis and the history that led to this point.

2.1 A Brief History on Allocators and Their Functionality

Before the genesis of UNIX, there was Multics, a time-sharing system developed by Bell Labs employees and eventual creators of UNIX Ken Thompson and Dennis Ritchie [1]. Key to the concept of Multics was their approach to handling data, namely that the system treated all data as if it was already written into memory. If it was found to be not by the user, the system would then load in the data at the time of interaction. This, however, was deemed to be not optimal by Thompson, as this system treats all data as if it were the same, which is not practical. Frustrated with the winding down of the project by Bell Labs, Thompson began writing his own version of Multics, with distinctions made for instruction sets and data, replacing the former single-store system [2]. This system, through much trial and error, would eventually become the UNIX operating system that set the foundation for modern computing today. Importantly, the proliferation of this system and the language B (and eventually C) through unofficial means resulted in a large open-source community collaborating on the system.

One of these collaborators was Doug Lea, an early inventor of the general-purpose allocation system, fittingly titled *DLMalloc* [3]. This allocator became the default allocator for Linux, and even in modern times with much develop is considered the base-level stable allocator by which to compare to [4]. Within this, and other similar allocators such as *PTMalloc* and more modern implementations such as *SuperMalloc*, the key is that data is allocated in chunks, with each chunk carrying a certain bit of information about the strategies used to implement it as well as the data [5][4]. While the strategies to handle these chunks differ by allocator, the concept spans all of the above, namely that they primarily look to expand currently existing memory regions with *sbrk*, which increases the boundary limit of the address space, and only directly allocate chunks with *mmap* when necessary, which tended to be when chunks surpassed a certain threshold and could not be directly expanded with *sbrk*.

2.2 On Virtual Memory

Tom Kilburn, one of the influential researchers in the development of the Atlas computer in the early 1960s, was a key component of the development of virtual memory [6]. Up until this point, programs had to be written with physical memory in mind, with developers manually controlling which sections of memory would be used and which would be expanded if necessary. This resulted in programs being system-specific, making portability of development nearly impossible. The solution the team developed was referred to as the "one-level store", a system by which the developer would have the "the illusion that he has a very large main memory at his disposal, even though the computer actually has a relatively small main memory" [7]. When the operating system was passed an address, the operating system would use a lookup table referred to as the page table to determine which place in physical memory corresponded with the given address, and if it did not exist, the operating system would handle the loading and processing of that data. By letting the operating system handle this, the system could not only be portable, but also incredibly more efficient, as the operating system would always know exactly how much memory it could utilize at any time and optimize its operations behind the scenes to ensure full efficiency.

Given this format for memory, a general overview for allocation can be seen at a system level. When a chunk as defined in Section 2.1 is to be allocated by *mmap*, the system learns of how large the chunk needs to be and designates a space in the virtual page table to be reserved for the chunk. Then, the operating system finds a suitable and preferably contiguous spot within the memory and claims it for the chunk, leaving an entry in the page table denoting where to find this piece of memory. Now that this exists, the operating system is free to move around the piece of physical memory as needed, while leaving the reference to it in the page table intact for access by a user.

One such common example of doing this is the *madvise* command, which is utilized to give the kernel advice on what strategies to use with different spans of memory [8]. This can be used to tell the kernel how often something might be accessed, who the data belongs to, and more, and the kernel can then manipulate the data to be utilized most optimally.

2.3 A Brief History on Hashing Functions and Hash Tables

To "hash" a piece of data is to convert a block of data into an identifying number for that piece of data. This concept was birthed by Hans Peter Luhn in 1958, at an international conference where he introduced the KWIC, or Key Word In Context algorithm [9]. This system would create an index of articles of varying length by calculating the "hash" of the words in an article and create an index of the processed data. This was a particular advantage in terms of finding data, as to find something, one only had to plug what they were finding into the hash function, which would return an identifying number of where to find it. In modern times, hash algorithms are everywhere. One common location of this technology that is often unrecognized is in the checksum function, which calculates a hash of a selection of data. This can be utilized to verify the integrity of data, or to check to see if the data has been changed. This is due to the fact that the number generated by a hash function can change greatly with a tiny amount of influence to the piece of data. For example, replacing one letter with another in a sample document could completely change every part of the number. As such, hash functions are incredibly useful in many daily tasks. One of the primary places they find use, however, is within hash tables.

Hash tables take the concept of "hashing" data, and apply it to data storage. One of the constant issues with data storage is finding the data that has been stored. For example, if one were to look for a piece of data starting at the beginning of a list, there is a non-zero chance that they will have to traverse the entire length of the list to find the object, which can take an incredible amount of time as data sizes increases. Hash tables solve this issue by taking the computed hash of the data to be inserted and using the modulo function to bound the resultant hash to the size of the table, such any inserted piece of data will have a place to be put somewhere within the table. This, however, poses two issues.

Firstly, what happens when the table begins to reach capacity? The current solution for hash tables is to create a larger table and reinsert the values from the old table into the new one, redoing the modulo function in order to properly bound the hash to the new length of the table to ensure an even distribution of data across the table. This can be seen in Figure 1.

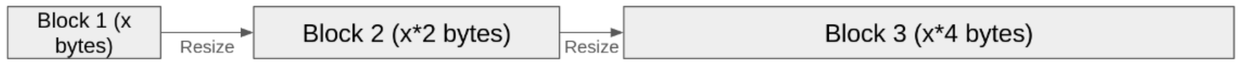

Figure 1: Standard Hash Table Resize Functionality. Dynamically reallocates blocks of memory and re-enters the data into the new blocks as a linear time operation

Secondly, what happens if two values were to be placed in the same spot? Since the space of the table is finite, hash values may be bounded to the same value, causing what is known as a hash collision [10]. There are many different methods of dealing with this conflict, but there are two primary methodologies, chaining and linear probing. For a chained collision resolution, each table is considered to be a table of linked lists. Whenever a value is inserted into the table, it is appended to the linked list at its proper location. As such, whenever a value is inserted into the table in a place that already contains a value, it is appended to the list, which can be traversed quickly in the case of needing to find the data stored there. The primary issue with chained hash resolution is that it takes up a large amount of space, as linked lists require at minimum a pointer to the next object and the overhead that comes with initializing a new entry in it, which can add up in the memory over time. The second and more commonly used variation is linear probing. When a value would be inserted into a spot that is full within the table, the program will then look at the adjacent spot in the table to see if it is full. If it is empty, the value will be inserted there, and if it is not empty, the process will continue until it finds an empty slot. This way, when the user is looking for data, they can find the initial spot that they calculated the value should be at, and probe along the table sequentially. They will keep probing until they either find their element, or reach an empty spot, which would indicate that the object was never inserted at all. This method requires no extra space constraints, and minimal time loss, with the main issue being that the probe lengths can grow rather large as more data is inserted.

However, there is one issue with this model of linear probing: the case of deletion. Take for example that inserted in a single probe distance is the values 3, 13, and 23, sequentially stored next to each other. Suppose that a user were to delete the value 13, and then look to find the value 23. They would first probe 3, and then move to the spot which used to contain 13, which is now empty. Seeing an empty spot, they would return, assuming that the value could not be found, which is incorrect. To resolve this, we introduce the concept of a tombstone array. A tombstone array is a parallel array of the same length as the value array that contains one of three information values: full, empty, or tombstone. Full and empty values work exactly the same as listed above. However, if a value is deleted, its status in the tombstone array is changed from full to tombstone, indicating that if there were to be a probe into this position, the probing should continue past this spot, despite the lack of a value filling the slot. This adds some additional space complexity, but due to the small size required to store these 3 values, it is considerably negligible, and results in no time loss compared to other mechanisms.

Notably, due to the incredibly quick times to find and delete data, hash tables are one of the most popular data structures in use in computing. As such, much research has gone into the development of increasingly more optimal hash functions, looking to spread out data as evenly as possible to reduce the amount of expensive resize operations that are inevitably required during use in a hash table. In addition, it has been determined that after a certain percent of the hash table is full, there are diminishing returns in finding data, as the probe distance grows too long relative to the size of the table. Current research suggests forcing the table to resize once it has reached somewhere between 62-75% capacity [11].

2.4 Discussion

One other important data structure that requires much less instruction is the vector, which can be thought of as an sequential, self-growing array. Elements can be added to the end and popped from the end in constant time, which results in an easy-to-use data structures that permeates many areas of computer science. Importantly for general use, if not necessarily this paper, is that vectors also implement the standard library concept of iterators, which allow for the abstraction of access to the elements and access to the powerful and fast algorithm library. Iterators do pose a significant syntactical burden on data structures, however, and for the intended purposes of this study they do not pose a significant enough advantage to be considered for addition or consideration at this time.

The above sections primarily discuss UNIX-based operating systems, which should clarify why this study is intended for use with UNIX systems specifically. While there are certainly similarities within Windows and MacOS, both system handle many things differently than UNIX, especially with regards to kernel advice and allocation. Consideration may be given to adapting the project to other operating systems eventually, but as most high-performance computing currently occurs on UNIX-derived systems, it is deemed unnecessary at this time.

An important factor to remember about computer memory is that the memory is broken up into pages that are logged in the page table, and that these page sizes vary from system to system. For most x86 based systems, it is typically true that a page is 4096 bytes. For the purposes of this program, when requesting data, it is done by incrementally requesting full pages until the requirement is met, as when requesting pages of memory, the full page will be given to the program even if it only requests a portion of it, leaving the rest mostly unused.

3. IMPLEMENTATION

Memory in programming is a complex topic, and as processors and graphics units grow faster and faster, so too does the need for efficient memory access. The concept of the "memory wall" is that gains in processing speed are consistently greater than gains in memory bandwidth [12]. It is due to this that optimizations in code take place in not only reducing instruction count, but also reducing memory latency and bandwidth. One key place where programs can be found lacking in efficiency lies within the standard allocation function *malloc*. As a generalization, allocating memory is the process of requesting a chunk of page-aligned memory from the kernel, which takes the form of a set of empty pages committed to the virtual page table and returned to the program. Due to the generic nature of *malloc*, there is a great amount of logic that goes into the allocation process, which involves deciding how much memory should be allocated, whether it should request an entirely new set of memory or simply extend the border of the heap, and more, which for simple operations, can take some extra, unnecessary time. The Virtual Memory Architecture toolkit looks to reduce this unnecessary time by skipping over the *malloc* logic entirely and request the pages of memory directly from the kernel. By solely requesting unfilled memory pages, the included data structures will, on large datasets, perform better than their counterparts.

In addition, a big time save throughout the toolkit involves the utilization of virtual memory paging in relation to physical memory locality. One important facet of the virtual memory system described in Section 2.2 is that “although physical memory is always precious, virtual address space on a 64-bit machine is relatively cheap” [5]. What this means is that virtual memory can be allocated rather freely, as long as the allocated pages are not committed to physical memory until strictly necessary. VMA structures are created with the intention of allocating large page chunks quickly in the virtual memory, and lazily mapping them to the physical memory whenever necessary. This applies to deletion as well, where the program will not directly reclaim the memory, and instead direct the kernel to reclaim the physical memory at its leisure. These small speedups result in large time saves across many operations throughout all applicable libraries.

3.1 Selected Data Structures

3.1.1 Bag

Firstly, a bag is effectively a vector that lacks the sequenced property, meaning that it cannot be considered to be ordered in any way. In a standard vector, when an object is deleted, the program first finds the object, deconstructs it, and then shifts all proceeding elements down to fill in the gaps left behind. In a worst case scenario, such as deleting off the head of a vector, this could result in having to perform a move operation on every element within the vector, which is costly. The bag solves this issue by not guaranteeing order, which results in the movement step being able to be skipped, leading to faster times in storage cases that do not care about sequence.

3.1.2 Stable Hash Table Set

An index set is just a hash table that solely stores keys, and not any associated data. The standard hash table implementation discussed in Section 2.3 discusses the expensive operation of resizing and how that impacts the utility of the table. In the VMA structure, instead of creating a new array and rehashing every element when the maximum capacity is reached, the program instead remaps the memory, keeping the current mapping, but increasing the size by a variable factor, as seen in Algorithm 1.

Algorithm 1 Stable Hash Table Resize

```
$set = mremap(set$ from $capacity$ to $capacity + (capacity\_at\_max\_level * level\_scaling\_factor))$
$capacity = capacity + (capacity\_at\_max\_level * level\_scaling\_factor)$
$num\_levels++$
```

The previously allocated memory is now considered “Level 1” of the structure, and the newly allocated section is now considered “Level 2”, as seen in Figure 2. Now, the program can continue to work within this second level, functioning off of the same hash value modulated by a different value to bound the hash to the size of the new level, and insert into it, without having to touch any of the mappings in the “first level”.

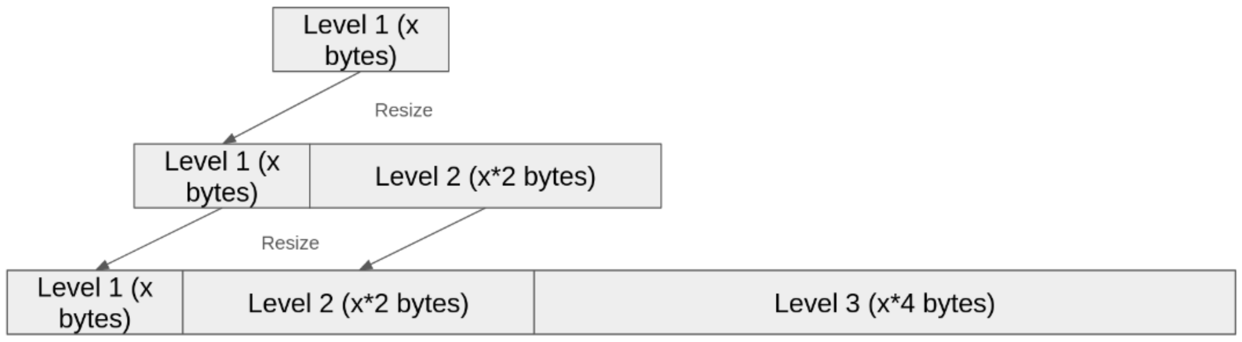

Figure 2: Stable Hash Table Resize. Keeps old memory block and allocates new, bigger block as a constant time operation.

Notably, this means that, unlike a standard hash table, the address to a particular element will always be constant once it is inserted, meaning that a handle of the slot can be returned to the program for easy and efficient access later, a feature which can be seen in Algorithm 2. This has a variety of use cases. Primarily, this is intended to support the implementation of a dataframe-like data structure in C++. A dataframe is essentially a map where one can dynamically add values to the key. The implementation of a stable hash table is essential for this structure to be formed, and has many use cases across data analysis, as seen in the Python pandas library [13]. In addition to this, stable hash tables are essential in building graphs where there exist more nodes than edges, specifically in hyper-sparse matrices [14]. The existence of the stable hash table provides a way to translate from a system of open address naming of vertices into a real, mathematical matrix. This type of matrix has many uses, such as mapping all possible strings, UUIDs, or all possible md5 hashes, that have widespread utility [15].

Algorithm 2 Stable Hash Table Insert

```
$hsh = hash(val)$
$loc = hsh\%level\_capacity$
**for** $current\_level = 0, \ldots, max\_level$ **do**
    **for** $probe = 0, \ldots, probe\_distance$ **do**
        **if** loc + probe is empty **then**
            mark slot for insertion if no location found yet
            $break$
        **else if** loc + probe is a tombstone **then**
            mark slot for insertion if no location found yet
        **else if** loc + probe is full **then**
            **if** $slot\_value = val$ **then**
                mark that duplicate has been found
                $break$
        **if** $probe = probe\_distance$ AND $current\_level = max\_level$ **then**
            $resize$
    **if** slot not marked for insertion **then**
        $loc = (hsh\&(level\_capacity - 1)) + level\_offset$
**if** slot marked for insertion **then**
    $insert$
    $return$ handle for value
```

3.2 Fingerprints

This section will be dedicated to the explanation of a more complicated facet of the VMA index set, the fingerprint. Recall the hash table implementation discussed in Section 2.3. Consider the tombstone array mentioned in this implementation, the parallel array utilized to track how to properly probe the table. This array, however, can perform another use case. Consider that the aforementioned hash function returns a 64-bit integer. Only a certain subset of the integer is being used at a time, while the rest go unused, and for the case of this example consider that the last 8 bits are being used for the hash. Theoretically, then, it would be possible to take a certain subset of that unique hash from somewhere else (for example, the first 8 bits), which will henceforth be called the fingerprint, and store it within the tombstone array. As such, a 0 would indicate empty, a 1 would indicate tombstone, and any other number would indicate a full space occupied by a value with hash as generated above. This way, when searching through the array, instead of having to perform equality checks on the objects, which can be expensive, the operation can instead be performed on the fingerprint first. If the fingerprints are the same, then that slot is possibly the same value, and as such would have to be thoroughly checked, however in the fingerprints are not the same, it is not possible for the value to be the same object, resulting in a significant speedup when comparing objects along the probe path.

Given all of this, as levels grow deeper, the hash may grow stale if the same values continue to be utilized, resulting in clustering at higher levels. To prevent this, the hash is cycled at every level by a variable amount of bits. Effectively, this means that the last bit(s) of the full hash are removed from the end and placed at the front, shifting the hash by one bit to the right. This results in a different distribution between levels, resulting in better residency rate throughout the structure.

3.3 Testing Methodology

There are several categories to evaluate these structures on. For the stable hash table, a load factor test will be performed by varying the level scaling and probe distance to determine the optimal combination on both a set of random data and a set of half duplicate data, as describe below. This will be evaluated both on the time to completion of the process, and the amount of pages committed to physical memory, as measured by the *mincore* functionality. Then, there will also be a comparison test between the stable hash table, the standard hash table, the naive index set, and the unordered set.

Something that has not been previously discussed is the *naive index set*. An index set is a map of values where a unique index is assigned to every value. The requirements of the naive index set, however, is to able to be traversed bidirectionally, meaning one can move from the set of indexes to the set of values and vice versa. The naive index set is the primary way of accomplishing that in C++ currently, which involves a map of keys and values, and a vector of iterators to values within the map. The vector of iterators is necessary due to the map not being a stable structure, so in order to access the elements in the map by an index (through the vector), there must be an iterator associated with each element. This is incredibly costly, however, due to the massive overhead associated with it for a considerably inefficient structure. The stable hash table should solve these issues by allowing for a hash-based structure that allows for key to value mapping that also remains stable, meaning that any given index associated with a value will always refer to that value slot.

For each data structure, a series of tests will be used to compare their standard forms to their VMA toolkit implementations. For each structure, several datasets will be ran to determine performance under differing environments. These testing suites are:

- Sorted Insertion
- Random Insertion
- Half Duplicate Insertion (insert half the amount, then reinsert the same values again)
- Sliding Window Delete

For all of the above datasets, assume that all random data will be seeded for repeatability utilizing the C++ *rand* library. Also, it is notable that, with a good hash function such as the currently utilized MurmurHash64, clustering should be minimized no matter what data is included, so the sorted dataset should have little to no difference compared to the random dataset [16].

For the dataset itself, data sizes 65536, 1048576, 16777216, and 67108864 will be utilized as page-aligned quantities of 64-bit integers to insert. Outside of the comparison test, statistics will be analyzed with the size 67108864 dataset.

The memory usage will be calculated with the *mincore* system call, which determines how many pages in a given range have been committed to memory. Recalling the discussion in Section 2.2, pages allocated with mmap are not physically represented until modified by the program. This test determines the efficiency of the memory being used, as the comparison between implementations will tell which is using more memory to store the same amount of elements. The goal of the stable hash table is to take a large amount of virtual memory and commit as little of it to physical memory as possible. With the current approach, it should be expected that outer levels will not be able to remain fully uncommitted, as outer levels by definition should be mostly sparse, however it should be possible depending on data and hashing mechanisms to achieve a similar memory complexity to the standard hash table.

For the sliding window delete, the intended use case is when a user is receiving a great amount of data, but only wants to keep the freshest data of a certain amount. This amount of data to keep is referred to as the window size, and after the data begins to exceed this size, the stalest data is deleted, so there will only ever be at most the window size amount of data in the structure. This use case is intended to evaluate the effectiveness of the delete functionality, as well as the ability of both the standard and stable structures to traverse across the tombstones left behind by the deletions. Notably, this process is ignoring the possibility of rehashing after reaching a certain density of tombstones to purely evaluate the ability of the structure to effectively evaluate these states.

4. RESULTS

4.1 Bag

4.1.1 Expectations

For the bag data structure, the time complexity remains the same across all functionalities, other than the delete function. In a standard vector, a delete operation is a worst-case of linear complexity, requiring every element to be moved up, as discussed in Section 3.1. For the bag, however, the delete operation is constant time even in worst case, due to not having to move any of the data. Notably, this is not guaranteed to be an efficiency save on any single operation, as a standard delete can be as fast as constant time as well. In addition, the virtual memory implementation of bag should also result in some time saved from mapping less pages to physical memory, as discussed in Section 2.2. As such, it is expected that bag should run significantly faster in delete operations from random places within the structure, and slightly faster in all other operations.

4.1.2 *Empirical Data*

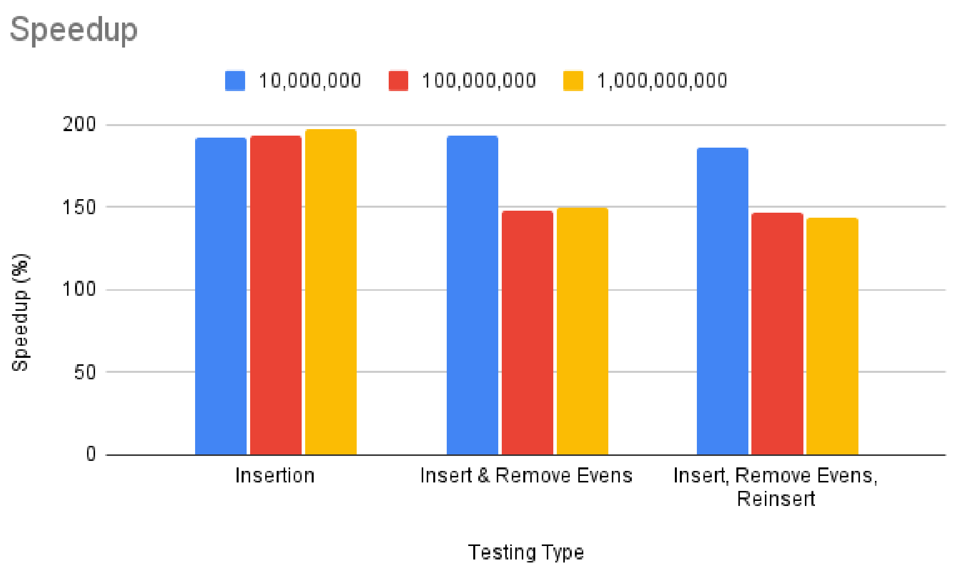

Figure 3: Observed speedup of the bag data structure relative to the standard vector. Each test case demonstrates that the VMA bag performs up to twice as fast and at least 40% faster than the STD vector.

Across the data, there is roughly a 150-200% speedup across all operations, with larger data sizes having greater speedups. This indicates that the virtual memory paging system creates larger efficiency gains across larger data, which corresponds with the design methodology of the system. For the delete operations, time gains were not as significant as indicated in Section 4.1.1, likely due to the random nature of the delete function meaning that not many of the delete operations being performed are worst case. The efficiency gains throughout the data indicate that, for the purposes of storing data more efficiently, the bag will perform faster and more efficiently than a vector.

4.2 Stable Hash Table

4.2.1 *Time Complexity*

When looking at the time complexity of the aforementioned structures, it is clear that, complexity-wise, there is not a large difference between unordered set and the standard hash table, with both having a constant time insert, find, and delete operation, and a linear resize operation. With the stable hash table, however, the operations are reversed, with a worst-case insert, find, and delete of logarithmic time (as discussed in 2), counterbalanced with a constant time resize (as discussed in 1). In theory, given a sufficiently large quantity of data, it is possible for the stable hash table to perform faster, given that the standard hash table is forced to commit to several extremely costly resizes. However, for most reasonable cases, this will not be the case. It is expected that there will be time loss in experimental data, with the understanding that as long as it is not significantly slower, the advantages gained outside of time in stable handles will create a niche of utility for the structure.

4.2.2 Load Testing: Unique Insertions

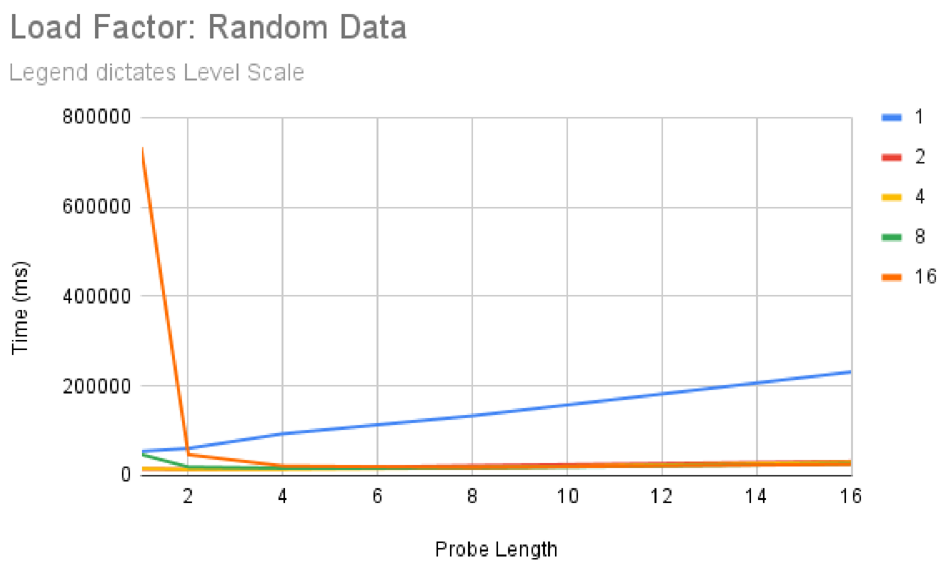

Figure 4: Observed effects of varying probe length and level scaling factor on insertion time of 67M elements without duplicates. Gains are for the most part minimal as values increase, with a level scale of 4 and probe length of 2 performing roughly optimally.

When testing the load of level scaling versus the probe distance, the empirical data indicates that a factor of four for level scaling and a factor of two for probe distance is ideal for unique insertions, as seen in 4. This balances the space requirements of the level resizing and the requirement of the probe to travel a distance equal to the amount of levels. Notably, a factor of two for level scaling and a factor of two for probe distance is also close to optimal, and due to structure, I expect it to be mostly optimal as sizes increase as well.

4.2.3 Load Testing: Duplicate Insertions

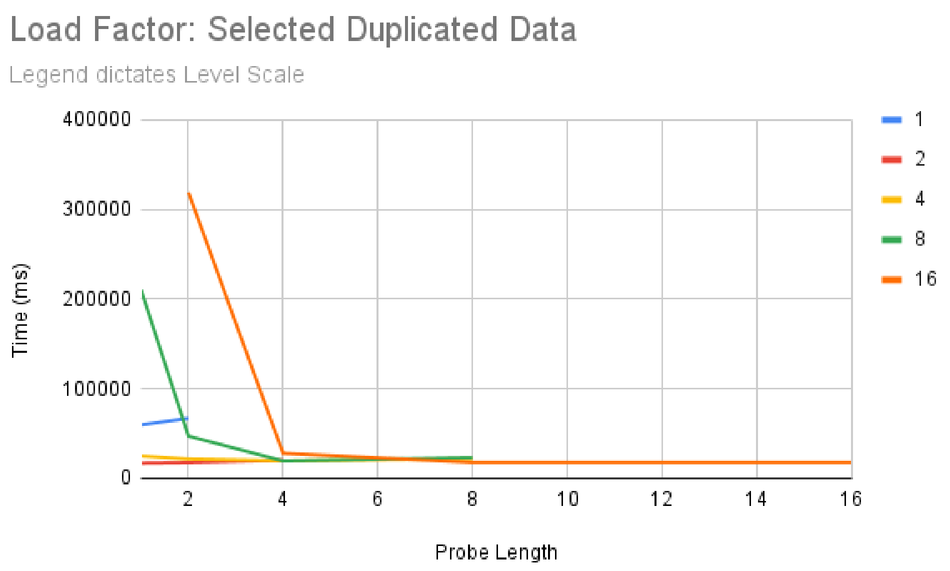

Figure 5: Observed effects of varying probe length and level scaling factor on insertion time of 67M elements with duplicates. Performance was more variable at lower level scales and probes, but level scales of two and four were deemed to be most efficient with a length of two or four.

When testing the load of level scaling versus the probe distance with a half set of duplicates, the data indicates that there are much less disparities between groupings of level scales and probe distances, with the optimal times being found at a scale of 16 and a probe of 8, with very close times found in every scale from 2 onwards, as seen in Figure 5. Notably, the time for the duplicate insertions is worse than the unique insertions as well. This indicates that the issue with duplicate insertion lies not in the size of the levels, but instead solely on the probing distance, and specifically in the probing that occurs to check for duplicates. This indicates that a good next step could be attempting to add speedups to duplicate detection to speed up this purpose.

4.2.4 Pages Resident in Core

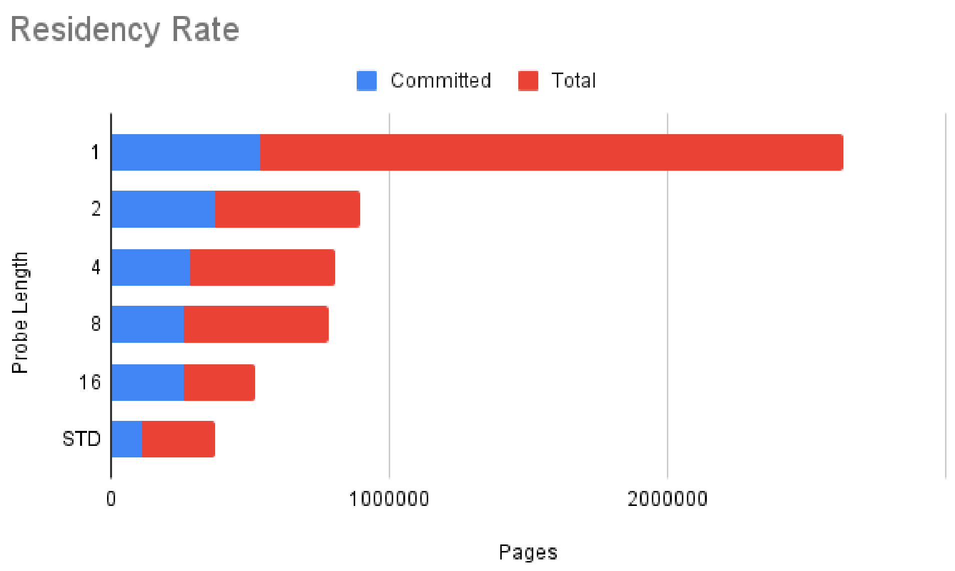

Figure 6: Observed effects of varying probe length on the total pages committed to physical and virtual memory for the stable hash table compared to the STD hash table. The STD hash table is more memory efficient in all cases.

Empirically, probe length is the most important factor to determine the amount of pages committed to memory, as seen in Figure 6. As probe length decreases, the amount of pages committed to memory increases drastically compared to the standard hash table, likely due to the distribution of data from the hash function. Because a new level will be created whenever the maximum probe distance is reached, smaller probe lengths mean more levels with more pages total and committed. This is relatively inefficient compared to the standard implementation, which requests a much smaller space to distribute data between and is generally much more compact. Further research into this topic could involve finding a way to distribute data such that sparse levels will also densely fill pages, instead of sparsely committing them after a resize, which would allow for smaller probe lengths to be more feasible.

4.2.5 Time Comparison: Stable vs Standard vs Unordered Set vs Naive Set

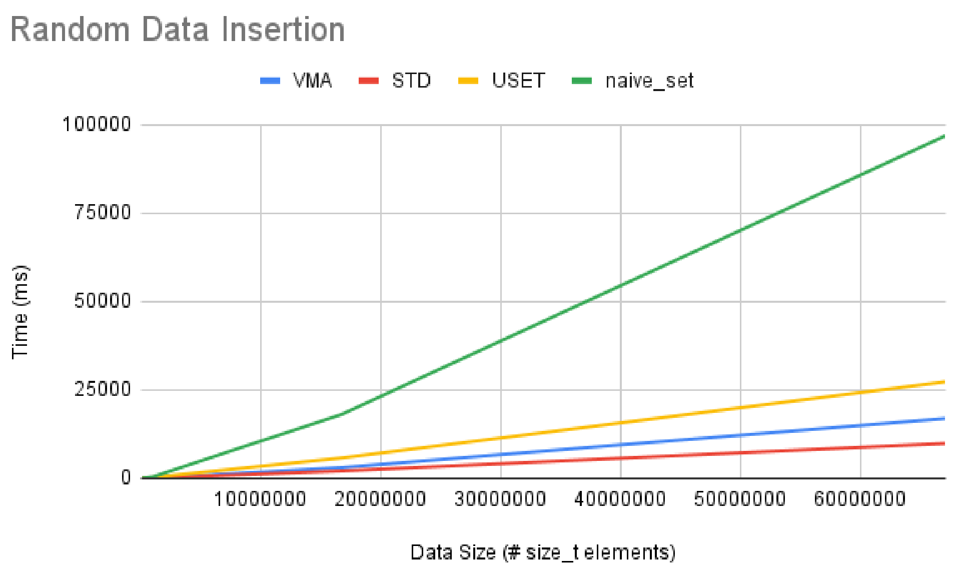

Figure 7: Struct random insertion time comparison varying data size (VMA struct utilizing level scale of 4 and probe length of 2). STD performed fastest, followed closely by VMA, and then unordered set. Naive index set performed the worst by a large margin.

When looking at the insertion of varying data sizes into the aformentioned structures, it becomes clear that, while the VMA hash table outperforms the unordered set in all cases, it is beat out by the standard hash table implementation, most significantly at larger data sizes, as seen in Figure 7. Within these parameters, unordered set performs worse than both VMA and STD structures, and the naive set performs magnitudes worse than all of the aformentioned structs.

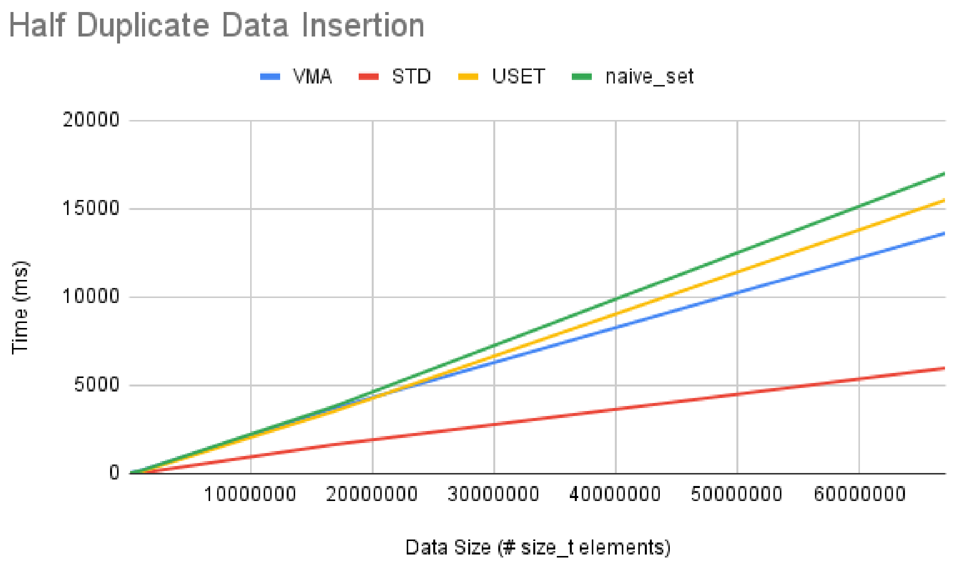

Figure 8: Struct half duplicate insertion time comparison varying data size (VMA struct utilizing level scale of 4 and probe length of 2). STD performed the fastest by a large margin, followed by VMA, unordered set, and naive index set, in that order.

This trend overall holds when duplicates are introduced, with the gap between the standard struct and the rest widening, as seen in Figure 8. This indicates that overall, the constant time resizes do not pay out for the probe length required for the stable hash table to insert at higher level counts to the point of where it performs more time-efficient than the standard implementation. Notably, however, as data sizes grow larger and larger, the stable struct times grow at a slower rate than the standard struct rates, meaning that as the data size grows, the efficiency of the stable struct grows relative to the standard struct.

When looking at the naive set compared to the stable hash table, however, it is clear that the stable hash table outperforms the naive set in all cases, especially in data sets with a lower count of duplicates. This comparison the key contention of this testing suite, as the uniqueness of the stable hash table is only comparable to the index set. Considering that the stable hash table is more time-efficient than the naive index set, it is possible for further expansion on top of the stable hash table in the algorithms described in Section 3.1.2.

4.2.6 Sliding Window Delete

The sliding window data indicates that there is a large spike in time usage after reaching a sliding window of 100,000 elements which then performs roughly constant as the data size increases, as seen in Figure 9. Within this sample, the span between 10,000 and 100,000 is the first and only time the STD struct is forced to perform a resize operation, as discussed in Section 1.

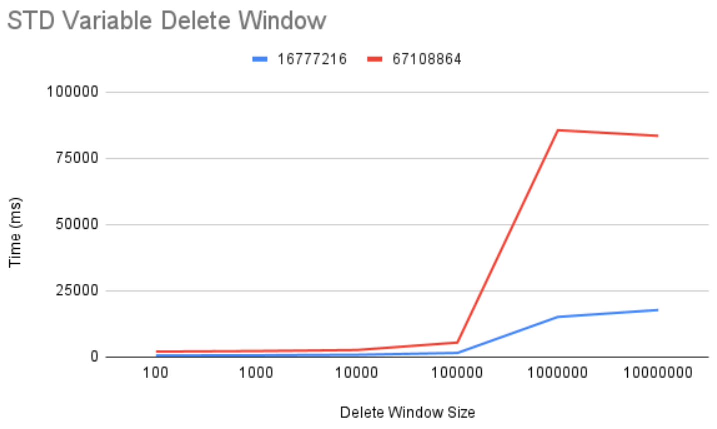

Figure 9: STD hash table sliding delete window time comparison varying window size across large data sizes. STD performs more efficiently when the data window is small enough to not require a resize.

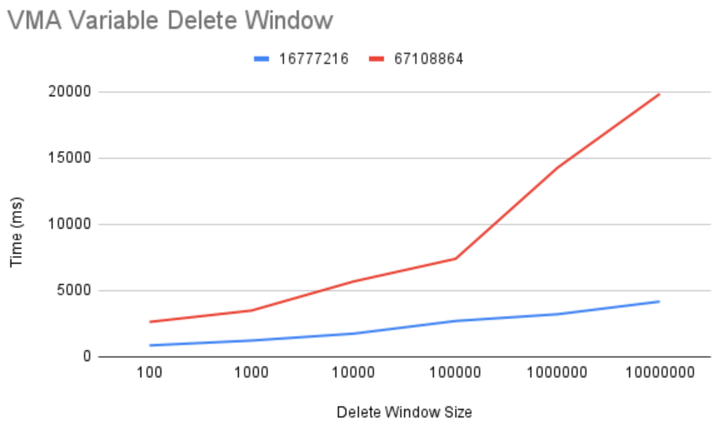

Figure 10: VMA hash table sliding delete window time comparison varying window size across large data sizes. VMA performs more efficiently when the data window is large enough to require a resize.

Meanwhile, within Figure 10, it is clear that the same increase occurs around 100,000, where the first resize takes place as well. However, it causes a much less significant time jump and increases linearly, enough to where the VMA struct performs better at all data sizes that require at least one resize.

This can be attributed to two factors: tombstone traversals and resize calculations. Due to the capped maximum probe distance on finding and deleting from the static hash table, tombstones cannot substantially increase the probe distance when looking to insert or delete. However, with the standard struct, this probe distance can grow greatly as tombstones become more dense, resulting in more real time spent probing than its stable counterpoint. In addition, that spike specifically corresponds with the first resize of the standard struct, where all the data must be re-entered into the array, resulting in not only a large immediate time loss, but also extra time loss in the future due to having even longer probe distances with greater masses of tombstones once a density has been reached again. This also explains why the data for 1,000,000 and 10,000,000 window values are very close together, as when the structure is resized, the tombstones are removed, resulting in much less increase while strings of tombstones are less present.

5. CONCLUSION

In conclusion, the VMA toolkit offers robust solutions for high-performance computing and allow for new use cases for previously common data structures. The increase in speed of the bag allows for it to be a more effective method of data storage that does not require direct management of memory by the user, increasing safety and performance metrics. Meanwhile, the stable hash table might not perform faster on average use cases than the standard hash table, but instead enables truly constant access to any element by its handle, a feature only accomplished due to the lack of redistribution of elements associated with the stable hash table, as long as the system is able to afford the increased virtual memory space requirements. This also appears to be moreso applicable to data sizes larger than the test environment was able to provide.

Further steps to improve upon this research would be to acquire testing on much larger data sizes to determine if the trends seen in the data hold in the billions and trillions of data, and see if the stable struct does at some point perform faster than the standard struct. Another improvement that would be made would be to implement a map over the indexed set hash table utilized, as the constant handle would prove to be very useful for a map with a key and a value. Finally, I would look to find a way to optimize the distribution of data within levels of the stable hash table, as sparse outer levels contribute to many unnecessary committed pages.

The toolkit can be accessed at `https://github.com/Fairo20/vma` under the MIT license.

REFERENCES

[1] W. Toomey, "The Strange Birth and Long Life of Unix," *IEEE Spectrum*, nov 28 2011. [Online; accessed 2024-02-25].

[2] D. Cooke, "Unix and Beyond: An Interview with Ken Thompson: Computer: Vol 32, No 5." https://dl.acm.org/doi/abs/10.1109/MC.1999.762801, may 1 1999. [Online; accessed 2024-02-25].

[3] D. Lea, "A Memory Allocator." https://gee.cs.oswego.edu/dl/html/malloc.html, apr 4 2000. [Online; accessed 2024-02-25].

[4] "malloc." https://pubs.opengroup.org/onlinepubs/9699919799/functions/malloc.html. [Online; accessed 2024-02-25].

[5] B. C. Kuszmaul, "Supermalloc: a super fast multithreaded malloc for 64-bit machines," *ACM SIGPLAN Notices*, vol. 50, pp. 41–55, jun 14 2015. [Online; accessed 2024-01-19].

[6] P. J. Denning, "Virtual Memory: History." [Online; accessed 2024-02-25].

[7] P. J. Denning, "Virtual Memory." [Online; accessed 2024-02-25].

[8] "madvise(2)." https://man7.org/linux/man-pages/man2/madvise.2.html. [Online; accessed 2024-01-18].

[9] H. Stevens, "Hans Peter Luhn and the Birth of the Hashing Algorithm," *IEEE Spectrum*, jan 30 2018. [Online; accessed 2024-02-25].

[10] P. Nimbe, "An Efficient Strategy for Collision Resolution in Hash Tables," *International Journal of Computer Applications*, vol. 99, no. 10. [Online; accessed 2024-02-25].

[11] O. Owolabi, "Empirical studies of some hashing functions," *Information and Software Technology*, vol. 45, pp. 109–112, feb 1 2003. [Online; accessed 2024-04-04].

[12] S. A. McKee, "Reflections on the memory wall." https://dl.acm.org/doi/abs/10.1145/977091.977115.

[13] “pandas.DataFrame.” https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html. [Online; accessed 2024-03-28].

[14] A. Buluc and J. R. Gilbert, “On the representation and multiplication of hypersparse matrices,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–11, 2008.

[15] *RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace*.

[16] Contributors to Wikimedia projects, “Murmurhash.” https://en.wikipedia.org/wiki/MurmurHash, apr 3 2024. [Online; accessed 2024-04-07].

[17] P. Celis, P.-A. Larson, and J. I. Munro, “Robin hood hashing,” in *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, IEEE, 1985. [Online; accessed 2024-03-27].

APPENDIX: DATA COLLECTION

Table A.1: VMA Bag Speedup vs STD Bag

Data Count	Insertion	Insert & Remove Evens	Insert, Remove Evens, Reinsert
10,000,000	192.3076923	192.8571429	186.3636364
100,000,000	193.0599369	147.8365385	146.8181818
1,000,000,000	197.4741236	149.8273669	143.5991869

Table A.2: Static Hash Table Insertion

	65536	**1048576**	**16777216**	**67108864**
Sorted	11	83	2492	13573
Random	10	82	3032	16955
Half Duplicates	38	173	3673	13614

Table A.3: Standard Hash Table Insertion

	65536	**1048576**	**16777216**	**67108864**
Sorted	7	71	2081	9196
Random	7	71	2087	9861
Half Duplicates	7	32	1629	5973

Table A.4: Unordered Set Insertion

	65536	**1048576**	**16777216**	**67108864**
Sorted	13	244	5893	27075
Random	7	279	5756	27319
Half Duplicates	2	113	3497	15490

Table A.5: Naive Index Set Insertion

	65536	**1048576**	**16777216**	**67108864**
Sorted	11	251	5201	23202
Random	13	442	18118	96843
Half Duplicates	7	172	3773	17000

Table A.6: Load Factor: Random

	1	**2**	**4**	**8**	**16**
1	54020	14629	15727	47256	734407
2	60700	14295	13703	19044	46543
4	93324	16357	13912	16633	21464
8	133326	22059	17556	16852	18005
16	231611	31755	30020	25939	25200

Table A.7: Load Factor: Selected Half Duplicates

	1	**2**	**4**	**8**	**16**
1	59805	16937	24827	210500	
2	67075	17562	21507	47164	319097
4		20132	19480	19358	27898
8			21754	23058	17727
16					17703

Table A.8: Page Residency

Probe Length	**Committed**	**Total**
1	538055	2095104
2	372991	522240
4	282705	522240
8	261295	522240
16	260096	260096
Standard	112285	262144