Linear Hashing for Flash Memory on Resource-Constrained Microprocessors

by

Spencer Donald James MacBeth

Supervisor: Dr. Ramon Lawrence

The Irving K. Barber School of Arts and Sciences

(Honours in Computer Science Major, Minor in Data Science)

The University of British Columbia – Okanagan Campus

April 2017

Abstract – Hardware widely adopted in recent times has stimulated the need for algorithms attuned to a wide array of physical environments. One set of environments which poses several implementation challenges are microprocessors with constrained resources such as the Arduino. Many implementations of data structures cannot perform as desired without significant modifications in the Arduino environment due to the low levels of RAM typically available, such as the 2KB of RAM available in the ATmega328. In addition, these devices typically use a microSD card or local erasable programmable read-only memory (EEPROM) for non-volatile storage. Since the performance of random write operations are much slower than are random reads on flash-memory devices, algorithms that interface with flash memory need to employ different strategies than hard disk-based devices to achieve the highest possible performance. IonDB is a performance-competitive associative-array implementation capable of running on Arduino devices. This document presents an implementation of the linear hash data structure optimized for the Arduino environment that adds a new set of performance trade-offs to the current suite of IonDB configurations.

Table of Contents

Table of Figures

Acknowledgements

I would like to express my sincere appreciation to Dr. Ramon Lawrence, who invested much time in me throughout my time at UBC Okanagan. I would also like to extend my gratitude to Eric Huang, who went above and beyond his assigned duties in managing my activities on this project. The mentorship and encouragement I was fortunate to receive from these two individuals has been as inspiring as it has been illuminating. Thank you.

1. Introduction

In today's world, advances in both hardware and software happen quickly. New technologies are constantly transforming the kinds of tools and challenges facing computing professionals. One such technology is flash memory. First presented by Toshiba in 1987, NAND Flash memory has become cheaper and more pervasive in recent years, increasing the need for algorithms better suited to this environment [14]. New computing paradigms, such as the internet of things, rely largely on flash-based devices. Some of these paradigms bring with them increased number of relatively resource-constrained computing devices. As the physical structures of computers change, the interaction between high level problem-solving tools used in practice and the physical environments in which they are implemented is an increasingly important area of research**.**

Developing algorithms for these environments is a challenging and exciting task. This work presents such an algorithm. Specifically, an implementation of the linear hash data structure for embedded devices using flash memory will be discussed in detail. The paper begins by discussing the domains of embedded devices, flash memory, and the IonDB platform to make clear its motivations. Next, a detailed background and discussion of the linear hash algorithm is presented. The specifics of the implementation that satisfies the motivations of the present work are then described. A thorough analysis of the expected performance of this implementation is given in the following section. Next, the results of several simulations run on the Arduino platform are discussed. Finally, the conclusions of the present work and suggestions for future research are presented.

1.1 Embedded Devices

Decreases in the size and cost of microprocessors led to the replacement of several previously analog devices, such as variable capacitors, with small computer systems known as embedded devices. The capabilities of a single embedded device could often be used to replace several analog controllers. Modern-day cars use advanced embedded systems to perform many functions which were previously analog. This includes tasks ranging from how music is played and controlled to the ignition. Trends such as this have led to new standards in the field of embedded computing devices.

The first embedded device was produced for the Apollo space program [13]. Dubbed the Apollo Guidance Computer (AGC), it centralized interfaces for underlying computations and processes related to the space craft's control, navigation, and guidance systems. Since this time, an increasing number of domains have utilized embedded devices. Consumer electronics, military technology, and national infrastructure such as telecommunications rely heavily on the capabilities provided by embedded devices.

Today, several vibrant embedded-device projects exist within the open-source community. Among the most popular of these is the Arduino project, which produces inexpensive programmable circuit boards. The wiring for the Arduino project was created for a graduate thesis at the Interactive Design Institute Ivrea in 2003 [2]. Arduinos provide a means for people with little knowledge about electronics to partake in the development of open source hardware projects. Developing for Arduinos is very accessible, and the Arduino IDE used to create Arduino projects is available on MacOS, Windows, and Linux operating systems [1].

The internet of things paradigm has seen a lot of Arduino-driven development. Arduinos allow less experienced developers to build internet-enabled devices capable of performing networked operations. Projects such as this were previously only accessible by people with knowledge of electrical engineering. This greatly widened the number of people capable of participating in internet of things projects. The benefits of the Arduino project have been numerous for developers wanting to get started in this field.

Currently, however, inexpensive embedded devices come with relatively limited amounts of computational resources. The Arduino Mega 2560, one of the most popular modern Arduino boards, has only 8KB of SRAM and a clock speed of 16MHz [9]. With such limited processing and main-memory capabilities, many modern applications cannot run on an Arduino. Consequently, there has been a drive to implement services necessary for these kinds of applications in such a way that they can be used in more resource-constrained environments.

Algorithms which take advantage of environment-specific considerations such as behaviour of their hardware medium are now developed in tandem with the hardware they are optimized for. One type of hardware which has spawned the development of several algorithms is flash-memory. The Arduino Mega 2560 uses flash-memory for program memory [9]. It also has a microSD card interface and as such sports a relatively high amount of non-volatile, flash-memory storage. The present study exploits this knowledge in its implementation of the linear hash data structure for the Arduino platform.

1.2 Algorithms for Flash Memory

As the price of flash memory has decreased with time, storage systems previously occupied by hard disk technology are being replaced with flash. An increasing number of personal computing devices are using flash-based storage devices. Many laptops now come with solid-state drives out-of-the-box, and both the local and expandable storage devices available in smart phones are flash-memory based. As the price of flash-memory continues to decrease, the pervasiveness of flash devices will likely continue to rise.

While flash- and disk-based memory generally serve the same functions, flash memory is not just an enhanced version of the same set of behaviours performed by disk-based memory. There are some important differences in the characteristics of these two storage formats. While the access time is generally much faster on flash-memory, the data transfer rate is relatively slow. Consequently, when calculating the total time taken to read some amount of data, the block size of the system is an important consideration [5]. However, flash-memory far outperforms disk-memory when the transfer size of data is reasonable. Running programs which are purely random read operations has been shown to be as much as 20 times faster on flash than on disk [5].

The behaviour of random write operations on flash memory diverges most dramatically from disk-based memory. The physical structure of flash memory is built using groupings called erase blocks. When a write operation is performed, the individual sectors of an erase block cannot be changed directly, the entire erase block must be updated. This means there is much more overhead for random write operations on flash-memory storage than on hard-disk storage. Consequently, programs which are purely random write operations have been shown to run a much as 15 times slower on flash than on disk [5].

Flash memory also has a limited life span. It can only perform a finite number of write operations to the same physical memory location before that location wears down. Therefore, to maximize the life of a flash-memory device, it is important to keep the distribution of write operations relatively uniform amongst the physical memory locations. The technique of extending the lifetime of flash memory in this way is known as wear levelling.

The differences in the performance characteristics of flash and hard-disk storage media does not necessarily mean that certain kinds of programs will always run optimally on one medium or the other. Efficient algorithms for flash memory are currently subject of many research and development projects. One pattern commonly adopted in data structures used in flash-memory systems is that of logging operations as they are requested then performing them in a sequential batch where possible. This not only reduces the amounts of random-write operations, but also produces a more even wear-levelling. The efficient log-structured flash file system (ELF) project applied this philosophy to the development of a file system for flash-memory based micro-sensor nodes achieved near-uniform wear leveling, greatly extending the functional life of the flash-memory system [4].

Other algorithms have used this idea to achieve a faster performance. The self-adaptive linear hash discussed in detail in a later section buffered logs of requested operations in main memory to achieve a notable increase in its performance [15]. By flushing buffered records operations affecting records in the same physical block of memory at the same time when freeing buffer space, the number of random writes was decreased considerably. Even research into hash functions designed specifically for the characteristics of flash memory is being done [3]. The limitations of the physical medium can be overcome with varying degrees of success, though there is still much work to be done in this field.

1.3 IonDB

IonDB is a data management system designed for Arduinos and other resource constrained devices [6]. It uses a simple key-value store interface with several underlying implementations with different performance trade-offs that can be selected from. Currently, there is no efficient interface for data management in Arduinos natively. IonDB aims to fill this void and expand the functionality of the Arduino platform. This means that Arduino developers don't have to develop their own data structures and algorithms for this purpose.

Massive amounts of data are gathered by the kinds of devices IonDB is designed for. Deriving information from this data has both great scientific and industrial value. With no viable local data management system, using networked devices for storage and processing is commonplace. Network communications are costly however. An experiment conducted by Pottie and Kaiser showed that it takes the same amount of energy to execute 3 million instructions at 100 MIPS as it does to transfer 1KB of data over 100 meters of network links [11]. Clearly, there is great value in being able to manipulate data locally on embedded devices.

Data management systems such as MySQL are a heavily-used resource in most computing applications. They provide an abstracted way to manipulate and store data, greatly improving programmer efficiency. Unfortunately, not even the least computationally demanding relational database software like SQLite can run on the Arduino platform. Prior to IonDB, a number of relational database implementations existed for the Arduino. However, the constraints on the resources of the devices restricted the features and performance of these systems. A key initial goal of the IonDB project was to push performance of simplified data management interfaces past what was currently available.

Currently, IonDB provides an extensible, high-level interface for a key-value store that gives users several different implementations to choose from, each with their own strengths and weaknesses. As of this writing, IonDB includes a skip-list, flat-file, and disk- and main-memory-based implementations of a hash map. All of its implementations support insert, update, get, and delete operations, as well as find and range queries. The implementations all perform these operations in their respective expected times [6].

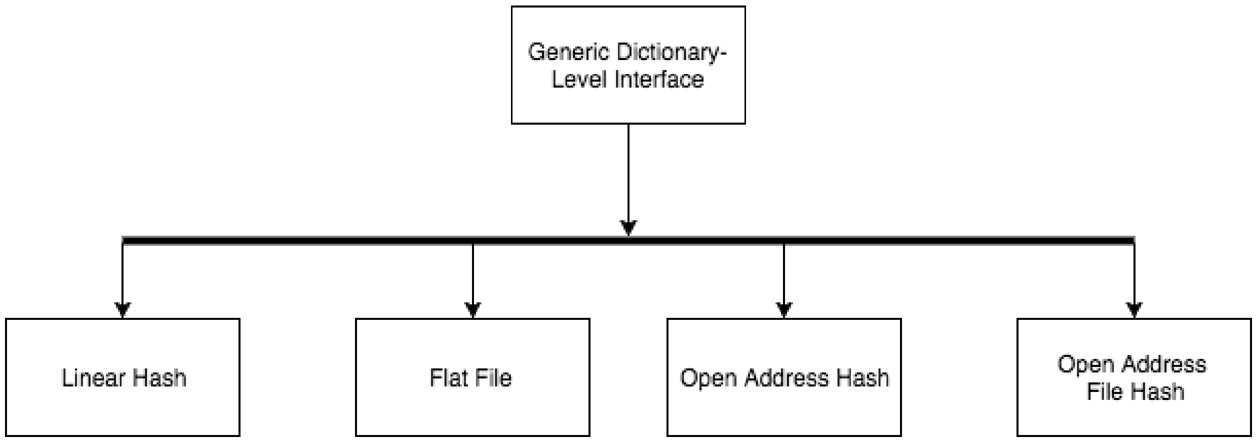

Figure 1: Graphical representation of the IonDB dictionary-level interface structure.

IonDB is open source. Through its freely available and extensible library of key value store implementations, it also aimed to educate Arduino developers on data structures and algorithms. Many Arduino developers are students, and have little exposure to programming data structures. This is especially problematic in the Arduino environment due to the lack of a data management layer mentioned previously. When using IonDB, developers can learn while using a fully exposed data management system, but don't have to start from scratch.

As more developers seek to build technologies that push the limits of the internet of things paradigm, the features offered by IonDB are becoming more widely sought. IonDB has received notable attention from the software engineering community. Featured on HackerNews in 2016, IonDB currently has 450 stars on its GitHub repository. Development on the IonDB project's GitHub repository is active. The data structure implementation discussed in the later sections of this paper was designed for use in the IonDB API on the Arduino platform.

The IonDB platform is written in the C programming language, and any example code in this document should be assumed to be C code.

1.4 Motivations

The motivations for the present study are many-fold. Hash algorithms such as hash maps are among the most useful and performant ever realized. Good hash functions allow for a near-constant time complexity for the basic hash-table operations. One consideration which must be made for hash maps which heavily affects their worst-case performance is how to handle the growth required by an unspecified number of insertions. This consideration, known as dynamic resizing, has led to a number of research efforts made in the data structures used to implement hash maps.

The linear hash data structure is dynamically resizable and has constant time complexity for the basic hash table operations. This kind of near-optimal performance is desirable in any environment, and would be an especially useful tool for resource-constrained environments such as the Arduino. As the linear hash described in the present study was made for the IonDB platform, Arduino developers can utilize the high performance of the linear hash presented through a simple interface.

Currently, there is no implementation of a dynamically resizable hash table for the IonDB platform. Adding a linear hash to the suite of implementations currently in IonDB would give developers on the Arduino platform another tool for data management. This is a very desirable thing for a platform with no accessible native data management layer. All the implementations currently available in IonDB exhibit a complexity growth of logarithmic or worse for at least some of the insert, update, get, and delete operations, or are not dynamically resizable efficiently. Adding a linear hash to the IonDB implementation suite provides a new, desirable configuration when managing data using the IonDB platform.

Since the Arduino uses flash memory, developing such a structure presents an opportunity to explore implementations that favour the performance characteristics of flash. In the Arduino environment, main memory is at a premium. As previously stated, ATmega256 model has only 8KB of SRAM and 256KB of flash memory [9]. For programs to scale with these constraints on processing and storage capabilities, optimal time and space complexity must be obtained.

These design challenges were approached in several ways. At times, strategies were explicitly used to minimize random write operations. These techniques are discussed in depth in later sections. Several implementations of flash-aware hash maps have been produced. Many of these, however, buffer log records in main memory. In environments such as the Arduino, the use of such techniques is heavily restricted. As such, other techniques for improving performance will be presented to the reader.

As Arduinos and other embedded systems become more common, improving programmer efficiency for these environments becomes of great value. IonDB and other platforms are generating the tools that will aid in such improvements.

2. Background

2.1 Linear Hashing

First introduced by Witold Litwin in 1980 [8], the linear hash data structure is a dynamically-resizable hash table which maintains constant-time complexity for all the basic hash table operations. The linear hash does this by keeping track of a small number of parameters. It achieves this using only linear space as a function of the number of records in the table. At the time of Litwin’s writing, no data structure with this level of performance was known.

The previous techniques used to increase the capacity of a hash tables often involved rehashing all the records stored after increasing the size. As the functions offered by hash maps are often desired in programs which need to deal with massive amounts of data quickly, even occasional resizing done in this way is unacceptable. Through the clever structure of the linear hash, this is avoided. Because of its desirable properties, popular database management systems such as PostgreSQL use implementations of the linear hash described by Litwin [10]. The algorithms used to achieve are as follows.

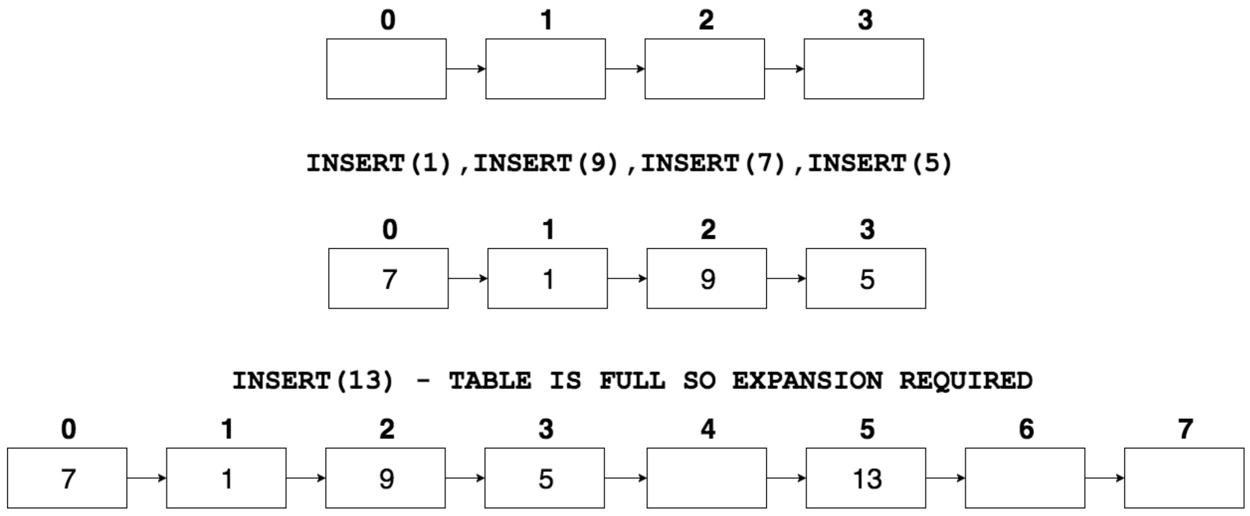
0
1
2
3
INSERT(1),INSERT(9),INSERT(7),INSERT(5)
0
1
2
3
7
1
9
5
INSERT(13) - TABLE IS FULL SO EXPANSION REQUIRED
0
1
2
3
4
5
6
7
7
1
9
5
13

Figure 2: Graphical depiction of a dynamic resizing of a simple hash table.

2.2 The Linear Hash Algorithm

2.2.1 Attributes and Parameters

The basic storage unit of the linear hash described by Litwin are buckets that all have the same, finite capacity. Upon initialization, the linear hash is allocated some number of buckets. These buckets are arranged in the fashion of a linked-list and are given an index, the first bucket being given an index of 0, the second an index of 1, and so on. Records are assigned to buckets using one of two hash functions that change throughout the life of the linear hash. The hash functions used to map keys to buckets have the property that if a key maps to bucket *m* using hash function h_0, then using *h1* it will map to the bucket with index *m* or the bucket with index *n*. When a record is assigned for insertion to a bucket that is already full, an overflow bucket is created and a pointer from the full bucket to the overflow bucket is assigned to the full bucket.

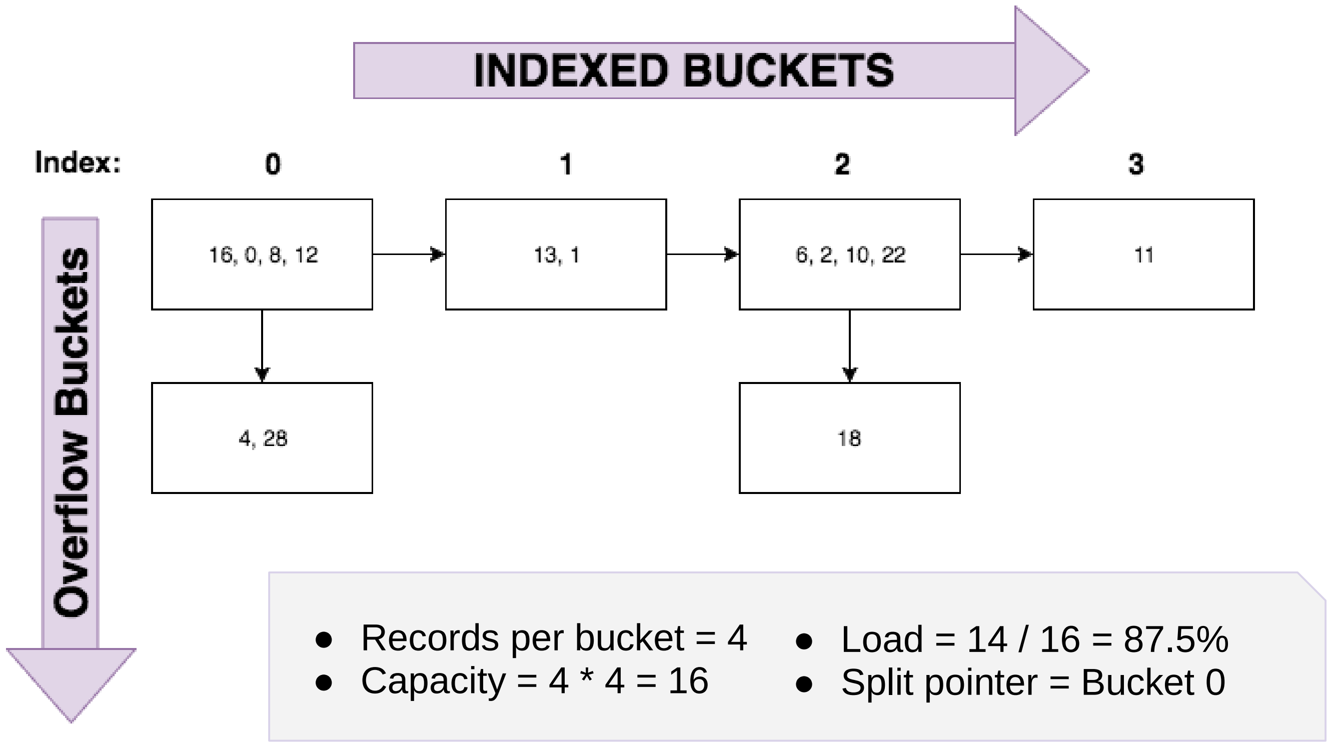

Figure 3: General structure of a linear hash table as described by Litwin.

The load of a linear hash table is the number of records in the table divided by the capacity of that table. The capacity of the linear hash table is the number of non-overflow buckets in the table multiplied by the capacity of buckets. Note that the capacities of overflow buckets are not factored in to the load of the linear hash, but the records they contain are. When the table reaches a certain load, a new bucket is created and about half of the records from one of the buckets in the table are reassigned to this bucket. This operation is known as a split, and it is what allows the linear hash to maintain its constant time performance while growing dynamically. The load at which splits are performed is declared upon initialization. A pointer tracks which bucket half of the records will be taken from during each split. This structure requires the tracking of several parameters, summarized below.

Table 1: List of the parameters of a linear hash table

Symbol	Value
t	*The load which, if exceeded, triggers a split*
s	*The initial size of the linear hash*
r	*The maximum number of records allowed in a single bucket*
b	*The number of indexes in the bucket list*
n	*The total number of records in the table*

capacity	*$r * b$*
i	*The index of the bucket to split next*
load	*n / capacity*
d	*The number of times the number of buckets in the table has doubled.*
h_0(key)	*hash(key) mod $s * 2^d$*
h_1(key)	*hash(key) mod $s * 2^{d+1}$*

<u>2.2.2 Basic Operations</u>

Insert

Insertions into a linear hash table happen in a constant number of disk accesses. The algorithm for insertions into a linear hash table is similar to simple hash tables. Recall that the linear hash tracks the index of the next bucket to be split *i*. Because of periodic redistribution of some of the records from the bucket with index *i* to the newly created bucket, the proper location to insert the record at could be in one of two places. If the result of h_0 is less than the index of the next bucket to be split, the bucket has already been split and the hash function h_1 is used. The terms 2^d and 2^{d+1} that appear in h_0 and h_1 respectively change as the number of buckets in the table changes. This allows for expansion of the address space that h_0 and h_1 map keys to.

Two additional processes can be triggered by inserting records into a linear hash table. Recall that the buckets in a linear hash table have a finite capacity. When a record is mapped to a bucket that is currently full, an overflow bucket is created. This requires the writing of a new bucket to disk as well as the updating of the current tail in the linked list of overflow buckets being inserted into. The gains in overall performance from this operation exceed the overhead it requires.

The other process which can be triggered by an insertion is the split operation. This operation requires the writing of a new bucket, as well as several insert and delete operations. Details of the split operation are discussed below. Fortunately, the proportion of records in at a bucket index and the rate at which the load increases are both inversely proportional to the number of buckets in the table. This means that as the size of the linear hash grows, the frequency of the creation of overflow buckets and split operations decreases as the table grows.

```
insert(key k, value v):
        bucket_index = $h_0$(k)
        if bucket_index < linear_hash.i
                bucket_index = $h_1$(k)
        bucket = get_bucket(bucket_index)
        bucket.append({k, v})

        if (bucket.number_records == table.records_per_bucket)
                create_overflow_bucket(bucket_index)

        if (table.load > table.load_factor)
                split(table.split_pointer)
```

Figure 4: Algorithm for insertions into a linear hash table.

Delete

The algorithm for delete operations in a linear hash is generally the same as a simple hash table with one modification. Because of periodic redistribution of some of the records from the bucket with index *i* to the newly created bucket, the key could be in one of two places depending on the state of the linear hash.

To determine which bucket to search for the key in, an additional check is again required as it was in the insert. If the bucket has not been split since the last time the next split index *i* was reset to 0, we search for the record to delete in the bucket with index $h_0(key)$. If this bucket has been split, we search for the record in the bucket at index $h_1(key)$.

The empty space created when a record is deleted from the table must be managed in some way for the linear hash to maintain its performance. One strategy commonly employed for this purpose is known as tombstoning. This involves marking the records with some sort of status flag, or “tombstone”, that signifies it is now empty space that can be reused. Then, when a record is inserted into the table, a scan through the chain of overflow buckets at the index a record is mapped to is done to see if there are any tombstones marking empty space that can be reused. This is very space efficient, however there is a lot of overhead as scans of the entire chain of overflow buckets is necessary to perform an insert operation. This is especially undesirable in situations where a large number of insertions are expected, such as in the logging of sensor data.

Another technique used to handle this issue is known as swap-on-delete. The swap-on-delete algorithm is used to ensure that there are no holes in a list structure. When a deletion occurs in the list, the terminal record is removed from the end of the list and used to fill the hole created by the deletion. This guarantees that if any empty space exists, it will be immediately after the last entry currently in the list. This algorithm favours insertions over deletions as some overhead is required to read the terminal record into memory and delete it from the end of the list. Insertions, however, become instant, as all the location of freed space (if any is available) is always known.

Some linear hash implementations support a contract operation which is essentially the reverse of a split. When the load factor falls below some threshold, a bucket is removed and its records are redistributed to another bucket. The split pointer is then decremented. This operation requires significant overhead periodically on deletions, and is beneficial when non-volatile storage is at a premium. However, as this was not a key limiting resource in the Arduino environment, the contraction operation was not used in the implementation discussed in the following sections.

```
delete(key k, value v):
    bucket_index = $h_0$(k)
    if bucket_index < linear_hash.i
        bucket_index = $h_1$(k)
    bucket = get_bucket(bucket_index)

    while bucket != null
        for each record in bucket
            if record.key == k
                remove record form the table
        bucket = bucket.overflow_bucket
```

Figure 5: Algorithm for deletions from a linear hash table.

Split

The periodic splitting and adding of buckets is what makes the linear hash more performant than other hash tables. Splitting makes the load of the linear hash remain relatively constant as the number of records in the table grows. As previously mentioned, splits are triggered when the linear hash has a load exceeding the threshold parameter *t*. The check for this event is done after a record is inserted in the table.

The split operation involves several steps. Frist, a new bucket is created with an index of *n* and is added to the table, where *n* is equal to the largest index currently in the table + 1. Next, the bucket at index *i*, the index tracked by the linear hash as the index of the next bucket to split, is read in to memory. For each record in this bucket, h_0 and h_1 applied to its key and the results are compared. If the result of h_0 and h_1 are not equal, the result of h_1 will be equal to the index of the new bucket just created. The record is then deleted from the bucket it is currently in and inserted into the new bucket. This same process is then applied to all the overflow buckets for the bucket chain at index *i* at the time the split was triggered.

After the split is complete, the linear hash will be below its split threshold *t*. The index of the bucket to split during the next split operation *i* is incremented at this time. If the number of buckets in the table has doubled since the last time *i* pointed to index 0, *i* is reset to 0 and the number of times the table has doubled *d* is incremented. This is done to prevent buckets which have been in the table longer (i.e. the buckets at the lower indexes) from accumulating disproportionate amounts of records.

While the split can be relatively expensive when compared to the insert, update, get, and delete methods as it requires multiple insertions and deletions, the cost of splits remains constant as the table grows. This is possible because of the expansion of the address space that occurs when *d* is incremented. Recall that *d* appears in both the h_0 and h_1 and hence they are changed when *i* is reset to 0. Because of this, the overhead cost of occasionally performing splits is more than made up in the long run by the properties it guarantees.

```
split():
    bucket = get_bucket(linear_hash.i)

    while bucket != null
        for each record in bucket
            if $h_0$(record.key) != $h_1$(record.key)
                remove record from current bucket
                insert record into new bucket
        bucket = bucket.overflow_bucket

    linear_hash.i++

    if linear_hash.i == linear_hash.s * $2^{linear\_hash.d}$
        linear_hash.i = 0
        linear_hash.d++
```

Figure 6: Algorithm for performing a split operation.

Get

The algorithm for the get operation even more closely resembles a simpler hash table than the delete and the insert algorithms do. To perform a get, the bucket to look for the record is resolved the same way as it was in the delete by comparing the results of $h_0(key)$ and the index of the next bucket to split. If bucket has been split, that is, if $h_0(key)$ is less than i, then the bucket at index $h_1(key)$ is searched. The records in the bucket at the index determined in this way are then checked to see if they have the key specified. The first record with the key specified is then returned.

```
get(key k):
    bucket_index = $h_0$(k)
    if bucket_index < linear_hash.i
        bucket_index = $h_1$(k)
    bucket = get_bucket(bucket_index)

    while bucket != null
        for each record in bucket
            if record.key = k
                return record
        bucket = bucket.overflow_bucket
```

Figure 7: Algorithm for gets from a linear hash table.

Update

The algorithm for the update operation is essentially the same as the get operation except it updates the first record encountered instead of returning it. To perform an update, the bucket to look for the record in is resolved the same way as it was in the previous methods. The records in the bucket at the index determined in this way are then checked to see if they have the key specified. The value of the first record with the key specified encountered is set to the value specified in the update operation.

```
update(key k, value v):
    bucket_index = $h_0$(k)
    if bucket_index < linear_hash.i
        bucket_index = $h_1$(k)
    bucket = get_bucket(bucket_index)

    while bucket != null
        for each record in bucket
            if record.key = k
                record.value = v
                break
        bucket = bucket.overflow_bucket
```

Figure 8: Algorithm for updates on a linear hash table.

2.3 Variations of the Linear Hash

2.3.1 Log Records

As discussed in the beginning of this paper, a key difference between flash- and disk-based memory is the performance of write operations. The reason for this is that individual sectors in the memory array could not be manipulated because of the physical structure. Individual sectors are grouped together in erase blocks which are usually much larger than a page, greatly affecting the performance of random writes. Designed with this in mind, some hash table implementations buffer logs of successive operations before flushing the result to disk. This can often decrease the total number of read and write operations required to arrive at the same result, and, in some environments, allow for some writes that would have been performed randomly otherwise to be performed sequentially. This is a desirable result in any environment, but especially when the cost of random write operations is very high and large numbers of them are expected.

As a simplified example, consider a basic key-value store structure where no duplicate keys are allowed. If we were to give the hash table the operations INSERT(1, 5), UPDATE(1, 10), UPDATE(1, 5), DELETE(1), there would be no record with key = 1 in the key-value store after all operations are completed. However, achieving this state took 5 operations. Since the last operation was a delete and the data was not needed for any get operations after it was inserted and before it was deleted, nothing about the state of the key-value store actually needs to change, and 3 write operations can be avoided.

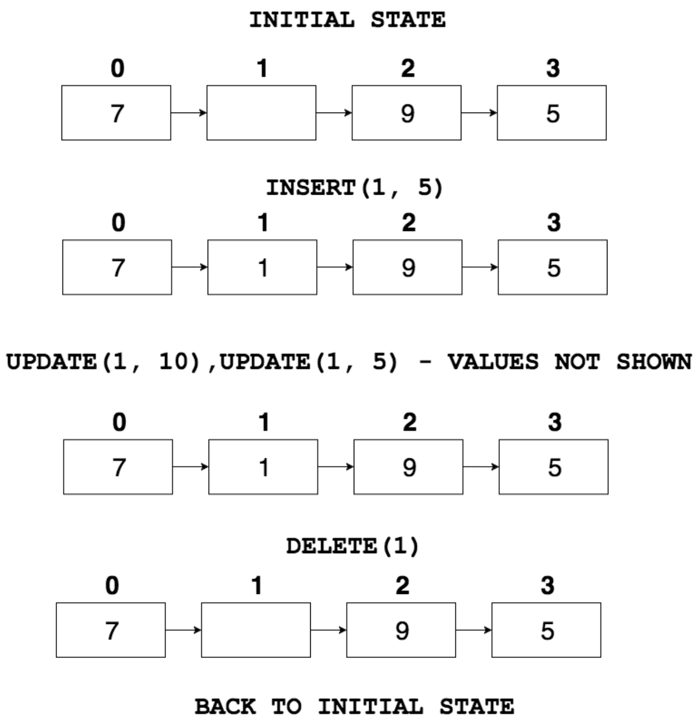

Figure 9: Example of a series of operations which has operations that can be removed with no consequence. Since the record with key 1 was deleted after insertion and was never needed in a get operation, the insertion and the updates can be skipped and the same state will still result.

To achieve this efficiently in practice, the log records that store which operations are performed are buffered in memory. Exact implementations differ, but some general patterns are as follows. Typically, when a buffer is full, inconsequential operations are removed from the buffer. If none can be removed, then data needs to be flushed. If possible to guarantee that certain pieces of data are stored sequentially, the largest string of sequential data is flushed from the buffer to disk. Several hash tables have applied this idea in practice, including variations on the linear hash such as the implementation of the Self-Adaptive Linear Hash described in the next section [3].

This technique of using logs records to improve of the performance of data structures has been applied for many years. In 1992, Rosenblum and Ousterhout introduced a log-structured file system for environments in which reads were assumed to be cheap and writes costly [12]. Their implementation far outperformed the Unix file system in terms of bandwidth usage, achieving a bandwidth usage of 70% in environments where Unix file systems could only achieve between 5% and 10%.

Projects have since applied this idea to achieve other desirable effects such as the previously mentioned ELF data structure which achieved near-uniform wear leveling [2]. Clearly, log records are a useful algorithmic tool, and can mitigate some of the issues that come with using flash memory.

There is of course a main-memory cost associated with buffering anything. This makes the technique of logging operation records difficult or even futile in heavily constrained environments. Consequently, it was not used in the embedded flash implementation of the linear hash described in later sections. Examining the viability of this technique for the linear hash in the Arduino environment could be a fruitful research endeavour.

2.3.2 Self-Adaptive Linear Hashing

A specialized implementation of the linear hash for solid state drives known as the Self-Adaptive Linear Hash (SAL) takes the use of log records a few steps further. Presented by Yang, Jin, Yue, and Zhang, the SAL adds higher levels of organization to the linear hash to achieve more coarse-grained writes to improve the bandwidth [3]. In the SAL, a *group* contains some set number of buckets. When the load factor of a SAL is exceeded, an entire group is created instead of a single bucket. A *set* in the SAL is composed of a number of groups. Attached to each set is a log of operations that affects only members of that set. This amortizes the cost of writes by guaranteeing the locality of buckets and thus writes can be performed at a coarser grain for improved bandwidth. To try and determine in advance the location of an update operation for a particular bucket in the log of operations for the set it belongs to, bloom filters were used.

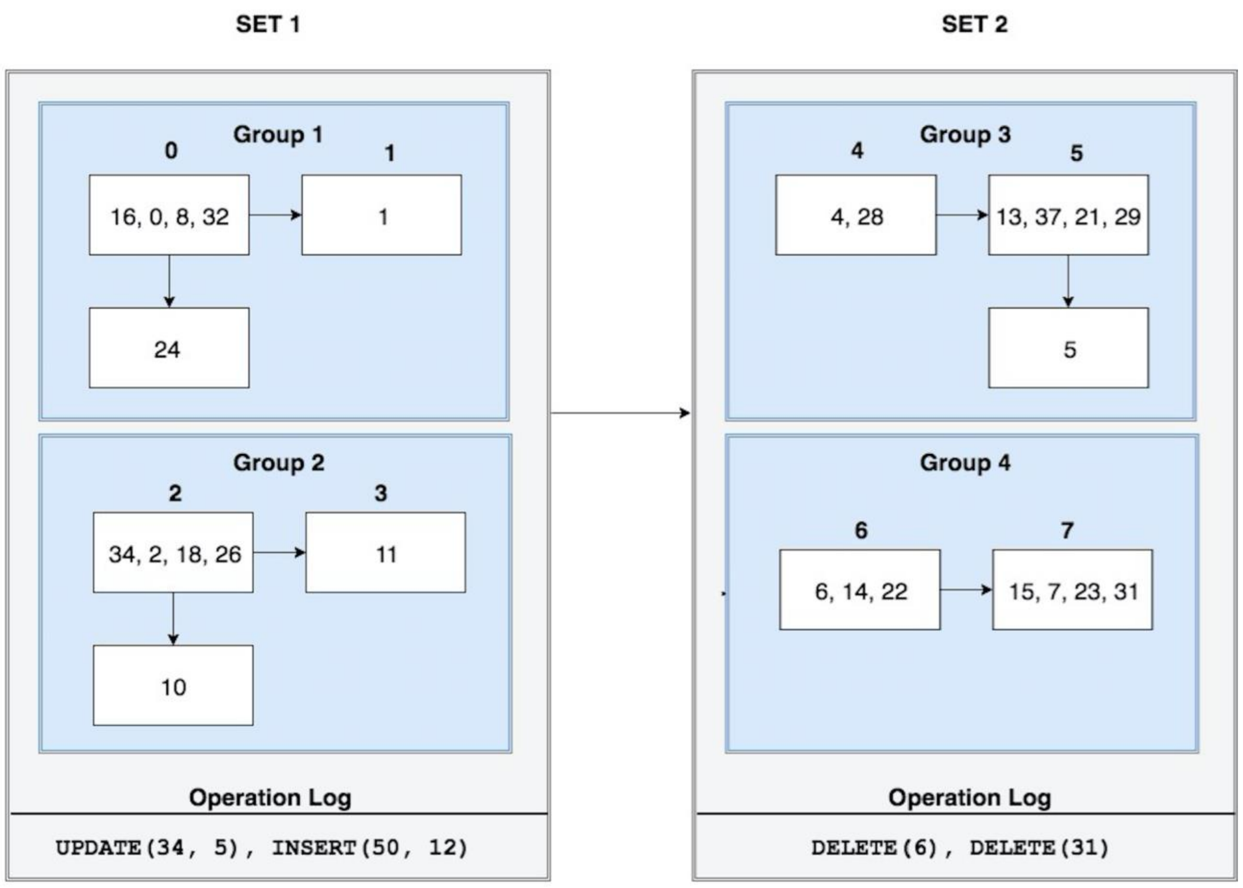

Figure 9: The structure of the Self-Adaptive Linear Hash

There are costs associated with having these logs of update operations for every set. When performing a get operation, it is required that the logs be examined. Otherwise, there is no way to ensure that the result reflects the true state of linear hash after all the operations specified thus far. The overhead associated with this structure grows with the number of log records stored for a set. Thus, using these structures alone, there is a trade-off made between read and update performance.

Thankfully, the way in which the SAL is "self-adaptive" mitigates these undesirable effects. An online cost-based algorithm is used to detect sets which are frequently searched. If such a set is detected, the operation logs are fully merged so that they no longer need to be searched. This may greatly offset the cost of the gains in write performance. However, there are certain problem instances in which the read performance of an SAL will lag behind a linear hash which is not organized in this way.

2.3.3 Spiral Storage

Spiral storage is a technique that can be used in combination with linear hash. By applying an additional transformation to the result of the linear hash bucket mapping functions h_0 and h_1, spiral storage intentionally distributes the records amongst the buckets unevenly. Whenever a bucket is added to the linear hash, the address space is expanded. New space is added in the form of the new bucket, and some space will be freed in the bucket that will be split during this expansion.

While the expected performance of the linear hash is constant, there is some cyclical variation. In simple terms, this is because the buckets in the lower end of the address space have been around for longer. While the split operation, the periodic resetting of the split pointer, and the updating of the hash functions largely deals with this, a segment of the address space remains slightly bloated.

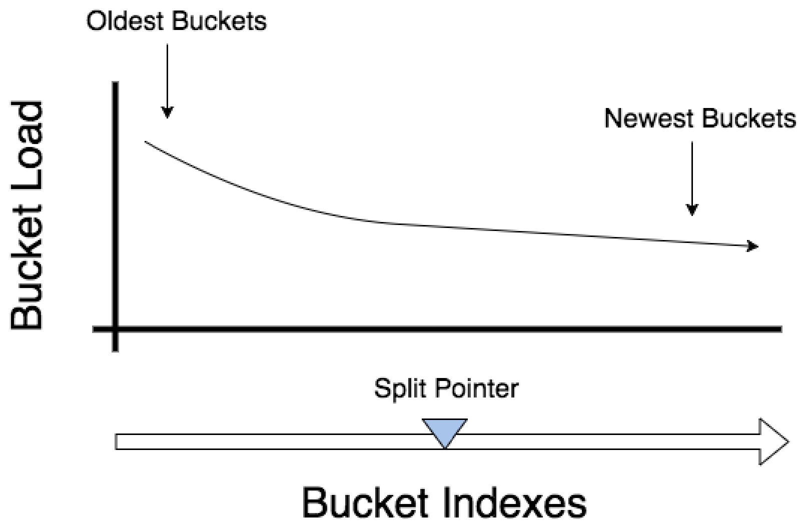

Figure 10: Distribution of records in buckets relative to split pointer. Spiral storage can be used to make the distribution of records more uniform.

This may seem unintuitive at first, but consider this example. Suppose a linear hash table that had an initial size of 5, had 9 buckets, and had a split pointer pointing to the bucket at index 3. That means that buckets at the indexes 0, 1, and 2 have been split. For the purposes of this example, imagine that a record is inserted to a bucket at random every time step t_i. This means that at any time t_i a bucket *b* would have (t_i – *b.time_created) / linear_hash.n* in it if *b* has not been split and (*b.time_split / linear_hash.n)*/2 + ((t_i –*b.time_split) b.time_created / linear_hash.n* records if it has been split. Since for all buckets b_i in the linear hash table that have an index less than s * 2^d there is a bucket b_j such that b_i*.time_split* = b_j*.time_created* and that, given any reasonable hash function, (*b.time_split / linear_hash.n)*/2 will be greater than 0, b_i*.record_count* = b_j*.record_count*. This will difference will remain the same for b_i and b_j until the next time the split pointer is reset and both b_i and b_j are split again.

While this is a very contrived example, it illustrates why spiral storage techniques can be beneficial for the linear hash. These were not used in this first iteration of the linear hash data structure created for IonDB. The implementation described in the following sections could benefit from such an addition, and such a research project could further push the limitations of hash indexes for embedded devices. The mathematics behind the function used to achieve this property are beyond the scope of this paper. A spiral-storage based hash index which makes use of these techniques is described by Larson in [7].

3. Embedded Flash Implementation of the Linear Hash

3.1 Structures

3.1.1 Linear Hash Table

The implementation used a set of parameters necessary to capture the information required for the algorithm described in the previous section. The initial_size attribute of the linear hash type defined was used to store the value of linear_hash.s * 2 $^{\text{linear\_hash.d}}$, which is the value used in h_0 and h_1. Doing so saved on repeated calculations. In practice linear_hash.s and linear_hash.d parameters described previously are not required for anything else. The index of the next bucket to split, the load threshold at which splits occur, total number of records, total number of buckets, the total size of a record, and the number of records per bucket allowed in the table were all tracked either out of necessity or to avoid repeated calculations.

To store the data for the linear hash, two data files were used. The first data file was used to store information about the linear hash's state. This file was only read from on initialization and written to on destruction. This state data file was tracked by the state attribute on the linear hash. The second was used to store the record and bucket data currently in the linear hash table. This data file was written to repeatedly throughout the use of the linear hash methods. The database attribute was used to track the pointer in this data file.

As previously mentioned, the implementation being discussed was made for the IonDB platform and thus conformed to its design specifications. The IonDB platform has a generic interface called the dictionary interface. To implement polymorphism in the C programming, IonDB defines a dictionary type which serves as the super class for the low-level implementations of the key-value store like the linear hash.

Conforming to this, linear hash type defined had an attribute titled super which contained data that would be used by any of the low-level implementations such as the key type, key size, and value size. A generic dictionary struct tracks a pointer to the instance being used and a handler struct which is a collection of function pointers that are set to the methods of the implementation level structures such as the linear hash. With this organization, IonDB can provide a generic interface for the implementations. After initializing the structs, the linear hash methods can be called using a function call such as dictionary->handler→get(key).

Two caches were used. One served a mapping of indexes to file offsets in the data file. This was pointed to by the bucket_map attribute. The other was used to store data that would have otherwise been lost due to the split method. This attribute was simply titled cache as it was just a generic array of bytes that could be used for any purpose. The uses of these two attributes are described in detail in the following sections.

```
typedef struct {
  ion_dictionary_parent_t super;
  ion_dictionary_size_t  dictionary_size;
  int                initial_size;
  int                next_split;
  int                split_threshold;
  int                num_buckets;
  int              num_records;
  int              records_per_bucket;
  ion_fpos_t           record_total_size;
  FILE                 *database;
  FILE                 *state;
  array_list_t         *bucket_map;
  ion_byte_t           *cache;
  int              last_cache_idx;

} linear_hash_table_t;
```

Figure 11a: Example of the definition of the type implementing the embedded flash implementation of the linear hash.

3.1.2 Buckets

A very simple structure was used for the buckets. All that had to be tracked were the index of the bucket, the number of records it contained, and the location of its overflow bucket in the data file. The same struct was used for both overflow and non-overflow buckets. To indicate that a bucket did not have an overflow bucket (i.e. that it was the terminal bucket in a linked list of overflow buckets) a special value was used for that bucket's overflow location.

```
typedef struct {
        int                   idx;
        int                   record_count;
        ion_fpos_t    overflow_location;
} linear_hash_bucket_t;
```

Figure 11b: Example of the definition of the type implementing the buckets used in the embedded flash implementation of the linear hash.

3.1.3 Records

No actual struct was defined for the record data. Whenever needed in main memory, two byte arrays and one single-byte variable were defined to store data read from the data file (an example of this process is shown below). IonDB supports keys of multiple types which are not guaranteed to be the same size. This is information is stored in the super parameter and defined upon initialization. As such, a statically defined struct for linear hash records was not appropriate. Consequently, the total size of a record for this linear hash was computed upon initialization of the linear hash and added as a parameter to the linear hash state information in main memory to avoid repeated calculations.

In addition to having a key and a value, records also had a status flag that consisted of a single byte. If a record was deleted, this status was given a value of linear_hash_record_empty, signifying that it could treated as empty space during an insertion.

```
ion_byte_t    *key             = alloca(linear_hash->super.record.key_size);

ion_byte_t    *value = alloca(linear_hash->super.record.value_size);
ion_byte_t    status = linear_hash_record_status_empty;

// read entire record into main memory
fread(record, linear_hash->record_total_size, 1, linear_hash->database)

// read copy individual data elements to structures
memcpy(status, record, sizeof(*status));
memcpy(key, record + sizeof(*status), linear_hash->super.record.key_size);
memcpy(value, record + sizeof(*status) + linear_hash->super.record.key_size,
linear_hash->super.record.value_size);
```

Figure 12: Example usage of record data for the embedded flash implementation of the linear hash.

3.2 Swap-on-Delete

Swap-on-delete is a technique used for managing empty space caused by deletions in a list. The algorithm does this by ensuring that there are no holes in a list of items. Whenever an item is deleted in the list, the item at the end of the list is plucked from the end and placed in the hole where the deleted previously was. In our implementation, swap-on-delete was used on the logically contiguous list of records that exists amongst a chain of overflow buckets. This means that whenever a record was deleted in any bucket, the last record in the last bucket of the overflow chain was read into memory and written to the location the previous record on disk and in the cache. For the linear hash, using the swap-on-delete technique guarantees that if any free space is available it will be in the first open space in the last bucket of that bucket's overflow chain.

The previously discussed tombstoning technique involved simply placing a status flag on records and assigning this attribute the tombstone value being used when it is deleted. Tombstoning was used in the early stages of development but was found to be less performant than swap-on-delete for the IonDB specification. Since tombstones just bloat the table with junk data unless they are overwritten, a deterioration of performance can often happen when using tombstoning. This is especially true for the linear hash which must scan through an entire linked list of overflow buckets to look for available space during an insertion when using naïve tombstoning. Swap-on-delete guarantees us this knowledge at all times. While there is some overhead performance cost during the delete method to implement swap-on-delete, insertion time was greatly improved after its implementation.

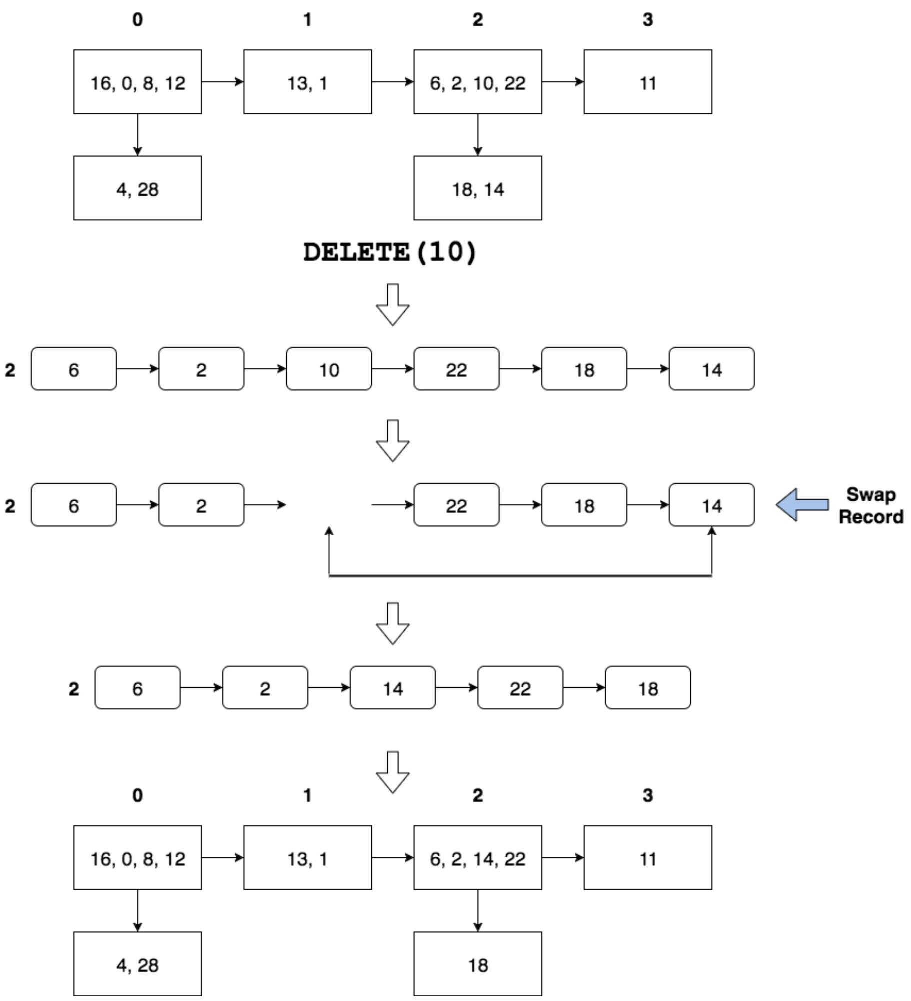

Figure 13: Graphical representation of the swap-on-delete algorithm.

3.3 Eager Deletions During Swap

Using swap-on-delete required an additional three disk accesses. First, the bucket has to be read in to memory to obtain the swap record. Second, the swap record needs to be written to the bucket where there is now a hole. Since the cached was now invalid as records have moved from the tail bucket to the bucket currently in main memory on disk, a disk read is required to update the cached bucket.

This overhead can be reduced slightly when deleting as per the IonDB specifications, however. Since all records with the key specified are to be deleted for a delete operation, if a swap record has the same key as the one specified for deletion, it too can be deleted without being inserted into the cache. Since the swap record is guaranteed to be the last record in the terminal overflow bucket, no holes need to be filled to maintain the benefits of swap-on-delete when this done. This eliminates the need for the additional reads and writes associated with deleting that record when it would otherwise have been encountered during the scanning of all the records in the bucket chain.

Using this strategy did impose some design challenges, however. During the split operation, records are redistributed from one bucket to another. This was implemented as series of insert and delete operations. Since a single delete operation could delete multiple records, the values associated with those records needed to be saved for re-insertion. To achieve this, a small generic cache of bytes was used. The gains received from using this strategy are proportional to the number of duplicate keys in the table.

```
eagerDelete(delete_record, swap_record):
        while(delete_record.key == swap_record.key)
                swap_record.status = deleted
                cache.add(swap_record.value)
                swap_record = linear_hash.getNextSwapRecord()
```

Figure 14: Algorithm used to implement the eager deletion strategy used.

3.4 Bucket Caching

To improve the performance of operations that may be performed on all records in a bucket such as key comparisons in the get operation, the entire contents of buckets were read in to memory in a single read. During the iteration over the records of the bucket, the individual records were then copied from the cached data into variables that could be manipulated. An example of this process is shown below. This reduces the amount of reads when scanning a bucket from the number of records currently in that bucket to 1 provided that the total size of the maximum amount of records in a single bucket is less than the page size of the device being used.

In some cases, however, the record cache required refreshing. The use of the swap-on-delete technique at the bucket level meant that the structure of the data used to fill what's currently in the cache was mutated. To work around this limitation, the cache was refreshed and the iteration was restarted every time a record in that bucket was deleted. This saved a number of disk accesses that was proportional to the number of records per bucket.

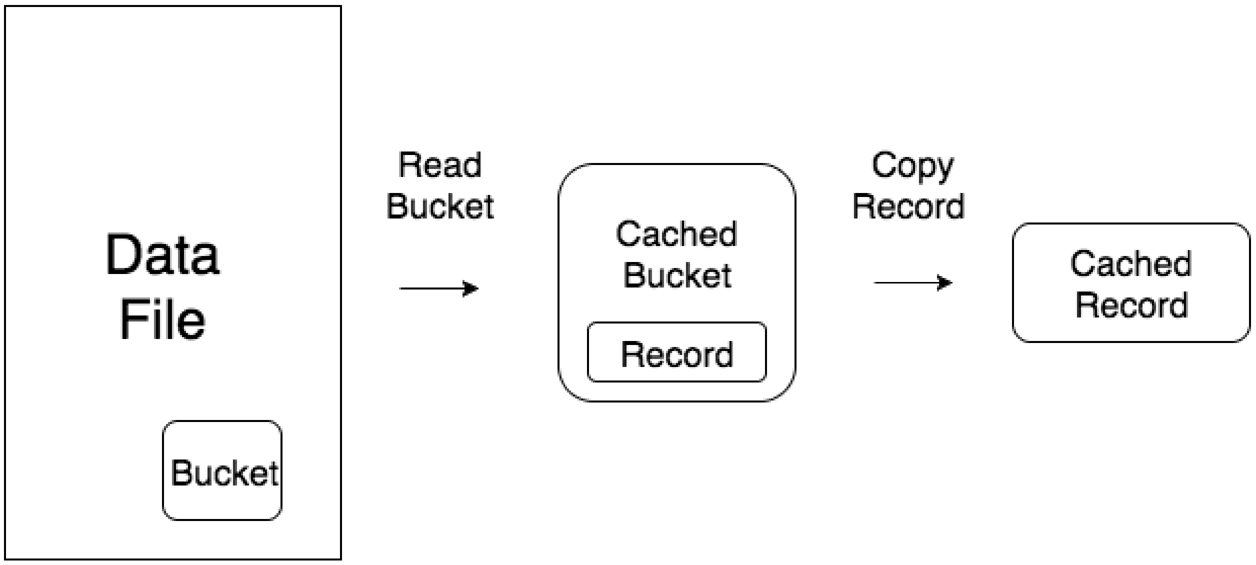

Figure 15: Graphical representation of the caching strategy used.

3.5 Linked List Structure

To remove the need for an additional read and write operation, insertions of new overflow buckets into the linked list were handled slightly differently than a traditional linked list. Usually, when a new node is added to the tail of a linked list, it is required that the previous tail be updated so that it points the new tail. To have this operation reflected in the data file at least two disk accesses must be performed. First, the newly created bucket must be written to disk. Second, the previous tail must be updated, which can be done in a single write operation in the simplest case

To avoid the need for the latter of these two write operations, newly created overflow buckets were given pointers to what would have been the previous head of the overflow bucket list it belongs to. The linear hash then indexes the location of the new overflow bucket which is now the new head of its bucket chain. This means that whenever a bucket chain is scanned, the first bucket read in to memory was the most recently created overflow bucket. The original bucket indexed by the linear hash always serves as the tail of the list of buckets.

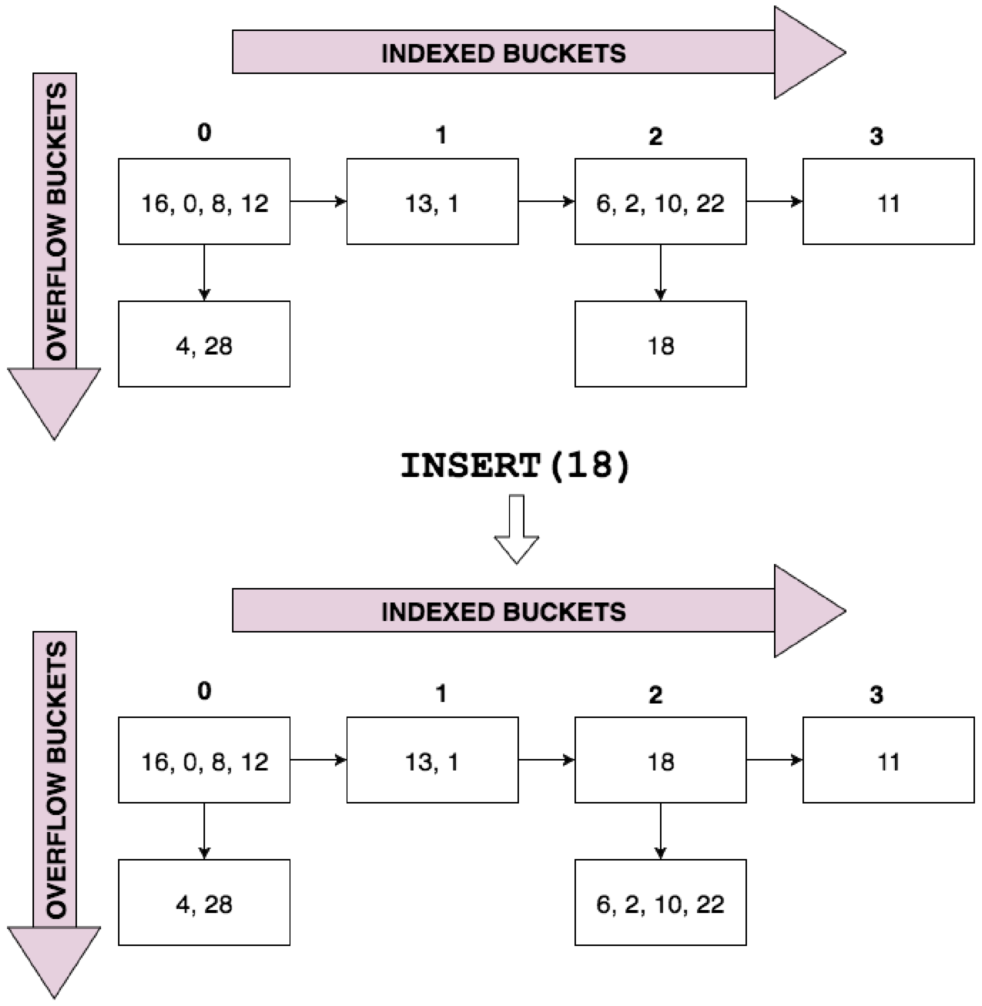

Figure 16: Graphical example of adding an overflow bucket to the embedded flash implementation of the linear hash.

To index the locations of the heads of the overflow bucket chains, a simple array list was used. The array list implementation used only needed to track its current size and a pointer to the its data array. The file offsets stored in the bucket map were stored in main memory. Since the bucket indexes were integers, the indices of the array could serve as the key in the key-value mapping of bucket indexes to file offsets. Whenever an index larger than the size of the table was given for an insertion, the table was expanded.

3.6 Mod 2^n by Bit Shifting

Division is the most expensive of the basic arithmetic operation computationally speaking. Modulo is even worse as it requires multiple divisions. It is unfortunate then that all lookups done in a linear hash table require this operation, sometimes even twice. By using an initial size that was a power of two, a useful logical equivalency could be exploited to make this a non-issue. It can be proved that for any power of 2, taking the bitwise and of that number - 1 is equivalent to the modulo operation. This eliminates the need for expensive division operations. As such, the implementations of the bucket mapping functions h_0 and h_1 use this operation, and the initial number of buckets in a linear hash table must be a power of 2.

$$h_0(key) = key \;\&\; (s^{d} - 1)$$

$$h_1(key) = key \;\&\; (s^{d+1} - 1)$$

Figure 16: The bucket assignment functions used to map values to the address space currently used in the linear hash.

3.7 Polynomial String Hashing

To distribute keys amongst buckets evenly, a polynomial string hash was applied to the raw bytes of the array key passed in. The results of this function were satisfactory, achieving a relatively even distribution of records amongst the buckets. First, the bytes of the key were transformed in to an integer. Then, the string of bytes that represents this integer had the polynomial string hash applied to it. Finally, the result of this transformation was passed in to the bucket assignment functions h_0 and h_1. The details of the performance of this hash function are discussed in the results section.

```
int polnomialStringHash(int key) {
        int hash                = 0;
        int i;
        int size_of_int = (int) sizeof(int);
        int coefficients[] = {0, 3, 7, 9, 53, 67, 5, 99};
        for (i = 0; i < size_of_int; i++) {
                hash += *(&key + i) + (i * coefficients[i]);
        }
        return hash;
}
```

Figure 17: The polynomial string hash used in the embedded flash implementation of the linear hash.

4. Analysis

4.1 Insert

The implementation described in Section 3 requires a varying number of computations made. In the most basic case, an index is calculated, a bucket is read to determine its first empty location, and the record is written to the first free location. This most simple case requires 2 disk accesses.

If a bucket is full and an overflow bucket is created, then a new bucket is written to the data file and the record is written to the first location in that bucket. These additional operations are required approximately ((1/records_per_bucket)/number_of_buckets)) of the time. As this number is inversely proportional to the number of buckets in the table, the frequency with which it is necessary to create an overflow bucket decreases over time. When creating an overflow bucket, a total of 3 additional disk accesses were made in the algorithm used. First, a bucket was read and found to be full. Next, a new overflow bucket is created and written to disk. A record is then written to the new overflow bucket on disk. Finally, the count of the records in the bucket was incremented by 1 and the bucket was updated on disk.

While it is possible to reduce avoid the writing of the record and the new overflow bucket separately, as well as the additional updating of the bucket after the record is inserted, specialized methods are required. Future refactoring of this implementation of this linear hash could benefit from including such specialized methods to further performance.

$$\textit{expected insertion disk accesses} \;=\; 4\left(\frac{1}{\textit{records per bucket}}\right) + 2\left(\frac{\textit{records per bucket} - 1}{\textit{records per bucket}}\right)$$

Figure 18: Formula to calculate the expected number of disk accesses required by a single insert operation.

4.2 Delete

To complete a delete operation, all overflow buckets in the linked list of overflow buckets at the index that the key specified for deletion maps to must be checked. Since, however, the calculation for the capacity of the linear hash table (which is then used in the calculation for the load of the linear hash table) does not count overflow buckets, there should not be more than one or two overflow buckets at most at any index given a reasonably good hash function. However, averaged over the entire table, there will be few as the load should rarely exceed the threshold, which should always be less than one. An estimate of the mean number of overflow buckets in the present implementation was determined using a simulation and was found to be approximately 0.3.

As mentioned in a previous section, the swap-on-delete algorithm used to manage empty space efficiently requires some overhead. Specifically, when a record with the key specified for deletion is encountered, the bucket containing the swap record is read, the swap record itself is read, the swap bucket is updated, the data on disk that defines the bucket currently cached is updated on disk, and the bucket currently cached in memory is refreshed. This means that if only one record has the key specified for deletion, a total of 6 disk accesses are required. There are a few processes here which could again be further reduced, however for code simplicity and reuse they were performed thusly. These optimizations can be added reasonably easily in future work.

When more than one record has the key specified, some of the overhead is reduced. As previously described, if a record to be inserted into a newly freed slot by the swap-on-delete algorithm has the key specified for deletion, it is just deleted immediately. This means that the portion of the data file that defines the cached bucket does not need to be updated, and the cache does not need to be refreshed. This gain is proportional to the average number of records that share a key with any given record.

$$\text{expected deletion disk accesses} \;=\; 6 \times \text{average records per key} \;-2\left(\frac{\text{average records per key}}{\text{average records per index}}\right)$$

Figure 19: Formula to calculate the expected number of disk accesses required by a single delete operation.

4.3 Split

Split operations are triggered by insertions require even more additional processing. If interested in the worst-case time for insertions, it would be the time of the split. On average, half of the records in a bucket being split will need to be deleted and re-inserted. Assuming a good hash function is being used, approximately ($t * r$) / 2 records will need to be processed in this way. Since the ratio of the records in the table to the number of buckets indexed by the table remains constant as the linear hash expands, the cost of splits remains constant as the size of the table grows. The split operation is just a series of delete and insert operations. As such, the number of disk accesses performed by a split can be expressed in terms of the number of disk accesses performed by these operations individually.

$$\text{expected split disk accesses} \;=\; \left(\frac{\text{average records per key}}{2}\right)(\text{expected insertion disk accesses} \;+\; \text{expected deletion disk accesses})$$

Figure 20: Formula to calculate the expected number of disk accesses required by a single split operation.

4.4 Get and Update

The algorithms used for the get and update methods are almost identical. As previously mentioned, the average number of overflow buckets was determined to be approximately 0.3 in the present implementation. All the records in each bucket in the bucket chain at the index that the key maps to are checked to see if their key matches the one specified in the operation. In the case of a get operation, once a record is found its value is just returned and no additional accesses are required. In the case of updates, one additional write is needed to update the record specified.

$$\text{expected get disk accesses} \;=1$$

$$\text{expected update disk accesses} \;=2$$

Figure 21: The expected disk accesses required by the get and update operations for the embedded flash implementation of the linear hash described.

5. Results

5.1 Testing Environment

All tests on an Arduino platform were performed on a ATmega256 model. This has 8KB of SRAM and 256KB of flash memory [9]. To expand the amount of non-volatile memory available, a Lexar 300x 16GB Micro SD card was used in the device. Time was measured using the ion_time() function in the IondB platform which uses different time functions depending on the platform. When used on Arduino, it is a wrapper on the Arduino's millis(). For all tests, an initial size of 4, a split threshold of 85%, and 20 records per bucket were used as the values for these parameters of the linear hash.

5.2 Insert

Figure 1 shows the time to complete 5 insertions into the embedded flash implementation in milliseconds plotted against the amount of records in the table. The expected time for insertions remains constant as the table grows. The three clusters coloured differently in Figure 1 correspond to insertions where overflow buckets are created in red, insertions where splits are triggered in blue, and insertions where no additional processing are needed in green. Clearly, the insertions which require the creation of an overflow bucket are costlier than insertions where this is not necessary, and splits are costlier still.

The mean insertion time when capturing the time taken to write overflow buckets and perform splits was found to be 33.16 milliseconds. Naturally, the standard deviation was high when grouping these together, amounting to 43.56 milliseconds. Drilling down into the individual segments showed more representative results for the individual kinds of insert operations. Inserts where no splits were performed and no overflows were created had an mean time of 10.71 milliseconds with a standard deviation of 3.36 milliseconds. The insert operations which triggered the created of an overflow bucket had a mean time of 74.79 milliseconds and standard deviation of 18.59 milliseconds. The split operation took by far the longest, demonstrating an mean time of 150.10 milliseconds with a standard deviation of 27.78 milliseconds.

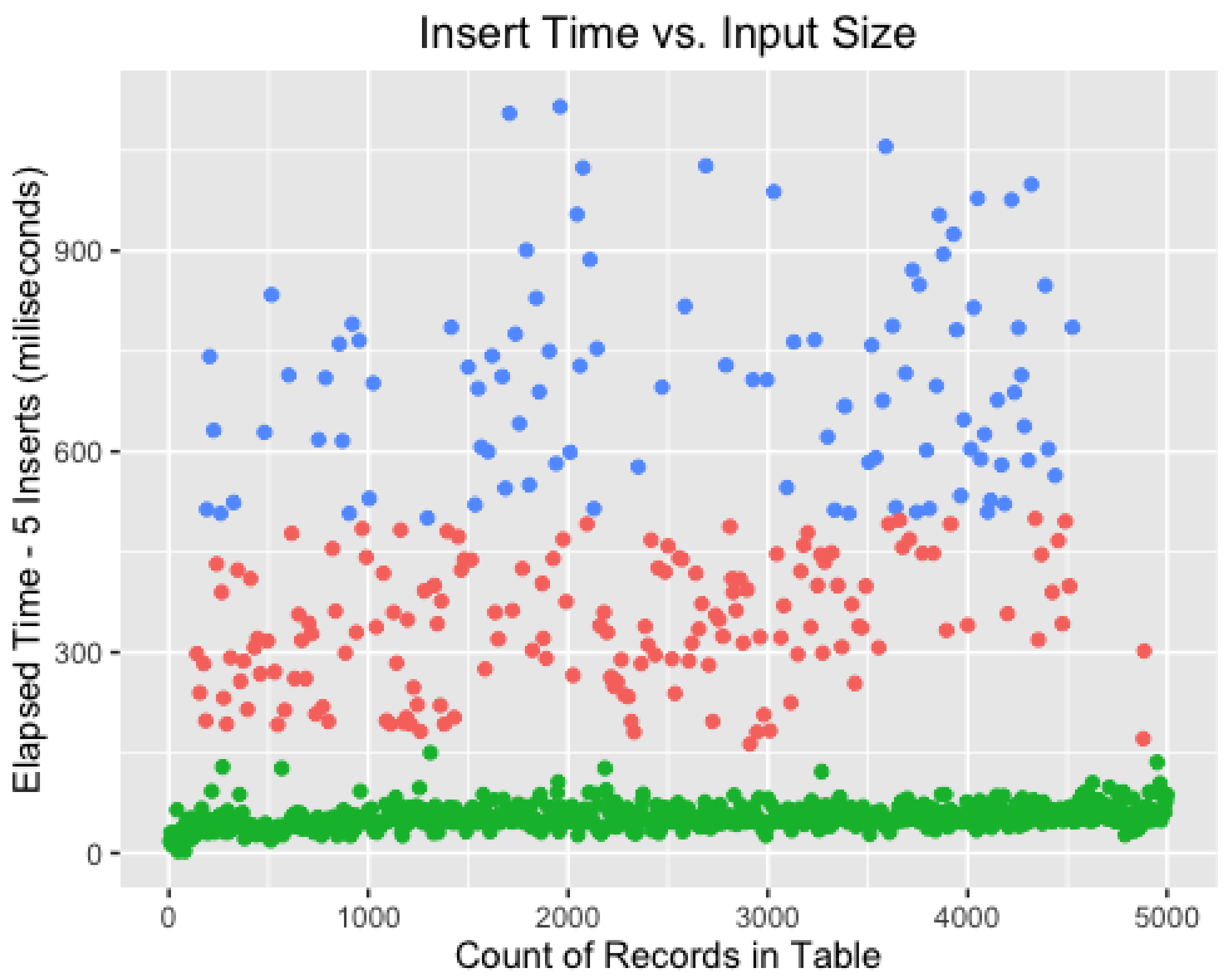

Figure 22: Visualization of the relationship between the time taken to complete a series of insertions and the input size for the embedded flash implementation of the linear hash. The input size does not affect the time taken for insertions. The blue dots correspond to insertions which trigger a split to occur, the red dots are insertions which required the creation of an overflow bucket, and the green dots required no additional processing.

5.3 Delete

The time taken in milliseconds to complete 5 delete operations is shown on the y-axis in Figure 3. Again, the x-axis denotes the amount of records in the table at the time the 5 deletes measured were performed. As discussed, the specification for deletes in IonDB requires the deletion of all records with the key specified. Consequently, the equivalent of multiple delete operations in a specification not as such must be performed. As the number of records in the table increases, the probability that a record will share a key with it gradually increases in most typical address spaces where duplicate keys are allowed. Consequently, there is an increase in the variance of delete operations as the number of records in the table increases. It is worth noting here again that this is the issue was the motivation for the eager deletions during the swap-on-delete process that was implemented and described in a previous section. The mean time for deletions was found to be 10.67 milliseconds with a standard deviation of 6.97 milliseconds.

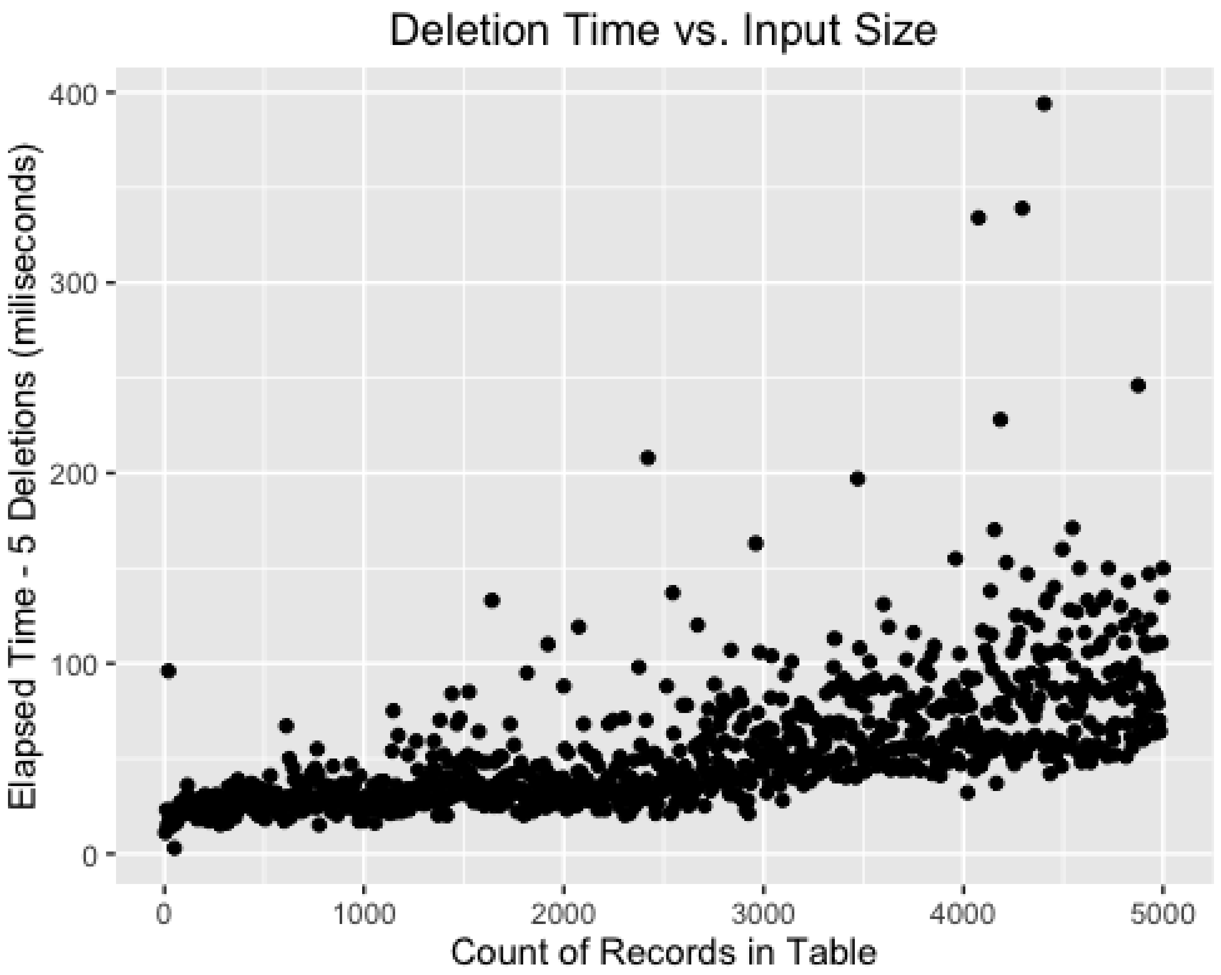

Figure 22: Visualization of the relationship between the time taken to complete a deletion and the input size for the embedded flash implementation of the linear hash.

5.4 Get

Pictured in Figure 2 is the time to complete 5 get operations. Again, the expected time taken to complete get operations remains constant as the table size grows. Several factors contribute to the variation observed. Importantly, a portion of the variation is due to the differing number of overflow buckets amongst the indexes. The mean time taken for get operations was 4.94 milliseconds with a standard deviation of 2.03 milliseconds. As the algorithm for the get and update operations are essentially the same, a thorough analysis of the update operation would be redundant and, as such, was excluded.

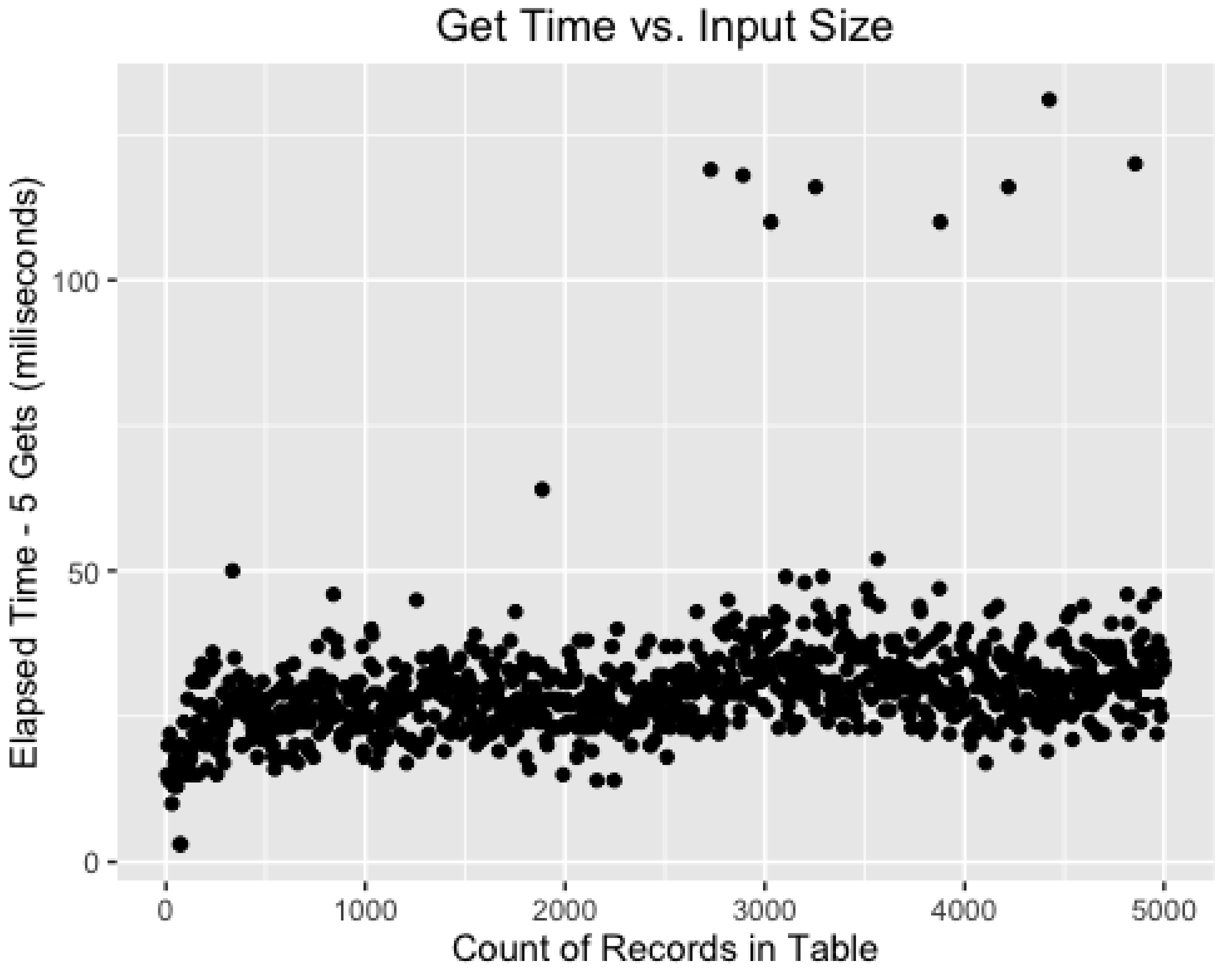

Figure 22: Visualization of the relationship between the time taken to complete a get and the input size for the embedded flash implementation of the linear hash. The input size does not affect the time taken for get operations.

5.5 Performance of Hash Function

To assess the performance of the hash function used, a total of 5042 records with randomly generated integer keys were inserted into a linear hash table. The resulting mean record count across all buckets was 18.88 records which is below the full capacity of a bucket. The mean number of overflow buckets was 0.33. Histograms at the finest grain (binwidth = 1 bucket) and a slightly coarser grain (binwidth = 10) are shown below in Figure 5 and Figure 6 respectively.

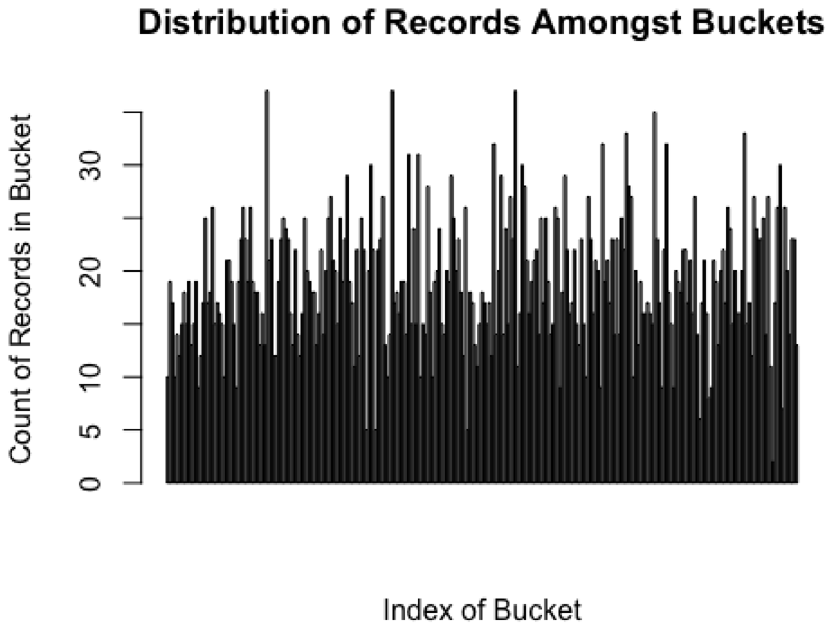

Figure 22: Visualization of the distribution of records amongst the buckets in the linear hash with approximately 5000 records in it.

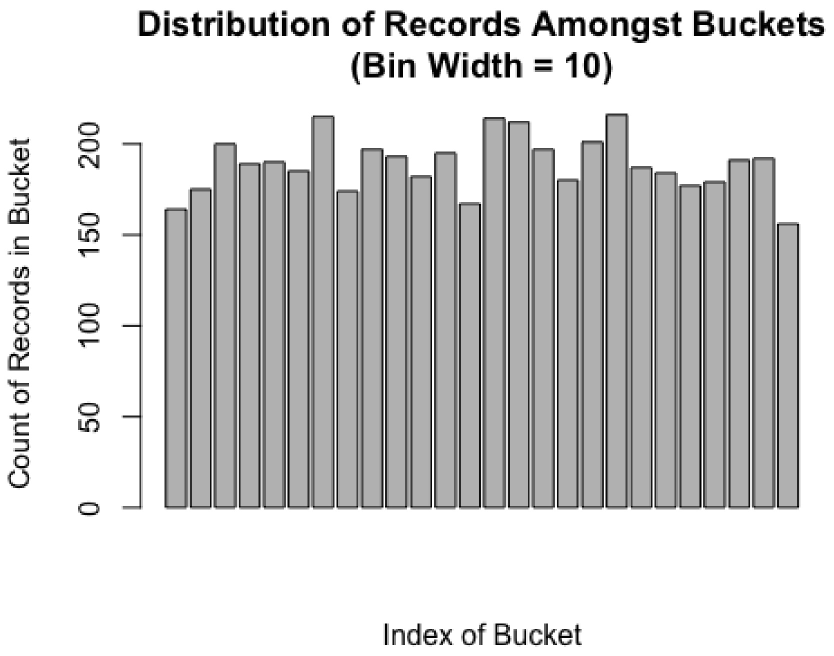

Figure 23: Visualization of the distribution of records amongst the buckets in the linear hash with approximately 5000 records in it grouped by 50 records. This demonstrates that the hash function used is not significantly biased toward any regions of the table.

5.6 Performance Comparison with Flat File

As previously discussed, IonDB currently has several different implementations which can be chosen from. A flat file is a simple data structure which can be used to implement a key-value store. An implementation of a flat file is available in the IonDB platform. To assess how the performance of the embedded flash implementation of the linear hash compared with a structure currently in IonDB, a simulation was conducted to assess performance. 2000 records were inserted into the table and time taken for gets was recorded as the table size grew. Figure 7 and Figure 8 visualize the performance of the flat file and the linear hash respectively.

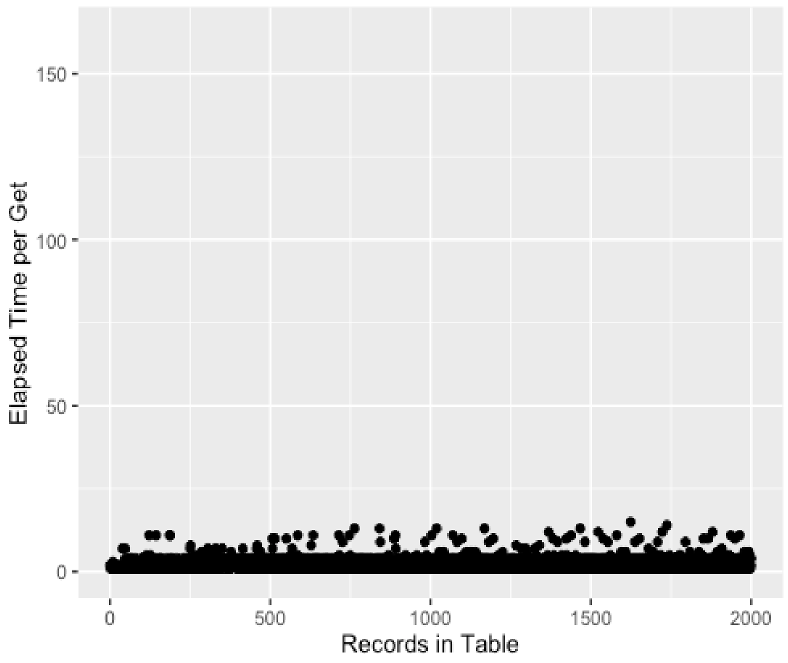

Figure 25: Visualization of the relationship between the time taken to complete a get for the embedded flash implementation of the linear hash and the input size from the performance comparison trails. The input size does not affect the time taken for get operations.

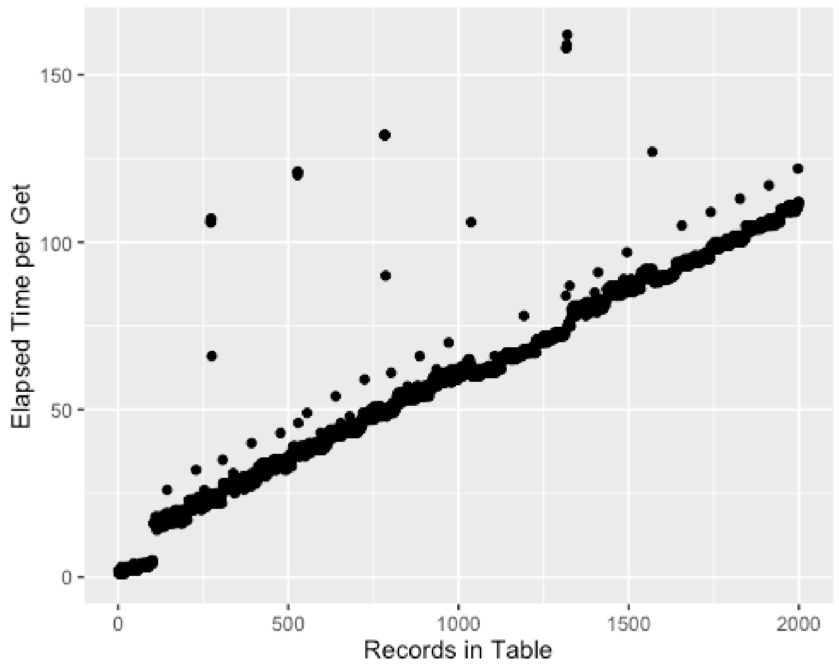

Figure 26: Visualization of the relationship between the time taken to complete a get for the flat file implementation in of the IonDB platform and the input size from the performance comparison trails. There is a clearly linear relationship between the input size and the time taken to complete the get operation.

A linear relationship between the number of records in the table is clearly demonstrated by the flat file, and a constant relationship is seen between these two variables for the linear hash. The mean times taken for get operations in the simulation conducted was 60.58 milliseconds for the flat file and 2.42 milliseconds for the linear hash. The mean is a poor basis of comparison in this case however, as the mean for the flat file will continue to increase while the linear hash will stay the same. In fact, in the last 200 observations of the data set, the mean for flat file gets rises to 104.44 milliseconds and the mean for the linear hash remains essentially the same at 2.43 milliseconds.

6. Conclusions and Future Work

The embedded flash implementation of the linear hash detailed in this work could maintain its desirable time and space complexities in the resource-constrained Arduino environment. Several implementation challenges had to be surmounted to attain this, and development highly performant data management systems for such environments was found to be a formidable task.

This implementation of the linear hash described far outperformed the flat file available in IonDB. This was especially true when the number records in the table grew large due to different computational complexities of each data structure. No other implementations currently available in IonDB can perform all the operations of the dictionary interface in a constant number of disk accesses regardless of the size of the table and remain dynamically resizeable.

While many algorithmic techniques were used, there are still areas in which the implementation presented could be improved upon. As noted in a previous section, ease of implementation and code reusability was favoured at times over sheer performance. To further improve the performance of the embedded flash implementation of the linear hash, a refactoring of the basic operations could beneficial. For example, code reuse in the linear_hash_insert() leads to an unnecessary write operation when creating an overflow bucket in the form of a call to linear_hash_update_bucket() that gets reused regardless of the insertion case encountered.

Exploration of the viability of higher levels of structuring like that used in the self-adaptive linear hash for resource-constrained microprocessors could also be a fruitful endeavour. Likewise, the development of a growth function that would enable spiral storage could also provide further gains in performance. While buffering a useful number of log records of operations in main memory in the environments like the Arduino seems like a challenging task, the benefits associated with the successful use of such techniques are well known.

The data structure presented as part of this work has already proven itself as a performance-competitive option for embedded devices. An even more performant implementation could be realized by using the embedded flash implementation of the linear hash described in this work as a foundation and could serve as a foundation for an implementation that incorporates the results of the future work described above.

References

[1] Barragan, H. (2016). *The Untold History of the Arduino*. Retrieved April 8, 2017. https://arduinohistory.github.io/

[2] Barragan, H. (2004). *Wiring: Prototypical Physical Interaction Design*. Interaction Design Institute Ivrea, Ivrea, Italy.

[3] Clemons, T., et. al. (2013). Hash in a flash: Hash tables for flash drives. *2013 IEEE International Conference on Big Data.* Retrieved April 12, 2017. http://users.eecs.northwestern.edu/~hardav/papers/2013-BigData-HashFlash-Clemons.pdf

[4] Dai, H., Neufeld, M., & Han, R. (2004). ELF: An efficient log-structured flash file system for Micro Sensor Nodes. *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems.* Pp. 176-187. doi: 10.1145/1031495.1031516

[5] Dumitru, D. (2007). *Understanding Flash SSD Performance*. Retrieved April 1, 2017. http://www.storagesearch.com/easyco-flashperformance-art.pdf

[6] Fazackerley, F., Huang, E., Douglas, G., Lawrence, R. (2015). Key-Value Store Implementations for Arduino Microcontrollers. *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering.* pp. 158-164. doi: 10.1109/CCECE.2015.7129178

[7] Larson, P., A. (1988). Dynamic hash tables. *Communications of the ACM, 31*(4), pp. 446-457. doi: 10.1145/42404.42410

[8] Litwin, W. (1980). Linear hashing: A new tool for file and table addressing. *Proceedings of the sixth international conference on Very Large Data Bases, 6,* 212-223. Retrieved April 2, 2017. http://hackthology.com/pdfs/Litwin-1980-Linear_Hashing.pdf

[9] Microchip Technology, Inc. (2017). *Arduino ATmega2560*. Retrieved April 13, 2017. http://www.microchip.com/wwwproducts/en/ATmega2560

[10] The PostgreSQL Global Development Group (2017). CREATE INDEX. *PostgreSQL 8.0.26 Documentation*. Retrieved April 27th, 2017. https://www.postgresql.org/docs/8.0/static/sql-createindex.html

[11] Pottie, G. J., & Kaiser, W. J. (2000). Wireless Integrated Network Sensors. *Communications of the ACM. 43*. pp. 51–58. doi: http://doi.acm.org/10.1145/332833.332838

[12] Rosenblum, M., Ousterhout, J. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems, 10*(1), pp. 26-52. doi: 10.1145/146941.146943

[13] Tomayko, J. (1998). *Computers in Spaceflight: The NASA Experience.* Washington, DC: National Aeronautics and Space Administration, Scientific and Technical Information Division.

[14] Toshiba America Electronics Components, Inc. (2017). *What Would You Do Without Flash Technology?* Retrieved April 12, 2017. http://www.flash25.toshiba.com/

[15] Yang, C., Jin, P., Yue, L., & Zhang, D. (2016). Self-adaptive linear hashing for solid-state drives. *2016 IEEE 32nd International Conference on Data Engineering.* doi: 10.1109/ICDE.2016.7498260