Faculty of Science and Technology
Department of Computer Science

MELT: The multidimensional key-value store performance evaluation framework

MELT: memory, energy, latency, and throughput

—

Tobias Blomli-Edvardsen

INF-3981 Master's Thesis in Computer Science - June 2017

Content

1 Abstract

Key-value stores have a very large variation in their design and implementation, while still adhering to the key-value abstraction. The available generic benchmarks cannot truly represent the performance a key-value store will have with a specific application, unless your application happens to have the exact same configuration and workloads as the benchmark. Moreover, most benchmarks only measure throughput and latency, ignoring performance metrics like energy efficiency and space efficiency.

Introducing MELT: The multidimensional key-value store evaluation framework, which can take any applications usage characteristics of a key-value store and test it on multiple different key-value store implementations with different concurrency and throughputs settings. In addition, it measures four MELT performance metrics, memory, energy, latency and throughput. With this evaluation framework the assumption is that concurrency is better than serial key-value stores in all situations. Here will be shown results that support the claim that for certain applications, with throughput demands less than 10 million operations per second, serial hopscotch implementation outperforms the concurrent Libcuckoo on most of the MELT performance metrics.

2 Introduction

2.1 The problem

Key-value stores are a widely used data structure in modern computing. It is a simple key to value mapping that offers a simple interface, easily adapted to most applications needs. This simplicity hides the large variation in implementations, and the importance of the key type and length. What datatype the key is, will to a large degree impact performance. Comparing integer based with string based key-value stores, is not an apples to apples comparison. However the same goes for the access pattern (workload), number of entries and the throughput rate (number of requests per second), not to be confused with maximum throughput which is the key-value stores, maximum number of requests that can be handled per second. Benchmarking usually only cares about the maximum throughput, even though it is likely that an application will not need that maximum throughput. What are the performance metrics at lower throughput rates? Truly evaluating a key-value store in a way that is representative for actual use cases, is a complex issue. All the interaction characteristics of applications used of a key-value store needs to be considered.

2.2 Motivation

Here is presented an evaluation framework that can take the interaction characteristics of any application's use of a key-value store and benchmark it on any key-value store, simulating different throughput rates and threads in use. Providing a full multidimensional analysis of the performance metrics of multiple key-value stores, on a specific dataset that simulate the access pattern of an application. Not only providing performance metrics like throughput and latency, but also reporting energy use and memory efficiency. It is extensible so that any data structure that uses CRUD (create, read, update and delete) operations.

It is often assumed that concurrent key-value store implementations are better than the serial ones. It is true that when measuring their maximum throughput, serial implementations cannot compete on modern hardware that focuses on parallel computing. This assumption does not consider that a lot of applications will never use a key-value store at anything close to the maximum throughput. Nor does it consider that managing concurrency is not free. It has not been shown that concurrent key-value stores are better than serial ones at lower throughput rates.

In the rapidly evolving ecosystem of different devices, which to larger and larger degrees are based on mobility and battery power, energy efficiency and power use is becoming a first class concern[1]. From HPC environments where him power use is a large part of the systems lifetime monitoring, to mobile phones where getting the battery to last the entire day is challenging with the computational demands of today.

2.3 Existing solutions

Every application is different in its workloads and in what performance metrics are the most important. Testing and evaluating multiple key-value stores is time-consuming. It is easy to forget to consider certain metrics and ensuring that the measurements are correctly implemented, and to consider all important factors is not a trivial task. This framework attempts to streamline this process in a portable way that can provide a holistic insight into the different characteristics of different key-value stores for the specific use case of an application, not a generic workload on a generic benchmark where certain interaction characteristics may differ from the application in question. The benchmarking tools like YCSB[2] provides some of this functionality, it can generate string key based workloads that can represent most applications. However, it only measures throughput and latency and its primary focus is large-scale cloud-based key-value stores and databases. It does not provide insights into all dimensions of a key-value store total spectrum of performance metrics. This evaluation framework provides a general purpose to four evaluating key-value stores.

2.4 Contributions

This paper has the following contributions:

- Here is documented the initial implementation of the MELT evaluation framework, and an analysis of this implementation.
- The benefits and drawbacks of using YCSB to generate workloads for use in this evaluation framework.
- Experiments using two theoretical application workloads, to test the benefits using this evaluation framework.
- Experiments to evaluation of concurrent versus serial key-value stores, to investigate if nonconcurrent key-value store still has a place in modern computing where multicore architectures are the norm.

3 Background

3.1 Key-value abstraction

The key-value abstraction is commonly used in computer science. It is essentially a key to value mapping where a unique key is linked to a unique value when an application uses a key-value store. Its use of the key-value store can be described as the access pattern, also called the workload.

3.2 Access pattern(workload)

The access pattern of any key-value store can be described as the percentages of the different CRUD operations. For example, an access pattern can be 50% reads, 25% inserts and 25% delete. YCSB[2] creates its traces from such a description and is derived by tracking the different CRUD operations performed on the key-value store over a duration of time. Therefore, it is in fact the average access pattern over the given duration to be precise. If the application has different access pattern at the start and end of its execution time, it can be described as two different access patterns, for its first half and second half. If this is divided into even more durations, you can accurately track applications access pattern over time. This is not commonly done when describing access patterns, instead it describes the average access pattern of the application's interaction with a key-value store for the whole duration of the applications runtime.

3.2.1 Operation failures

To be completely accurate there are more than four types of operations that can occur on key-value stores. That is operation failures, read, update and delete operations can fail because the key used has not previously been inserted, and I could theoretically be applications where these cases occur, it is however most likely very rare, and therefore insignificant. The operation failures aspect of access pattern is not used in practice.

3.2.2 Distributions

The access pattern is also defined by its distribution; distribution is the frequency of which the same keys are used. YCSB[2] defines three types of distributions:

- Uniform: All keys equally likely to be chosen.
- Zipfian: some keys are much more likely than others.
- Latest: the keys recently inserted are the most likely to be chosen.

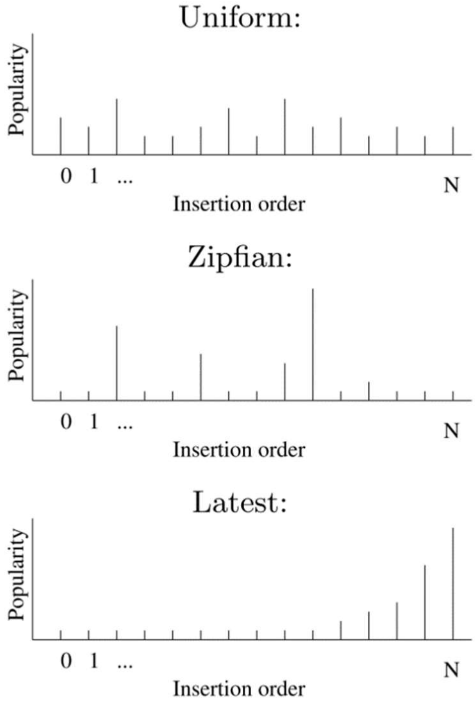


Figure 3-1 illustration of different distributions. illustration taken from[2]

3.3 Hash tables

Key-value stores can theoretically use just about any underlying data structure. However, by far the most prevalent data structure is the hash table as it in general performs the better. Tree structures are also not uncommon, for example Masstree[3]. The key-value stores in this report are all hash tables and this section will describe different concurrency schemes and hashing schemes of hash tables.

3.3.1 Concurrency schemes

In the current age of multicore architectures and parallel computing, concurrency is an important component. Hash tables can manage concurrency through a couple of different schemes that can roughly be divided into lock based and lock free implementations.

3.3.1.1 Lock based

The traditional approach to avoid potential race conditions is to create critical sections protected by locks. With this approach, the granularity of the locks is important. Each entry could have its lock or a range of entries can share a lock. This last approach is commonly called lock striping, and used in the Java's concurrent Hash map library[4]. From a performance standpoint, it is important to balance the number of locks used with the number of threads accessing it. Even though lock free solutions are gaining popularity, locks are still widely used[1][3][4][7].

3.3.1.2 Lock free

Lock free solutions avoid using locks, instead using atomic operations in specific sequences that insure that race conditions are avoided. These solutions often rely upon pointer based data structures like chaining (see section 3.3.2.1.2), combined with linked list structures like the Split ordered list[8]. The downside with lock free solutions is that they often get very complex when dealing with issues like deletion and garbage collection, there is also the lock-free resizing problem which has been the topic of these papers [8]–[10].

3.3.2 Hashing scheme

This section describes the most common hashing schemes, and highlights the core elements of each algorithm and there key differences.

3.3.2.1 Memory organisation in Hashing scheme

3.3.2.1.1 Open addressing

Open addressing schemes store the entries in sequential arrays. These arrays can be organized in one of two ways. Either with two arrays where the keys are stored in one array and the values in the other, or in a single array where the key and value are stored as a pair (See Figure 3-2). Which one performs better is dependent upon the other elements of the hashing scheme. The key difference is that with one array both key and value can be retrieved in a single memory operation, as they can both be contained within one cache line. Whereas with two arrays, key-value pair cannot be retrieved with less than two memory operations. However, in certain cases this may be beneficial, for example if the load factor is high, and the value is large. The single array solution may have to read more cache lines than the two array approach because the keys are adjacent. One array may need to read several values it does not need, but on the other hand, it may benefit from prefetching hardware.

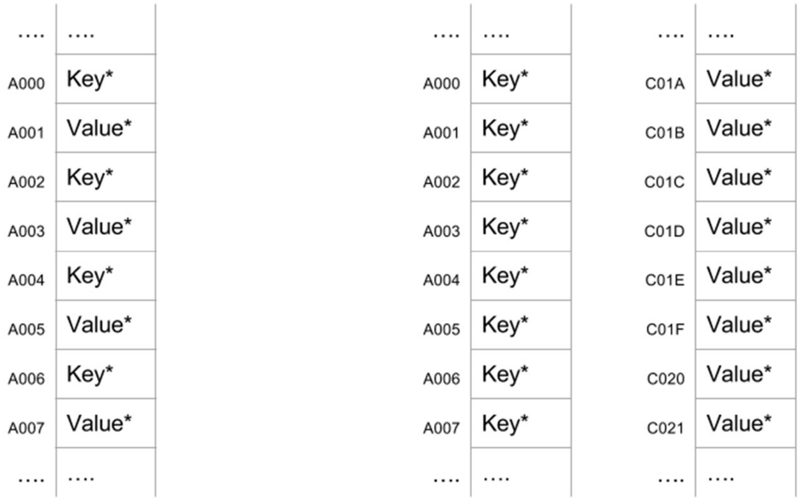


Figure 3-2 Example of memory organization in one or two arrays.

There are hashing schemes that store metadata with the key-value pair [7]. This will then typically be stored with the key in two array solutions.

3.3.2.1.2 Chaining

A different solution to memory organization is chained hashing. It uses an array of buckets were each key will uniquely identify to one bucket. Within each bucket there is a secondary data structure storing key-value pairs. This is usually a list structure but could theoretically be any structure, Figure *3-3* is an illustration of this with a simple unordered list as the data structure.

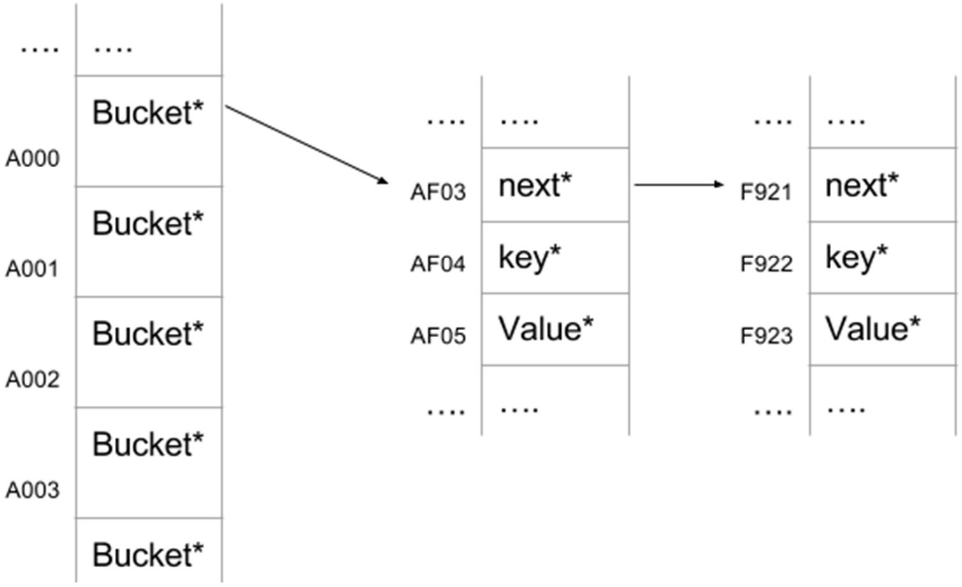


Figure 3-3 The simplest form of a hash chaining implementation. Using simple linked list approach in each bucket.

3.3.2.2 Linear Probing

Linear probing uses open addressing to organize the key-value pairs. Each entry has an optimal position within the index. This position is calculated based on its hash. A collision occurs if two keys have the same optimal position, to resolve the collision the index is scanned for the next free position. How many entries need to be scanned before a free position is found will depend on the load factor. The average number of positions that need to be scanned, can be described by the following equation[11].

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-L}\right) \qquad \text{(successful search);}$$

$$C'_n \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-L}\right)^2\right) \qquad \text{(unsuccessful search).}$$

n is the size of the index and, L is the load factor.

At a load factor of L = 0.7, it is expected that 2.17 entries will be probed to find an existing entry in the table and 6.05 probes if it is not. However, the exact details will depend on the implementation details and the hardware used. Linear probing can be said to be inherently cache oblivious[11] since it scans memory sequentially. When the hash table with linear probing reaches a load factor of 60%, it tends to cluster[12]. That is when several entries in a row get offset from their optimal position, this leads to longer lookup times.

3.3.2.2.1 Deletion

Deletion is problematic for linear probing since entries cannot simply be removed if they have entries offset from their optimal position after them. At this will breakout linear probing scans, the common solution to this is using tombstones. Tombstones indicate that this entry is deleted, but subsequent entries may need to be probed. This solves the issue but can lead to contamination[7]. As the number of tombstones increase over time more and more entries will be unnecessarily skipped, leading to longer lookup times.

The most common solution to this is to rehash the table, either partially or completely, which will remove tombstones. Another approach[12] which can delay contamination, is to check whether the next item is empty when deleting and only placing tombstones when necessary, reducing the chance of connecting clusters of entries.

3.3.2.3 Quadratic probing

Quadratic probing works on the same principal as linear probing, instead of scanning sequentially it scans quadratic. When collisions occur, it takes its current position and adds I^2 , where I is the number of probes attempted. This is the approach used by Google's SparseHash[13] and is proven to be less prone to clustering[12].

3.3.3 Chained hashing

Chained hashing uses the chained data structure. It is hard to generalize about hash chaining as the underlying data structure is very important. The focus will instead be on the newer Lock-Free Extensible Hash Table[8] the structure, and the more recent derived work that builds upon it [9][10].

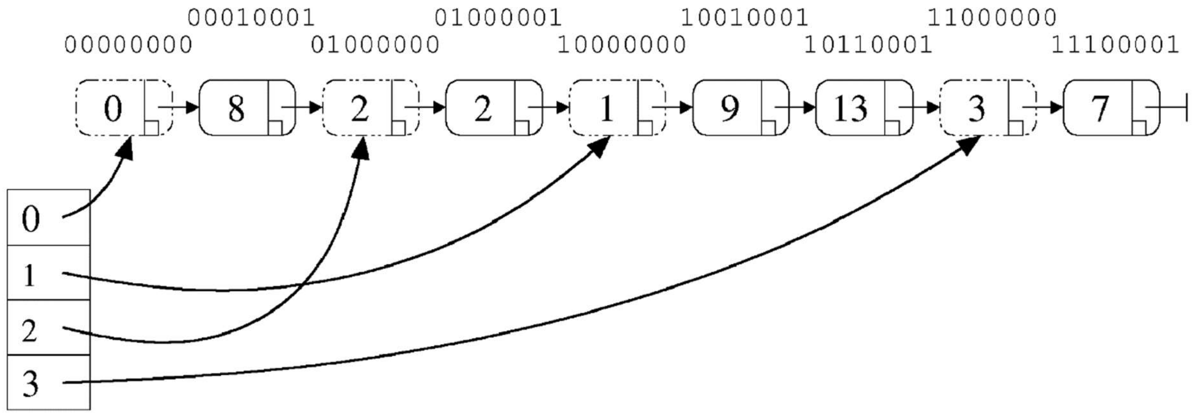


Figure 3-4 a Split ordered a hash table[8].

The key innovation of Lock-Free Extensible Hash Tables[8], is the Split-Ordered Lists that is continuous for all buckets. The buckets primarily work as an index for the list, and as this expands, more buckets pointing to the list can be added without having to change the list. The implications of this is that the lock-free resizing problem[8] is alleviated, and that the search speed can be kept low by adding buckets as the list increases in size.

Pointer chasing is still an issue as the underlying structure is still a link-list. However, link-list structures are well suited to be lock free, and is therefore a common approach for lock free hash tables.

3.3.4 Hopscotch hashing

Similar to linear probing and quadratic probing with the key difference, each entry has a bitmap of size H. This bitmap indicates which of the H -1 next entries optimally should be stored in that position. Therefore, when scanning only the positions indicated in the bitmap needs to be probed. This also means that when entries are displaced from their optimal position, they cannot be displaced more than H -1 from that position. If this is not possible, existing entry must be shifted to another position (see Figure 3-5). If this fails the hash table needs to be resized and rehashed.

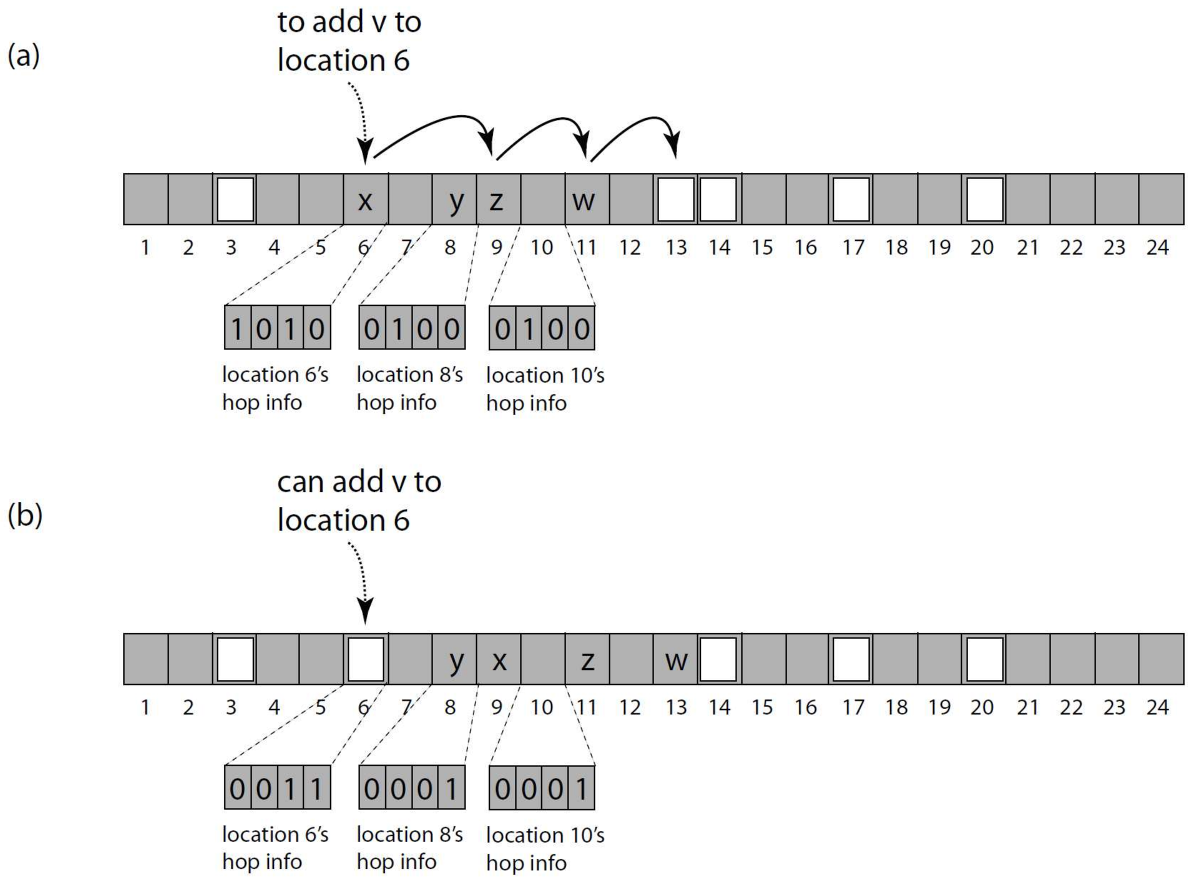


Figure 3-5 The blank entries are empty, all others contain items. Here, H is 4. In part (a), we add item v with hash value 6. A linear probe finds entry 13 is empty. Because 13 is more than 4 entries away from 6, we look for an earlier entry to swap with 13. The first place to look is H − 1 = 3 entries before, at entry 10. That entry's hop information bit-map indicates that w at entry 11 can be displaced to 13, which we do. Entry 11 is still too far from entry 6, so we examine entry 8. The hop information bit-map indicates that z at entry 9 can be moved to entry 11. Finally, x at entry is moved to entry 9. Part (b) shows the table state just before adding v." Figure and quote for[7].

3.4 Energy monitoring

The energy use of a system has over time become a first-class concern[1]. In large computing clusters the energy use has become a large part of the total monthly cost of the system. Laptops and smaller devices like smart phones are all now battery-powered, and maximizing the usage time of a charge is a priority form application and OS developers alike. To enable this hardware manufacturers, have over time made APIs that allow for measurements of power use like Intel's performance counter monitor(PCM)[14]. However, different hardware platforms have implemented different solutions using different APIs, which all require custom code to be used in application. Libraries like energymon[1], heartbeats[15] and the PAPI[16] support energy monitoring in our portable manner.

3.4.1 Cuckoo hashing

Cuckoo hashing also an open addressed hashing scheme, which used multiple hashing functions. Where each entry can be placed in one of H positions, were H's the number of different hashing functions employed. If all H positions are occupied, one of the entries in the occupied position, is moved to one of its H possible positions. Traditionally the number of hashing functions employed has been H=2, but more recent implementations have used H = 4 or greater. This is because with H = 2 performance starts decreasing at load factors greater than 50%, but with H = 4 it starts degrading at 90%[12]. However, performance generally decreases the higher the H value. Another approach to cuckoo hashing is using set-associativity, which is a hybrid open addressing solution, Where Each entry is a bucket of multiple entries. Organizing the entries in this way the number of hashing functions can be reduced to H = 2 with load factors of 90% or higher without a performance degradation[6].

3.4.2 Energymon

Energymon[1] is a lightweight cross-platform energy monitoring utility, which allows for the monitoring of energy use across any supported platform. It hides underlying variations in the different hardware platforms in the simple API.

3.4.3 Performance Application Programming Interface (PAPI)

PAPI is a large cross-platform performance monitoring utility. The API exposes performance counters hardware available found in most major microprocessors and it can monitor performance in real-time. It also has software components that can used for monitoring across the hardware and software stack. It has a primary focus on clusters and HPC environments.

3.4.4 Heartbeats

Heartbeat-simple[15], is a subset of the heartbeat[17] API, that does performance and power tracking. The larger heartbeat API is a framework for dynamic power management of applications and is developed by the carbon research group at MIT. It is used by poet for this POET[18]. So heartbeat is initially designed for more than just energy monitoring.

3.5 Yahoo! cloud serving benchmark

YCSB was originally developed by Yahoo![2] and later made open source[19]. It is a Java base framework for evaluating and comparing the performance of primarily no SQL database management systems. It currently natively supports a large amount of databases including Cassandra, Voldemort, MongoDB and DynamoDB and is designed to be extensible so that more can be added.

3.6 Core workloads

YCSB has defined six different core workloads, where E and F do not apply to a key-value abstraction and are not listed. Following is descriptions of each, quotes are all how YCSB describes these workloads

Workload A: Update heavy workload

"This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions."[19]

Workload B: Read mostly workload

"This workload has a 95/5 reads/write mix. Application example: photo tagging; add a tag is an update, but most operations are to read tags." [19]

Workload C: Read only

"This workload is 100% read. Application example: user profile cache, where profiles are constructed elsewhere (e.g., Hadoop)." [19]

Workload D: Read latest workload

"In this workload, new records are inserted, and the most recently inserted records are the most popular. Application example: user status updates; people want to read the latest." [19]

4 The evaluation problem

the issue with evaluating a key-value store implementations is that there are a set of interaction characteristics, that all constitute all the aspects of an application can use it. Even given the same key-value store implementation, variations these characteristics will impact the performance metrics. In applications unique use of a key-value store, can be can be described by six different characteristic variables:

- The size key and value.
- The access pattern.
- The access throughput.
- Number of entries.
- Number of threads used.
- Underlying hardware

4.1 Interaction characteristics

Below is described why each of these characteristics will impact the performance metrics, and therefore why one cannot do simple apples to apples comparisons when these characteristics are different.

4.1.1 Key and value

The key size and type is a very important aspect. You cannot compare the performance of two key-value store implementations. When one uses integer based keys and another fixed size the strings, the integer based key only requires one comparison to operations, while the strings would require one compare for each character. By the same logic you cannot compare implementations with different string lengths, the performance characteristics of a 16 by key versus a 32 by key are not comparable. They will at best be indicators. If the string is of variable length, this will also impact performance as each key is likely to be referenced by a pointer which could quickly lead to pointer chasing when key collisions occur.

The same goes for the value, as a blob of data of a fixed or variable size, in the overall performance metrics, will naturally be affected by the time it takes to transfer the value to and from the key-value store.

4.1.2 Access pattern

The different key-value operations have different performance costs associated with them. An insert operation is typically more expensive performance wise than a read operation. The same goes for updates and delete and the difference between them will depend upon the design and implementation used. Now the access pattern can be described as the percentage of different crud operations and their distribution (see section 0.)

4.1.3 Access throughput

Key-value stores are used in all types of applications. The key-value stores maximum throughput is mostly only interesting for high performance computing systems and

dedicated key-value stores like RAMCloud[20]. Most applications access the key-value store at some average throughput, which will usually be determined by external requests to the application or the speed at which the application processes the data stored, thereby limiting the throughput at which the key-value store is accessed.
An hypothetical example is an application which performs relatively heavy calculation on data sets stored in a key-value store. It reads data, performance calculation on the data and update or inserts a new value. If it uses 50% of the available data, its computational capacity running the calculation algorithm, and the rest of the capacity, is used to access the key-value store. It would only use 50% of the maximum throughput the key-value store could achieve on systems hardware, this assuming it's not bound by memory and buss speeds. For this application, the key-value stores performance metrics at maximum throughput are not relevant. However, the performance at 50% of maximum is highly relevant when benchmarking which key-value store implementation best fits the application.

4.1.4 Number of entries

The number of entries in a key-value store is relevant as it affects performance, most obviously if the size of the key-value stores is too big to store in memory, and secondary storage must be used. However, there are more subtle implications. It is not uncommon for key-value stores, especially hash table implementations, to increase in size by a power of two[9][6]. If the amount of entries are relatively fixed, around and amount, that is just larger than the power of two incremental resize point. The load factor will be just over 50%, where as if the amount of entries is just under the resize point, it would be closer to 100% see Figure 4-1.

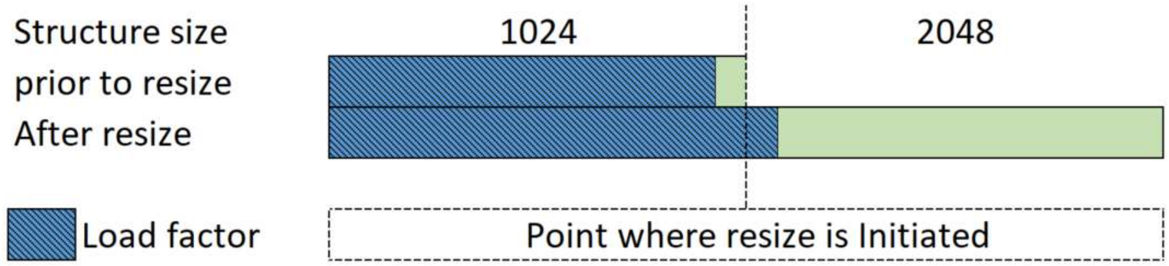


Figure 4-1 illustrates a structure that resizes when full by a power of two. It shows how the load factor is very different, even though the amount of data stored is almost the same.

However, this is a simplification as it does not consider that the load factor often is what triggers resize operations in many implementations. The load factor also affects performance[12], a load factor of 50% will likely perform better than a load factor closer to the 100%. That is performance in terms of throughput and latency. As an example Google's dense hash[13] sacrifices space efficiency for performance, and sparse hash does the opposite sacrificing performance for space efficiency.

4.1.5 Number of threads

Number of threads that simultaneously accesses a key-value store will affect performance. How many threads an application uses and how many of them access the key-value store will depend on the architecture of the application. There are two elements that determine how threads affect performance: The hardware on which the system is running, which will be discussed in more detail in the next section, and concurrency design of the key-value store. When it comes to hardware the number of cores and whether they are hyper-threaded, are likely to be the most important factor for performance when it comes to thread count. However, the concurrency design will also play a role here, particularly in how well a key-value store scales with the number of threads. In general terms there are to main variance of concurrency design lock based[3][5][7][6] and lock free implementations[5][8][10]. It is reasonable to believe that they will have different performance metrics.

4.1.6 Underlying hardware

How the system ultimately behaves is always based on the hardware. What CPU, GPU and memory is in use, and at what buss speeds they communicate. Is it an Intel x86, ARM or other architecture, how many cores do the CPU have and how are they interconnected? Which level of cash are shared between which cores? The complexity quickly becomes unmanageable; therefore, there is only one practical way to test how an application performs on different hardware. That is to test it on the hardware it will be running on. In most cases the algorithm is the most important factor and very large variations are not very likely on similar hardware systems.

5 Design

Key-value store evaluation is difficult. This details the design of a key-value store evaluation framework that can take the characteristics of any application's use of a key-value store, used CRUD Operations (Create, Read, Update, Delete) and use these characteristics to test it against multiple different key-value implementations, to determine different performance characteristics of each implementation.

5.1 Goal

The goal of this evaluation framework is to provide a tool to evaluate different key-value store implementations. Not by using static or synthetic benchmarks, but rather a benchmark based on their applications used characteristics of a key-value store, providing them with a better understanding of the performance characteristics of different key-value store implementations. This allows the evaluation of different performance trade-offs' specifically for an application, that as closely as possible reflects the real world performance of a key-value store.

5.2 Is concurrency better

It is assumed that concurrent key-value stores are the viable choice for new applications. A lot of work has been done in improving and coming up with new approaches for concurrent key-value store implementations[3], [5]–[10], [20], [21]. In this work, the performance metrics that is optimized for is maximum throughput, and in some instances latency, particularly for "cloud" or distributed key-value stores, where latency is a much larger problem than on local undistributed systems. However, for desktop, smart phones, and other small and mobile devices, maximum throughput might not be the key concern. Other metrics might be equally important, metrics like energy efficiency and space efficiency.

- My hypothesize is that depending on applications' throughput demand, there can exist a point, at which nonconcurrent key-value store outperforms a concurrent key-value store on some or all performance metrics.

The reasoning behind this hypothesis is that concurrency comes with extra overhead. Overhead in synchronization between threads, lock and lock free concurrent implementations. All rely on costlier atomic compare and swap operation as their fundamental building block, even though modern CPU architectures all have to rely on multiple cores with multiple threads. It is not thereby certain that the undoubted performance benefits this provides in high throughput systems, also applies for applications with a lower throughput need.

5.3 Evaluation benchmark design

Most of these input characteristics are assumed to be relatively constant for most applications. Even so, the throughput rate and the number of threads are the most dynamic of these characteristics and the ones that can easiest be modified to fit the applications needs. The framework will therefore evaluate, keeping the other interaction characteristics constant, while varying the number of threads and the throughput. Each possible variation of these variables constitutes a unique configuration, and each unique configuration has three different phases. The flow of the evaluation framework is easiest list described through pseudocode as seen below.

```
//the range of threads to be tested
for Threads in ThreadsRange {
    // the range of throughput rates to tested
    for throughput in ThroughputRange {
        // number of samples take for each unique configuration of threads and throughput
        for sample in sampleRange {
            //phase one measures the idle energy of the system
            phase one : idle
            //phase two load the key-value store and measures the process
            phase two : load
            // phase three runs the operations in the trace for the test duration
            phase three : run
        }
    // stops testing if the maximum throughput is achieved. If
    if throughput target not achieved
        break
    }
}
```

To get the most representative results the tests need to run for a significant amount of time. This hides any in precision in the measurement results of the hardware. The data set, should be large enough to ensure that the are enough operations to run for the entire test duration. Ideally up to several minutes.

5.3.1 Evaluation phases

The three faces evaluate different parts of the workload and system. The key is initialized prior to phase 1 and deleted after phase 3, to ensure the different samples cannot affect each other.

5.3.1.1 Phase 1 idle

Phase 1 measures the idle energy use of the system. This provides the baseline power use of the system. If the idle energy use is not constant during the evaluation, it can indicate that other processes might be running.

5.3.1.2 Phase 2 load

This phase pre-loads the key-value store at maximum throughput, measures the energy and time used and at regular intervals measures space efficiency.

5.3.1.3 Phase 3 run

Runs the operations based on the access pattern evenly at the throughput specified for the specified time duration, during which it measure time, energy and latency used, and the space efficiency at regular intervals.

5.4 Performance metrics specification

- Latency
    - The time it takes for a single operation to complete, for all the individual CRUD operations, described as percentiles.
- Energy
    - The energy in joules, measured as number of joules over time duration.
- Throughput
    - The amount of operations performed over time duration, not specified to individual CRUD operations.
- Space efficiency
    - The percentage of total amount of memory used, Divided by the total size of all key-value pair entries in the store. This differs from the load factor in that it includes all the size of the data structure itself, see definition below.
        - $\frac{Total\ size\ of\ all\ Key-Value\ pair\ entries\ in\ store}{Total\ memmory\ use\ of\ Key-value\ store} = Space\ efficiancy$

5.5 Extensibility

The evaluation framework needs to be extensible to support any key-value store implementation that support CRUD operations and it needs to do this dynamically enough to support different configurations of the same key-value store implementations. Many key-value store implementations allow for customizations like choosing which memory allocator and hash function to use which of course will impact performance. There are also more fine-grained settings that are unique to each implementation. Libcuckoo[22], for example, allows configurations on the compiler level of the number of slots per bucket, the initial size, the lock granularity and the minimum load factor. For most applications, this type of fine-grained union is necessary, but specialized applications might have need to fine-tune their key-value store and the evaluation framework should be flexible enough to support this.

5.6 Results evaluation

The extensive result output this evaluation framework will produce, leads to a challenge in parsing and analysing the data. However, by taking a specific use case and testing it by varying the throughput and the number of threads used, it should be possible to create an understanding of how they interact and how they impact performance metrics, for data specific use cases.

6 Analysis

Implementing the design for this evaluation framework has three main parts. The first part is taking the access pattern and generating a trace which can be tested by the evaluation framework. The second part is using the trace to run the benchmark, and measure all the performance metrics at different throughput rates and with different number of threads. The last part is taking the results and parsing it in such a way that it can be useful for the end-user.

6.1 Part 1. Access pattern

The trace is the access pattern described as a sequence of operations. In this implementation it is assumed that the access pattern of the application to be benchmarked is known. There are two viable options to choose from, either make a trace generation tool from scratch or use existing solutions. In this case, the existing solution is Yahoo's cloud serving benchmark (YCSB) which is a widely used benchmarking tool for database systems. For implementation of this framework, YCSB is used. The reasoning for this is detailed below.

6.1.1 Trace generator

Making a trace generator that generates random keys and values, is not very challenging and would allow for the customization of the key type and length. However, supporting different usage distributions is more challenging it would be more time-consuming.

6.1.2 YCSB

The YCSB benchmark can be used to generate a trace based on an access pattern. The YCSB benchmark supports a wider range of database options, but can be configured to support the key-value abstraction. However, it does not natively support delete operations, but support for it can be added. YCSB also provides some core workloads that are meant to be reflective of some use cases (see section 0).
The YCSB trace file is generated from the following inputs

- Percentage of read operations
- Percentage of insert operations
- Percentage of update operations
- Number of records (for preloading)
- Number of operations
- Usage distribution

Two trace files are generated, a "load" and a "run" file. The "load" file is for the pre-loading stage and it contains only insert operations. It contains the number of records specified to be preloaded into the key-value store prior to the benchmark. The "run" file contains all the operations specified by read insert and update operations as dictated by the percentages. The YCSB benchmark ensures that read and insert operations are only performed on keys already inserted. It also supports different usage distributions. YCSB biggest drawback is that it does not support multiple types of keys and lengths and does not natively support delete operations. However, its core workloads are the ones that will be used in the experiments, and it does at this point in time suit the needs of the operation framework. Except for inability to change key type and length, it supports all the interaction characteristics the framework needs.

6.2 Part 2. Throughput rate

The benchmark is designed around the concept of varying the throughput rate and the number of threads. Of these elements, controlling the throughput rate is the most challenging. The problem is evenly distributed the throughput over time duration. Naively running all the operations to be performed within the second to completion, and then sleeping for the rest of the second. It means you have run maximum throughput early part of the duration and then nothing for the last period of the duration see Figure 6-1.

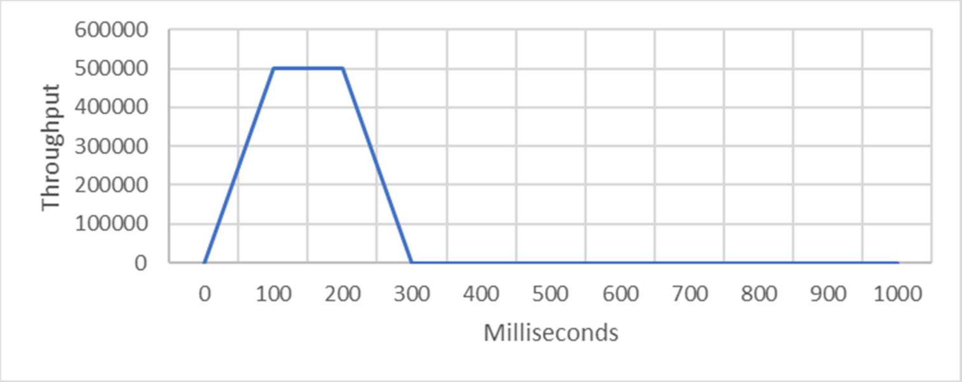


Figure 6-1 Illustration of a one second test duration, with an average throughput target of 1 million operations per second, on a system that can handle a maximum throughput of 500,000 operations per second.

In theory, you could get an even distribution if the thread slept a small amount after each operation. However, the amount of kernel calls is prohibitive and would in themselves skew the measured performance. The goal is to simulate an even throughput throughout the test duration.

6.2.1 Intervals

Dividing the total test duration up into small intervals that perform the number of operations that would on average have been performed in that time duration of the interval, and then sleeping for the duration of the interval. The proportion of time running versus time sleeping will depend on the throughput rate.

	Time ->					
Interval	1		2		3	
State	run	sleep	run	sleep	run	sleep

Figure 6-2 Illustration of three consecutive intervals, each interval will be a fixed time duration.

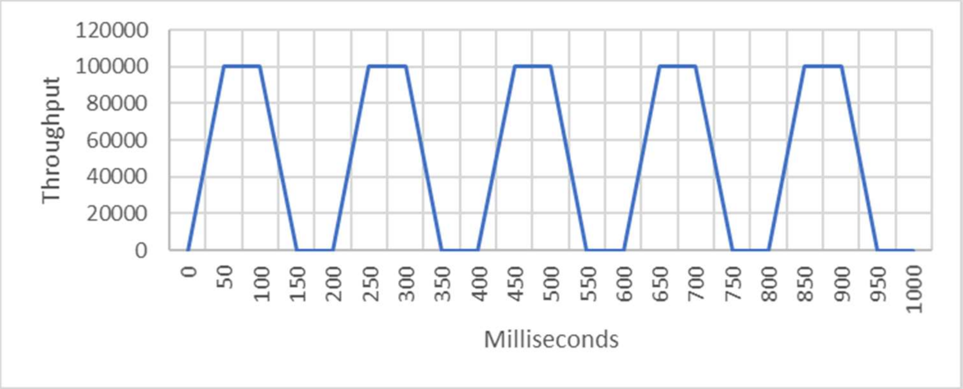
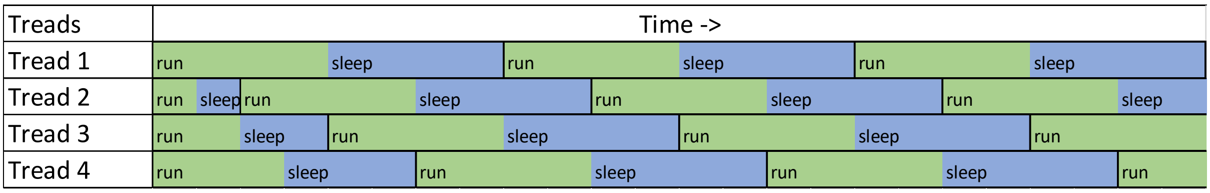


Figure 6-3 Illustration of a one second test duration divided into five intervals, with an average throughput of 1 million operations per second.

6.2.2 Interval offsets



<table>
<tr><th>Treads</th><th colspan="24">Time -></th></tr>
<tr><td>Tread 1</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td></tr>
<tr><td>Tread 2</td><td colspan="1">run</td><td colspan="1">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="2">sleep</td></tr>
<tr><td>Tread 3</td><td colspan="2">run</td><td colspan="2">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td></tr>
<tr><td>Tread 4</td><td colspan="3">run</td><td colspan="3">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="4">run</td><td colspan="4">sleep</td><td colspan="2">run</td></tr>
</table>

Figure 6-4 Illustration of multiple threads with offset intervals

Intervals mitigate the issues with throughput. However, when multiple threads run at the same intervals, they will all access the key-value store at the start of the interval, a situation that is unlikely to occur in the actual application. To mitigate this and achieve an as even as possible throughput throughout the test duration, each thread is offset slightly form each other so that their intervals do not stop and start at the same time. As illustrated by Figure 6-4. The first interval for every thread will have a different duration but the remaining intervals are of fixed length. In theory, this will make the throughput rate as even if as possible throughout the duration of the test. As illustrated by Figure 6-5, this is the technique which will be used to control throughput in the implementation of the evaluation framework.

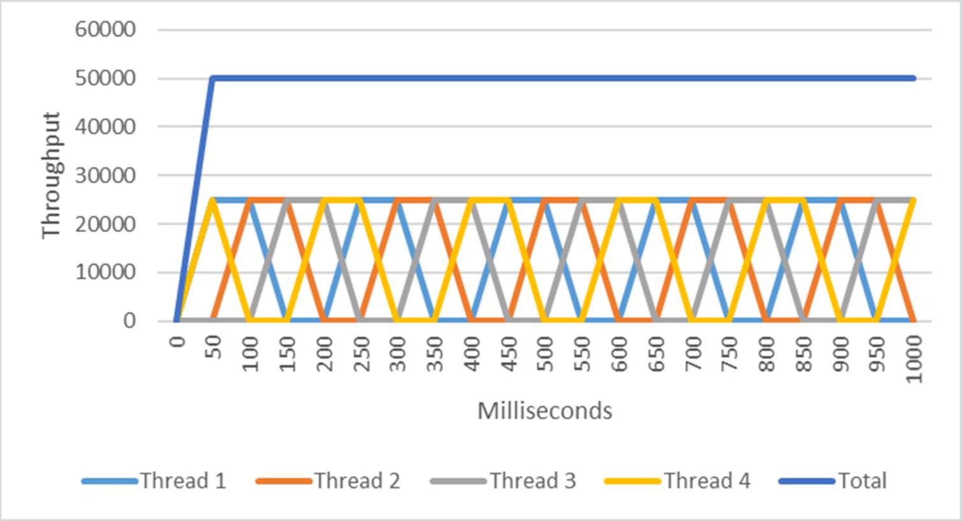


Figure 6-5 Illustration of four threads with offset intervals, running up throughput of 1 million operations per second.

6.2.3 Power measurements

Measuring and collecting the performance metrics, depend upon the ability to measure time and energy use. Time measurement is well understood and supported on all platforms and programming languages. Energy measurements, however, does not have the same native support, each platform if it supports energy measurements as a unique API of providing energy measurements. There are three candidates that support energy monitoring heartsbeat-simple[15], POPI[16] and energymon[1] all described in section 3.3. Of these POPI and energymon are the most viable. However, POPI is a general performance monitoring framework primarily focused on high-performance computing, whereas energymon is a simple and portable energy monitoring system which is simple to implement and is portable across multiple platforms which is exactly what is needed for this evaluation framework.

6.3 Part 3. Results handling

As this framework will measure all the different performance metrics, and as there might be correlations and interesting observations made across all these metrics, no more than the necessary calculations are performed. As far as possible all the raw observation data will be output in the result files. That means that large results output will have to be processed afterwards in a spreadsheet application.

7 Implementation

The evaluation framework takes as its input a two binary trace files which is generated from a YCSB trace file by a small utility application implemented specifically for this purpose. The entire trace gets loaded into memory prior to running the benchmark. As the different benchmark samples complete they are outputted to a results file. The implementation details of all these steps will be detailed below.
The design specifies measuring space efficiency, but measurement of this performance metric is not implemented in this version of the evaluation framework.

7.1 Language and library details

The evaluation framework is implemented in C and C++ to ensure maximum portability and performance. All concurrency code is implemented using pthreads. In addition

- All statistical calculations are performed with the GNU scientific library(GSL)[23].
- All calculations and conversion of time datatypes are using a subset of the csoft of general-purpose library[24].

7.2 Delete operation

The YCSB core workloads will be used for testing and experiments, and they do not use delete operations. Delete support has not been added to YCSB, however the entire evaluation framework has added delete support as far as practically possible, and deletion support can be added at a later date with minor modifications.

7.3 Trace file preparation

There are two trace files generated by the YCSB is the framework a "load" and a "run" file. The trace file generated contains a lot of unnecessary characters, so a small binary trace generator utility was made to generate a smaller binary trace only containing the necessary information. Header with metadata followed by all the key-value entries. This is done to minimize the memory and CPU footprint of the evaluation framework. The flat structure can be loaded directly into the evaluation framework without any pre-processing, and the file size of the YCSB core workloads traces is reduced by 27% and 54% approximately.

Header generated by the	**Type**	**Key**	**Value**	**Type**	**Key**	**Type**	**Key**	**Value**	**Type**	**Key**
Struct	insert	key a	value a	read	key a	update	key a	value b	delete	key a

Figure 7-1 Binary trace file structure, each entry has an operation type and key field. The value is only set for insert and update operation types.

The header is a C struct containing most importantly the number of key-value entries and their relative size in the binary trace. In addition, it also contains the relevant trace configuration parameters. The length between entries is not fixed, an operation with a value is larger than an operation without the value. The keys produced by YCSB is of the format "USER123781857088687" where the number of numbers after the "USER" string can vary from 5 to 20[19]. During testing the observed variation was between 15 and 20. Therefore, the binary trace generator sets a fixed length key size, padding where necessary to achieve this. The libcuckoo[22] key-value store had issues with variable length keys.

7.4 Evaluation framework

The evaluation framework loads the binary trace files (load, run) and stores them in memory. The benchmark is then run in a series of loops as described in the section 0. Each unique configuration runs a number of samples, each sample consists of three phases. Each of the phases are described in detail in sections below. In addition, how all performance metrics are measured and calculated.

To make it possible for different key-value store implementations to be tested, an interface layer is employed. This interface can be configured with any library that conforms to CRUD operations. This layer of abstraction will be referred to as the interface. Details of the interface and the different implemented interfaces will be discussed in section 7.6.

7.4.1 Configuration parameters

The table describes each configuration parameter the framework supports.

Configuration variable	**Description**
Test duration*	The maximum amount in seconds the to execute.
Idle duration	The amount of seconds the background power measurements lasts
Interval duration	The duration of each interval in milliseconds.
Starting throughput rate	The lowest throughput tested for
Throughput increase rates	The increment at which throughput is increased for each new configuration.
Maximum throughput rate	The maximum throughput tested for.
Number of samples per configuration	Number of samples collected for each unique configuration.
Minimum number of threads*	The lowest number of threads tested with.
Maximum number of threads*	The highest number of threads tested with.
Latency sampling interval	The number of intervals between each sampling of latency.

The variables marked with a "*" are currently configurable through arguments to the framework. The rest are variables in the code, the intent was to create a configuration file for all the variables.

7.4.2 Time and sleep measurements

All measurements of time are done using the "clock_gettime" function defined by POSIX[25], using the clock ID "CLOCK_MONOTONIC" which is not subject to change during the running of the benchmark as the “CLOCK_REALTIME” to ensure maximum possible accuracy. All sleep calls are done using “clock_nanosleep” function define by POSIX[25], which is the highest resolution sleep function available. All references to measurement of time and sleep are implemented using these functions. Both provide a resolution in nanoseconds, however the accuracy is limited by kernel implementation and the CPU model architecture.

7.4.3 Energy measurements

All energy measurements are done using the energymon library[26][1]. Energymon allows for the sampling of the number of micro joules used since its initialization. All measurements of energy are done by sampling the energy used before and after a phase as completed. The difference between the samples is the amount of energy used, the main thread and not individual worker threads do the energy measurements.

7.5 Configuration and samples

The first configuration is set to the minimum number of threads specified and the minimum throughput specified. For each configuration, the set number of samples are executed. If the target throughput is not achieved or the maximum throughput is achieved, the thread number is incremented and the throughput is reset to its minimum value. This continues for the range of threads specified by the configuration parameters. See pseudocode in section 0.

7.5.1 Samples

Prior to the execution of a sample, a newly initialized instance of the key-value store interface is created. After the sample has completed, all entries in the interface instance are deleted and the instances self is deallocated. This is done using the initializes and destroy function of the interface, see section 7.6.

7.5.2 Phase synchronization and measurements

Apart from phase 1 the execution of a phase relies on multiple threads. It is organized by one main thread that manages the threads that execute the traces. The execution of a trace is started when the main threads creates the number of threads which is set to execute the trace in the current configuration. Each thread initializes and waits on a barrier; the main thread also waits on this barrier. Therefore, when all threads have reached the barrier the trace is ready to be executed. All threads are released from the barrier; the main thread gets the starting time of the execution and sets it in a global variable accessible to all threads. The remaining threads immediately re-enter the barrier. When the main thread has set the global starting time it enters the same barrier triggering its release. Now all the threads can execute the trace, with the starting times set by the main thread. The main thread then enters the barrier waiting for the remaining threads to complete execution of the trace. As the remaining threads complete execution, they enter the barrier. When all threads have entered the barrier the phase has completed. The main thread has measured time and energy used for the duration of the execution.

- To do add figur

7.5.3 Phase 1: idle energy

This phase measures the idle energy of the system when the framework is not doing any work by measuring the energy used over the specified idle duration, in which the main thread is sleeping. During this duration, no other threads are initialized or active.

7.5.4 Phase 2: preloading

This phase preloads the key-value store with the key-value entries in the load trace. The records are loaded by the active threads with no limitations on throughput, measuring the time and energy used across the insertion period.

Each time the number of threads in a configuration increases, the number of records in the load trace is divided among the threads in the configuration. If the number of records is not dividable by the number of threads, the remainder is divided among a subset of the threads. Therefore, the threads may not have the exact same number of records to load, but it can only vary by one.

7.5.5 Phase 3: execution

Execution is initialized as described in detail in section (7.5.2). When execution has started, each thread uses the global start time, calculate its offset based on its thread ID and executes an initial shortened interval to initiate the offset (see Figure 6-5). Each thread continues to execute intervals till all operations have been executed or the tests duration has passed, which is triggered by a signal sent by at timer thread initialized by the main thread. This signal prompts any sleeping thread to wake, and any operation in progress will be completed before the thread ceases execution.

7.5.5.1 Interval target throughput

Trace is executed in intervals, each interval lasts a fixed duration defined by the configuration. Each interval has a target number of operations to be executed. This number is calculated from the throughput target. Taking into account number of intervals per second and the number of threads, the target number of operation is set so that the overall throughput per second target will be reached. Due to the conversion from floating-point numbers to integers, this number can be slightly lower than the set target throughput.

7.5.5.2 Sleep intervals

Each sleep interval is performed using the "clock_nanosleep" function as mentioned in section 7.4.2. The interval for which to sleep is not calculated from the point in time where all the intervals target operations are completed, but rather as a multiple of the global start time and the interval duration. This is possible using the "TIMER_ABSTIME" functionality of "clock_nanosleep" [25]. It sets the time the thread should wake rather than how long it should sleep, and by using the global start time as a reference, the intervals will not drift relative to each other due to timing imprecision. However, an estimate of how long a thread will sleep is calculated and added to sum of the total time this thread has slept during execution.

7.5.5.3 Iterating through the trace

Iteration through the trace is linear, the operation type is checked and the subsequent key and value depending on operation type, is executed. Execution continues by checking the next operation type.

7.5.5.4 Latency sampling

The configuration sets how often a latency sample should be collected. So, if it is set to three, a latency sample will be collected every third interval excluding the initial interval. Latency is measured by timing a single operation of the intervals target operations. Which operation type that is sampled will be random, but each type is stored in separate arrays. This does however mean that if 5% of the operations are insert on average, only 5% of the total number of samples are from insert operations.

7.5.6 Maximum throughput criteria

After all samples in the current configuration has finished, the amount of time all the threads have slept is summed up. If none of the threads has slept the maximum throughput is reached, and testing with the current number of threads end. This is because if threads have not slept in any interval, it has not reached its target interval throughput in any interval. Inherently this means it has reached the maximum throughput at the current number of threads.

7.6 Interfaces

The evaluation framework needs to be extensible. Therefore an interface layer is added between each key-value store implementation and the evaluation framework. This is implemented using an interface header file. This header file provides the following functions:

- Initialize
- Destroy
- Read
- Insert
- Update
- Delete

In addition to the basic CRUD operations, there is an initialize and destroy function. They are the method used before and after a sample execute. The initialize function initializes the specific key-value store implementation, in accordance with the API of that implementation. The same goes for the destroy function, using the key-value store implementations API the key-value store is cleared of all entries. Then its memory structure is deallocated. This ensures each sample uses an identical key-value store interface, as discussed in section 7.5.1.

The key-value store implementations must be implemented to perform their equivalent API calls for each of these six functions. In separate files, and by using pre-compilation definitions, the evaluation framework is compiled using one of the key-value store interfaces. The interfaces are all implemented using default settings, there configuration is intentionally customized as little as possible so that their default performance is reflected. It is not feasible to optimize each configuration as there were too many variables to configure.

7.6.1 Key-value entries

The type of key-value entry needs to be individually implemented in each interface. All the interfaces here are implemented with the same key and value type. Both key and value are character arrays of a size set by pre-compilation definitions. The key-value entries also need to have defined hashing function. And the hashing function used on all interfaces is CityHash[27].

7.6.2 Libcuckoo

Libcuckoo library[22] used was developed by the original offers concurrent cuckoo hashing papers [6][28]. And they refer to this library “this source code is now the definitive reference.”[22]. The C++ and the C port version were both implemented, but only the C++ version is used. Whenever Libcuckoo is reference, it is the c++ implementation version that is referred to.

7.6.3 Google Sparse Hash (and Denes)

Google sparseHash is a library developed by Google and later made open sourced[13]. The library contains two different versions, the sparse and the dense version. Both are implemented.

7.6.4 Hopscotch

The hopscotch library[29] used is a single threaded implementation, based on the hopscotch algorithm[7].

7.6.5 Unordered map

The concurrent map implementation is the standard C++ and unordered map are also.

7.6.6 Dummy

There are two dummy implementations, one which imposes a fixed delay and one which returns immediately upon being called. They have primarily been used for debugging, but the one that does not have a fixed delay can be used to benchmark the evaluation framework itself. Since it does no actual work, it can provide insights in to the performance metrics of the framework.

7.7 Results handling

After each sample completes, the output is written to a CSV file. The list below details all the data outputted:

- General sample information
    - ♦ Name of interface used.
    - ♦ Name of the workload used.
    - ♦ Configuration ID, the unique number of threads and target throughput of the system.

  - Number of threads used.
- Phase 1 results - Idle
  - The baseline idle energy use of system.
- Phase 2 results - preloading
  - Total number of records to preload (number of entries in the YCSB load trace).
  - Total time used in seconds preloading the key-value store interface.
  - Total energy used preloading key-value store interface.
  - Operations per second while preloading.
  - Joules per operation while preloading.
- Phase 3 results – execution
  - Total number of operations (number of entries in that YCSB run trace).
  - Total number of operations executed.
  - Target throughput rate
  - Adjusted target throughput rate (target throughput rate when divided across intervals and threads section 7.5.5.1).
  - Target duration of the execution (the configured duration of the execution).
  - Measured runtime of the execution (as measured by the main thread).
  - Average runtime per thread (the average runtime measured by all threads)
  - Average time slept per thread.
  - Total energy used over execution runtime.
  - Operations per second
  - Joules per operation (due to bug this is incorrectly calculated)
- Latency measurements, all the following values are individually listed for each operation type (read, insert and update), but for simplicity they are all referred to here under the single name *operation type*.
  - *Operation type* latency samples collected.
  - *Operation type* latency mean.
  - *Operation type* latency median.
  - *Operation type* latency 10.0th percentile - latency 90.0th percentile. (In 10th percentile increments by default, is configurable through pre-compilation definitions)

To maximize the accuracy of the results which use floating-point datatypes, their results is outputted as exponential numbers instead of decimal point numbers. This reduces the loss of accuracy to a minimum.

8 Experiments

The experiments are done with the YCSB core workloads, specifically A, B, C and D. These workloads were used with all the implemented interfaces executing them with every relevant combination of number of threads and throughput targets. Test machine specifications was a Lenovo ThinkStation P500 running a Intel® Xeon® Processor E5-1603 v3 @ 2.80ghz With 16 GB Ram at 1866 MHz. Running Ubuntu version 16.04 LTS. During testing the machine was disconnected from the network to avoid any background operations being triggered by a remote connection.
The evaluation framework had the following configuration:

Configuration variable	Value
Test duration	30 seconds.
Idle duration	5 seconds.
Interval duration	25 ms.
Starting throughput rate	1 million operations per second.
Throughput increase rates	1 million operations per second.
Maximum throughput rate	No upper limit.
Number of samples per configuration	Five samples.
Minimum number of threads	1.
Maximum number of threads	8 (when the interface was concurrent).
Latency sampling interval	1 per interval.

9 Results

All the relevant results from the experiments will be described in this section. First results can validate the evaluation framework implementation. A brief mentioning of the preload phase results, and then all the results from the execution phase are presented, as well as two theoretical applications use cases.

9.1 Implementation validation

9.1.1 Execution time

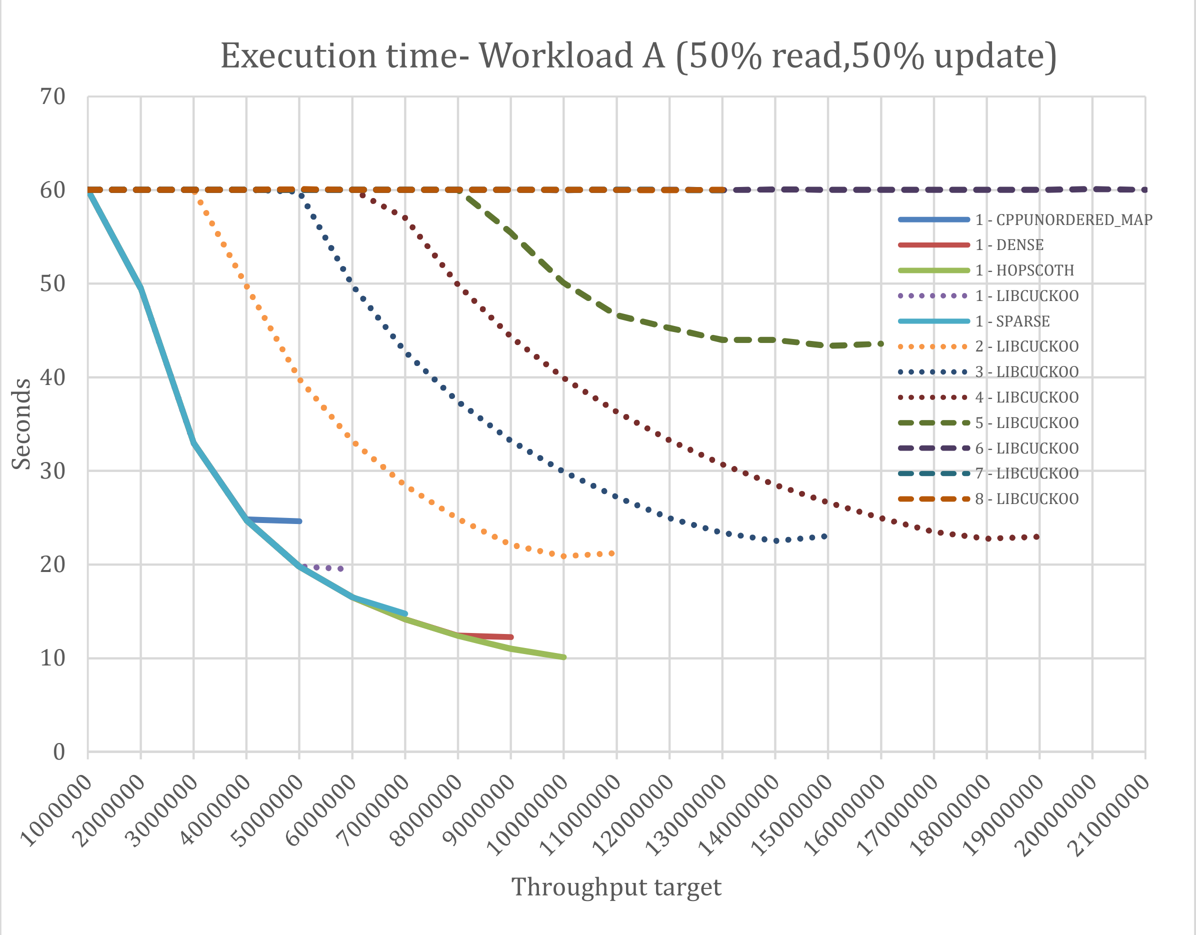


Figure 9-1 This graph shows the execution time at different throughput target rates. The number prior to the interface name is the number of threads tested with.

This graph shows the average running time of the different interfaces. As target throughput increases, the time used to complete the one million entries drops rapidly for all single threaded interfaces. Each thread used executes 1 million operations each, so the execution time increases as the number of threads increases. As an example, the fourth threaded Libcuckoo executes 4 million operations. The lower execution times for especially single threaded interfaces has implications for the measuring of latency, as will be shown in section 9.1.3.

9.1.2 Throughput control

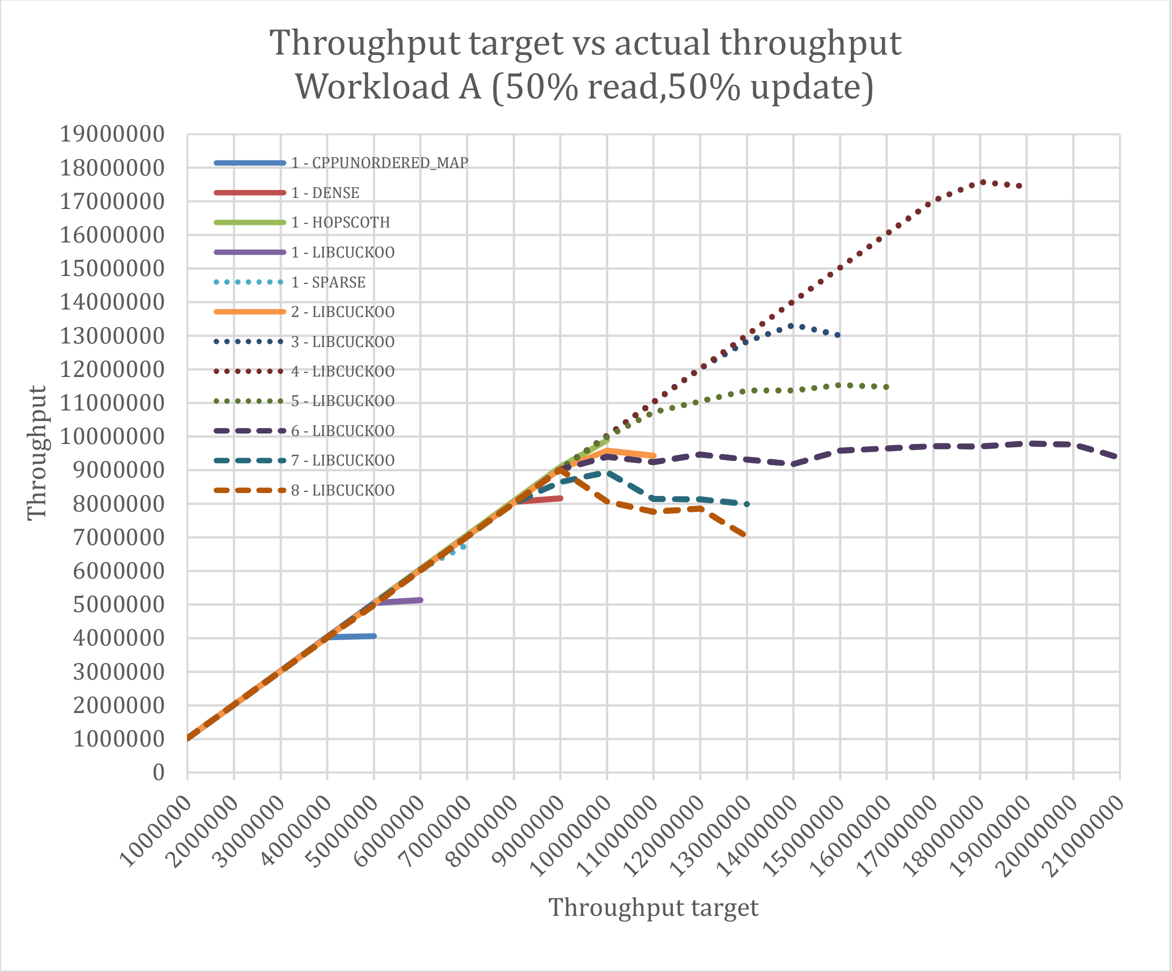


Figure 9-2 This graph shows the actual throughput at different throughput target rates. The number prior to the interface name is the number of threads tested with.

As seen in Figure 9-2, the throughput and the throughput target, matches, well up to the point where interfaces approaches its maximum throughput. Here they flatten out and end as the maximum throughput is reached, except for interfaces running with more than four threads, which is the number of cores the CPU has. The maximum throughput criteria fails to end execution, so the threads continued to run, even though they are not meeting the throughput targets.

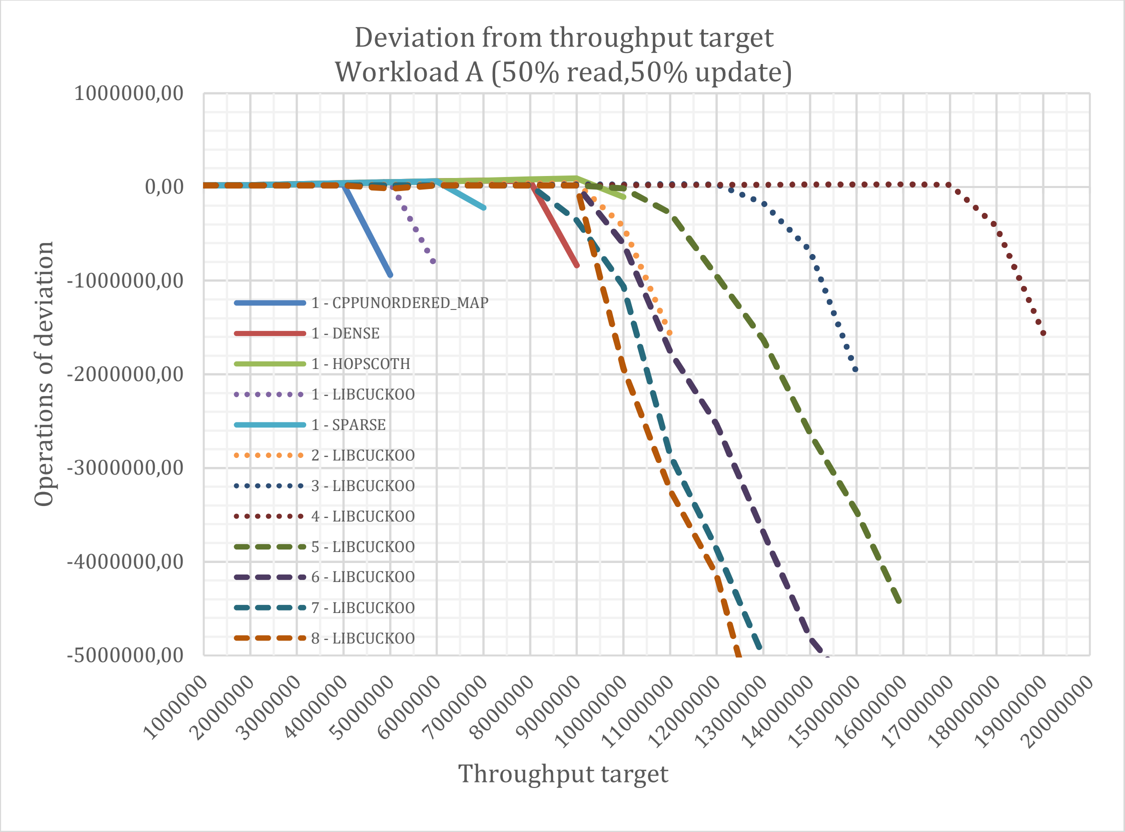


Figure 9-3 This graph shows the number of operations in deviation from the throughput target rates. The number prior to the interface name is the number of threads tested with.

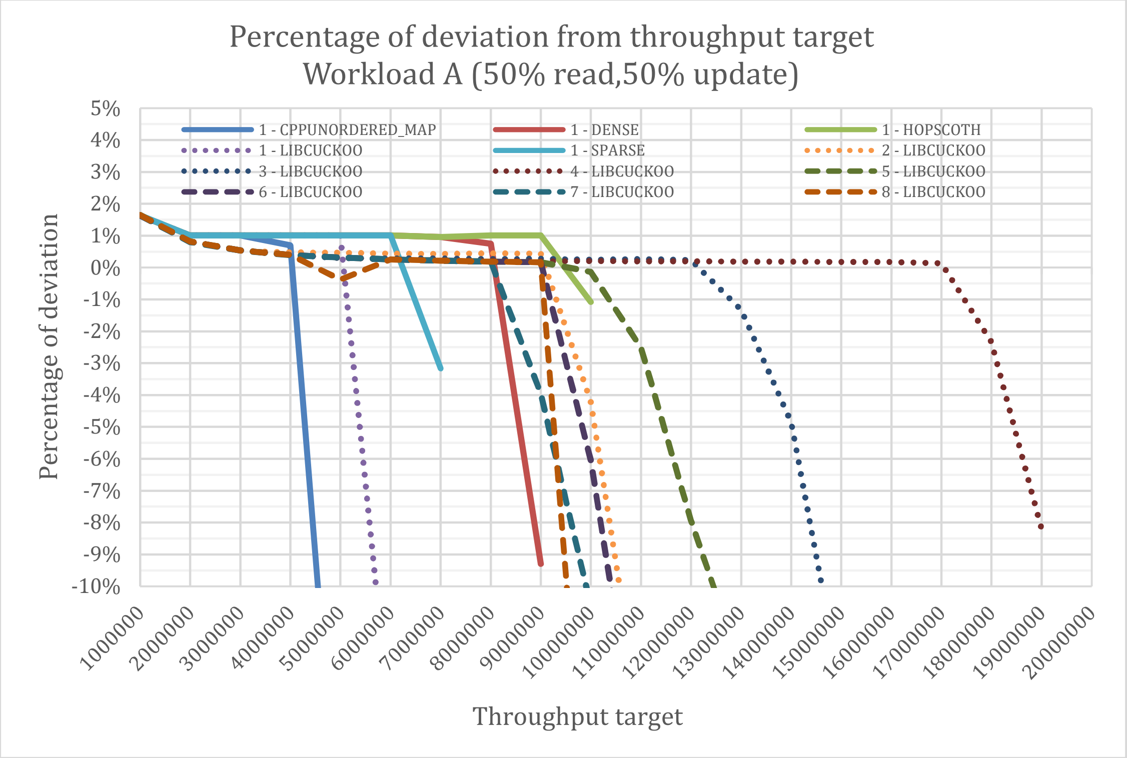


Figure 9-4 This graph shows the number of percentage of deviation from the throughput target rate. The number prior to the interface name is the number of threads tested with.

Figure 9-3 and Figure 9-4 show the deviation from the target rate in operations and percentage. The most interesting observation here is in the percentage of deviation in graph Figure 9-4. Prior to an interface reaching its maximum throughput, the percentage of deviation is constant at 1% when using one thread, and then using more than one thread it is slightly better closer to 0,5%. The most natural correlation with this result is that the number of operations performed is greater when more than one thread is used. Therefore, the execution time is longer. In addition to this there seems to be a slight deviation at the lowest throughput target of 1 million operation per second.

9.1.3 Latency sampling

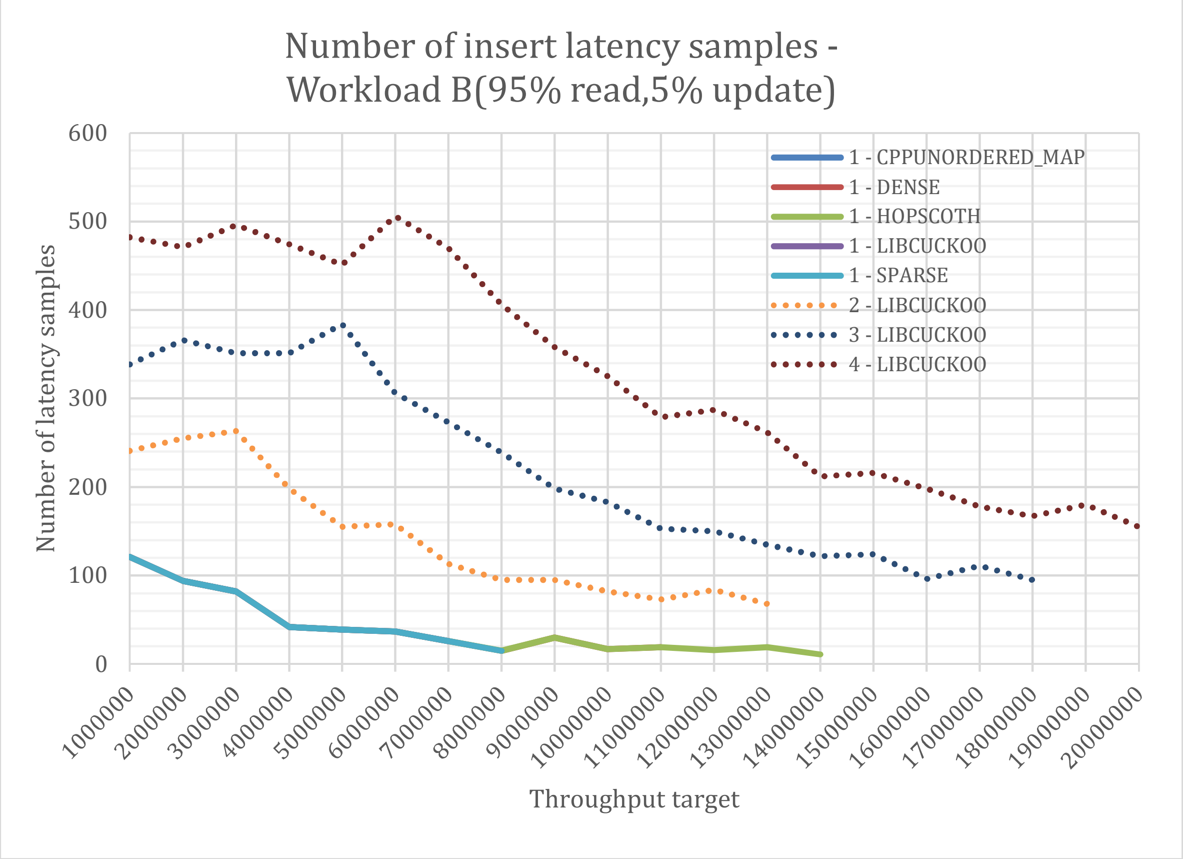


Figure 9-5 This graph shows the number of insert operation latency samples collected, which is the individual measurement of latency during execution with different throughput target rates. The number prior to the interface name is the number of threads tested with.

Latency results are calculated from a set of individual latency samples. Figure 9-5 shows the average number of samples in the set of latency samples, which the statistical latency results are derived from. All single threaded interfaces follow the exact same line, this is an artefact of how latency samples are taken, and that they all use the same trace which make this deterministic. Since workload B is 95% reads and 5% updates, only 5% of the total amount of latency samples are insert samples. When combined with the fact that the execution time decreases with the target throughput, so does the number of collected latency samples. This means that for workload B the number of insert samples taken is too low to be statistically significant.

9.2 Phase 2 preload

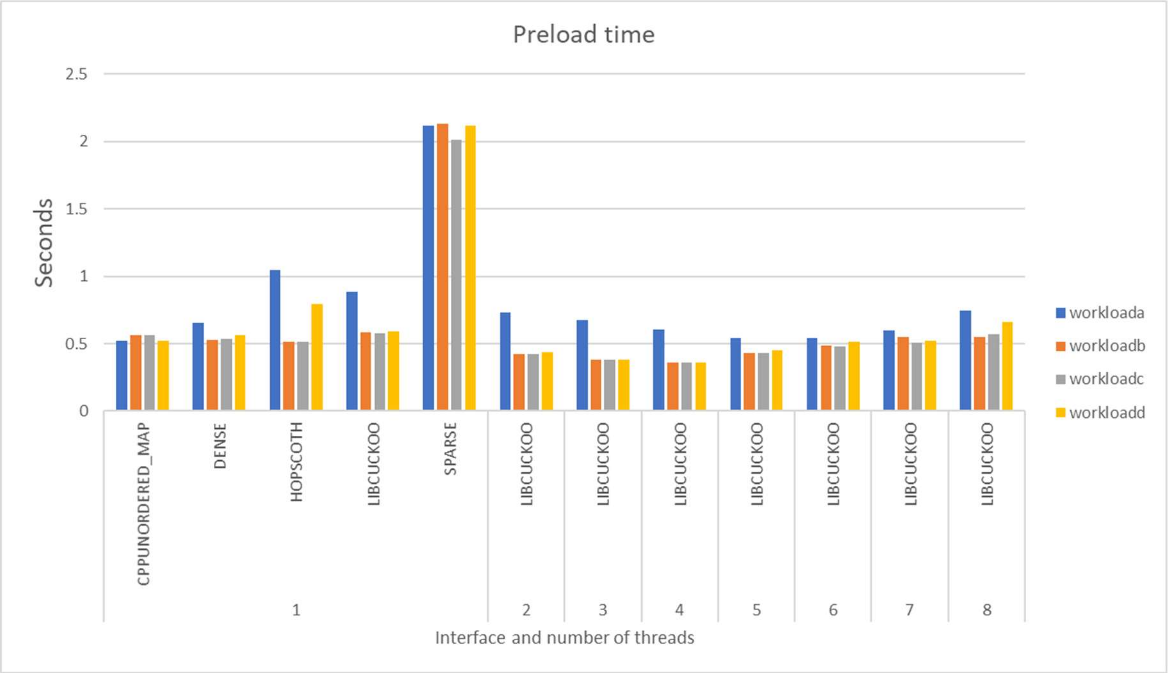


Figure 9-6 This graph shows the average time used to preload different workloads. As the preload phase is insert only, there is effectively no difference between the workloads.

Figure 9-6 shows time used to preload a million records into the key-value store. It is important to note that the variations between the different workloads are random. As the preload phase has no variation across workloads, they are all 1 million insert operations. The variations are likely to be there due to random fluctuations. The time durations are so short that the following graphs of energy per operation and operations per second, is likely to be inaccurate, and only general trends are in line with the results from the execution phase.

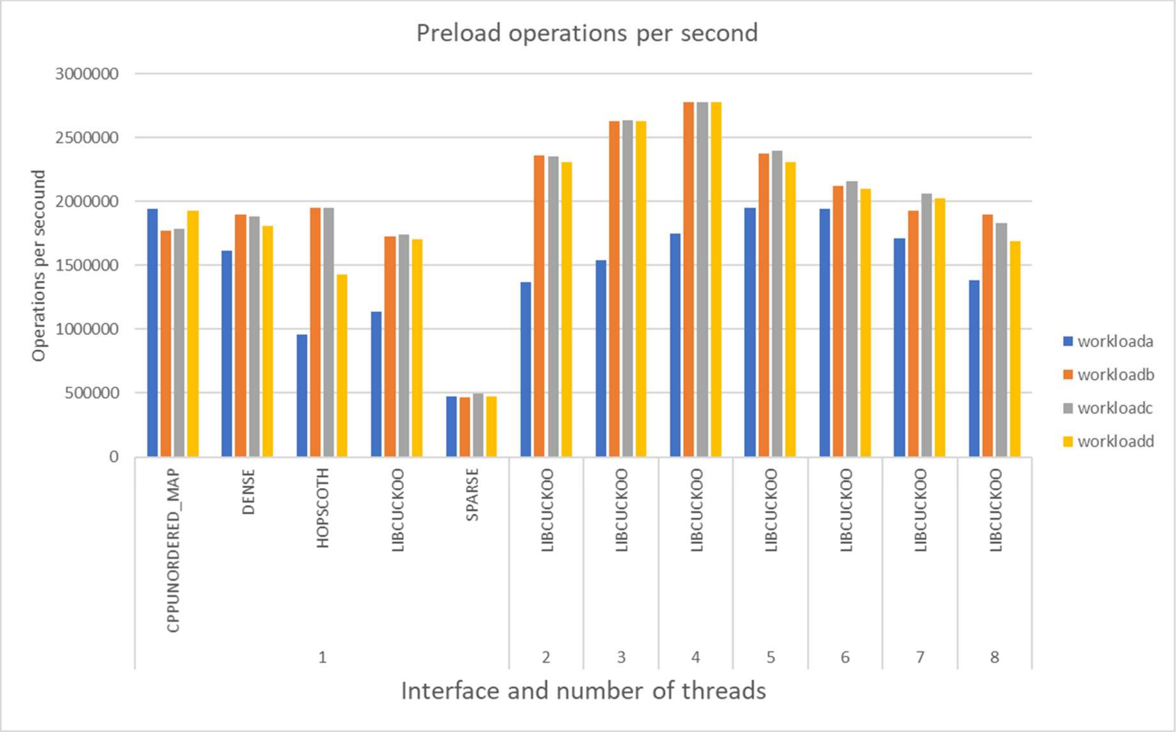


Figure 9-7 This graph shows the average operations per second used to preload different workloads. As the preload phase is insert only, there is effectively no difference between the workloads. It is uncertain why workload A stands out in this graph. This warrants investigation.

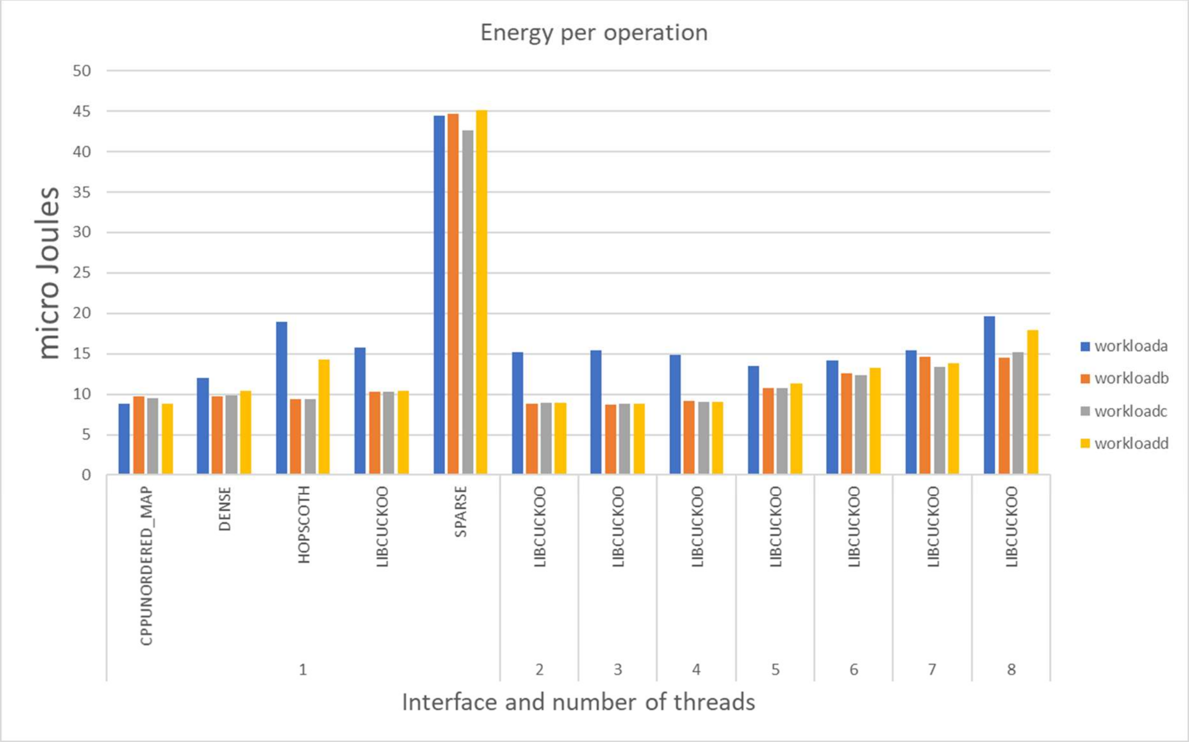


Figure 9-8 This graph shows the average energy use per operations used when preloading different workloads. As the preload phase is insert only, there is effectively no difference between the workloads. It is uncertain why workload A stands out in this graph which warrants investigation.

9.3 Phase 3 execution

9.3.1 Correcting for background energy use

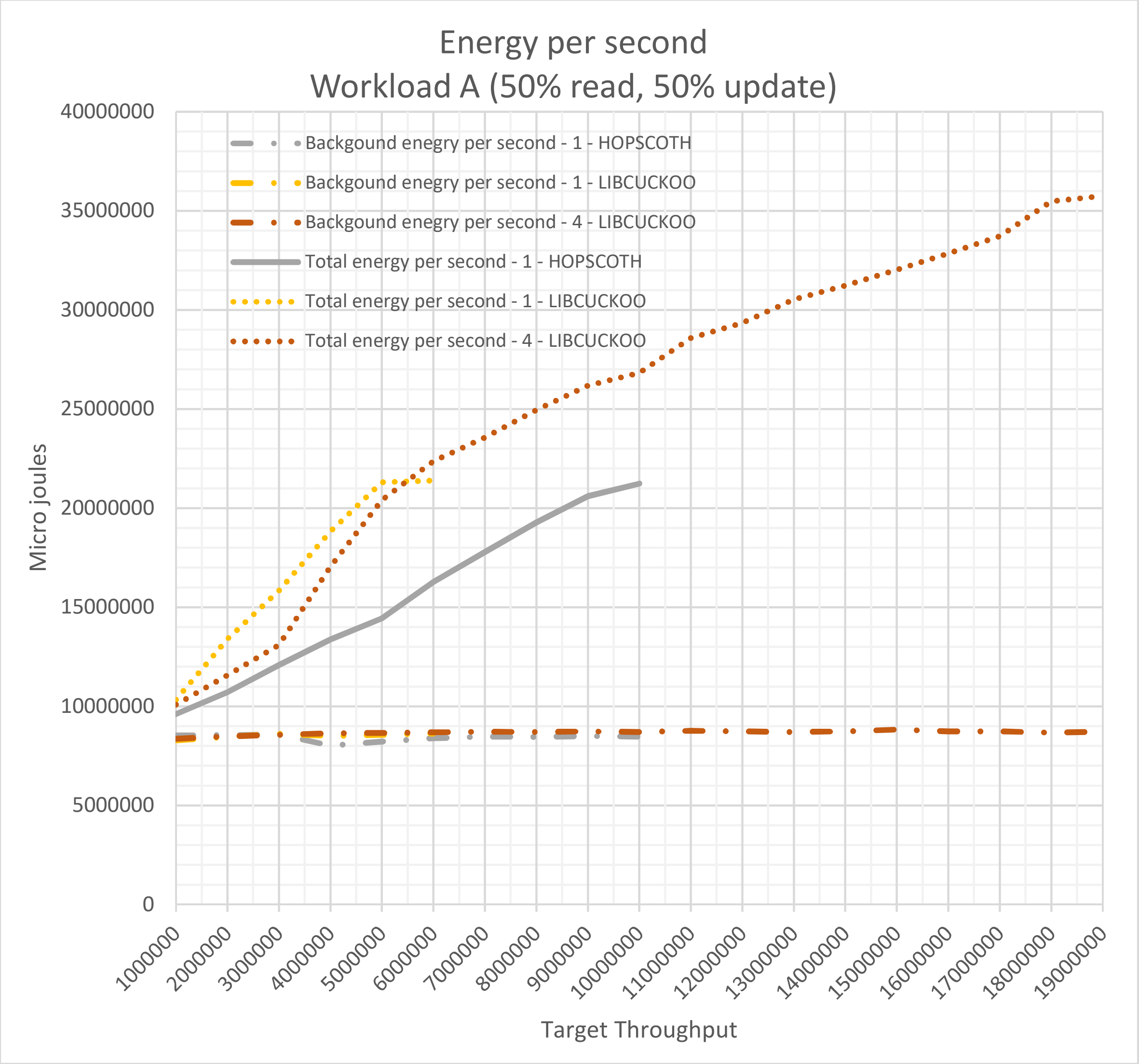


Figure 9-9 graph shows the energy use of three interfaces. The energy use per second while executing and the energy background energy use prior to execution, also per second. Number prior to interface name is the number of threads used.

The background energy use of the system is constant as can be observed in Figure 9-9. Whereas the energy use of the interfaces during execution starts just higher than the background energy, and increases at different rates as the throughput targets increase. If the background energy use is not considered when calculating energy per operation, results will be skewed so that lower throughput's get worse energy per operation (results see Figure 9-10). All the subsequent (accept Figure 9-10) results of energy per operation, correct for the background energy use. By subtracting the background energy from the total energy before dividing the number of executed operations.

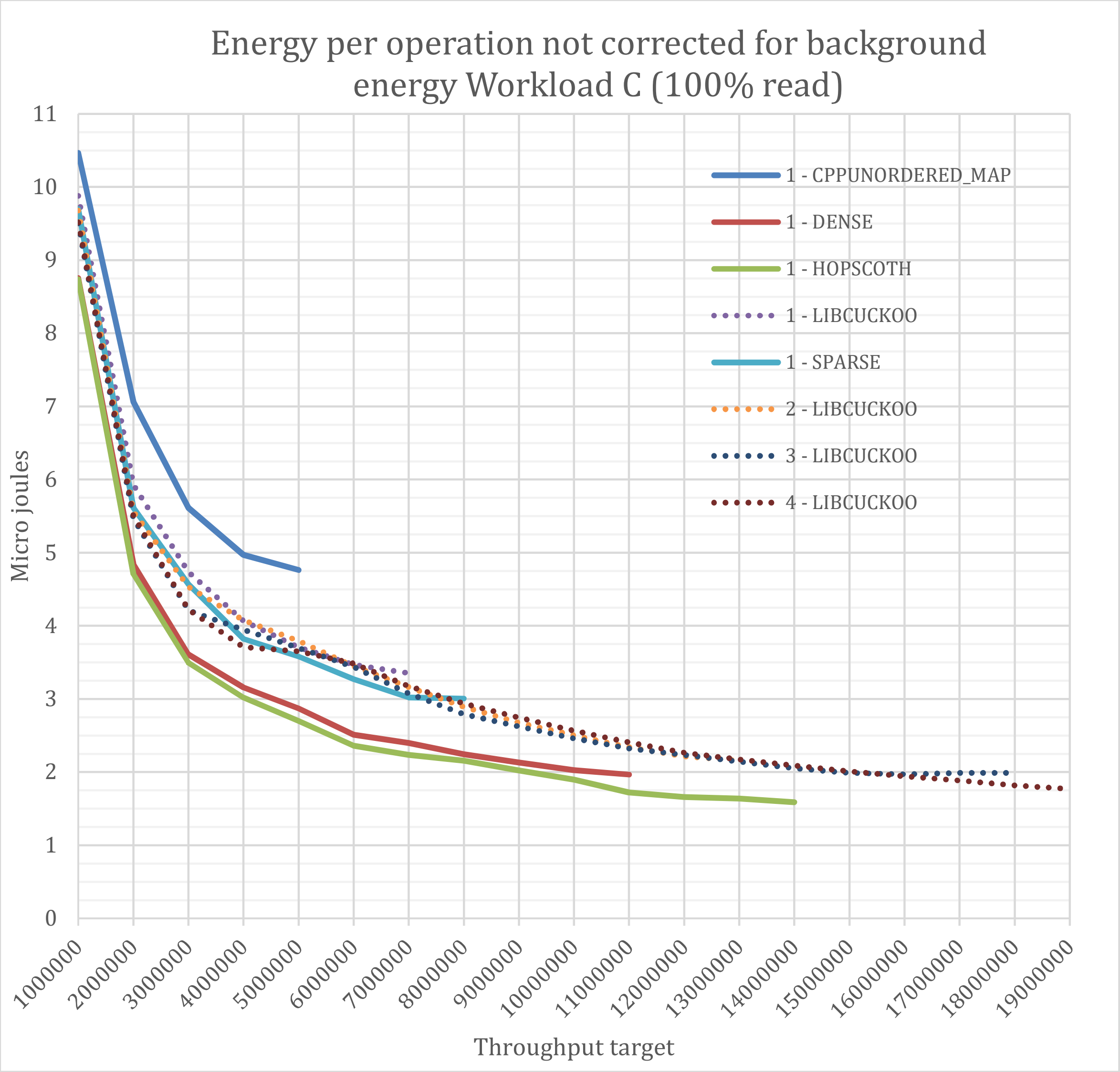


Figure 9-10 Graph shows energy per operation where the background energy is not subtracted for the total energy used prior to dividing it by on the number of operations. This skews the results negatively for lower throughput targets. Number prior to interface name is the number of threads used.

9.3.2 Energy per operation

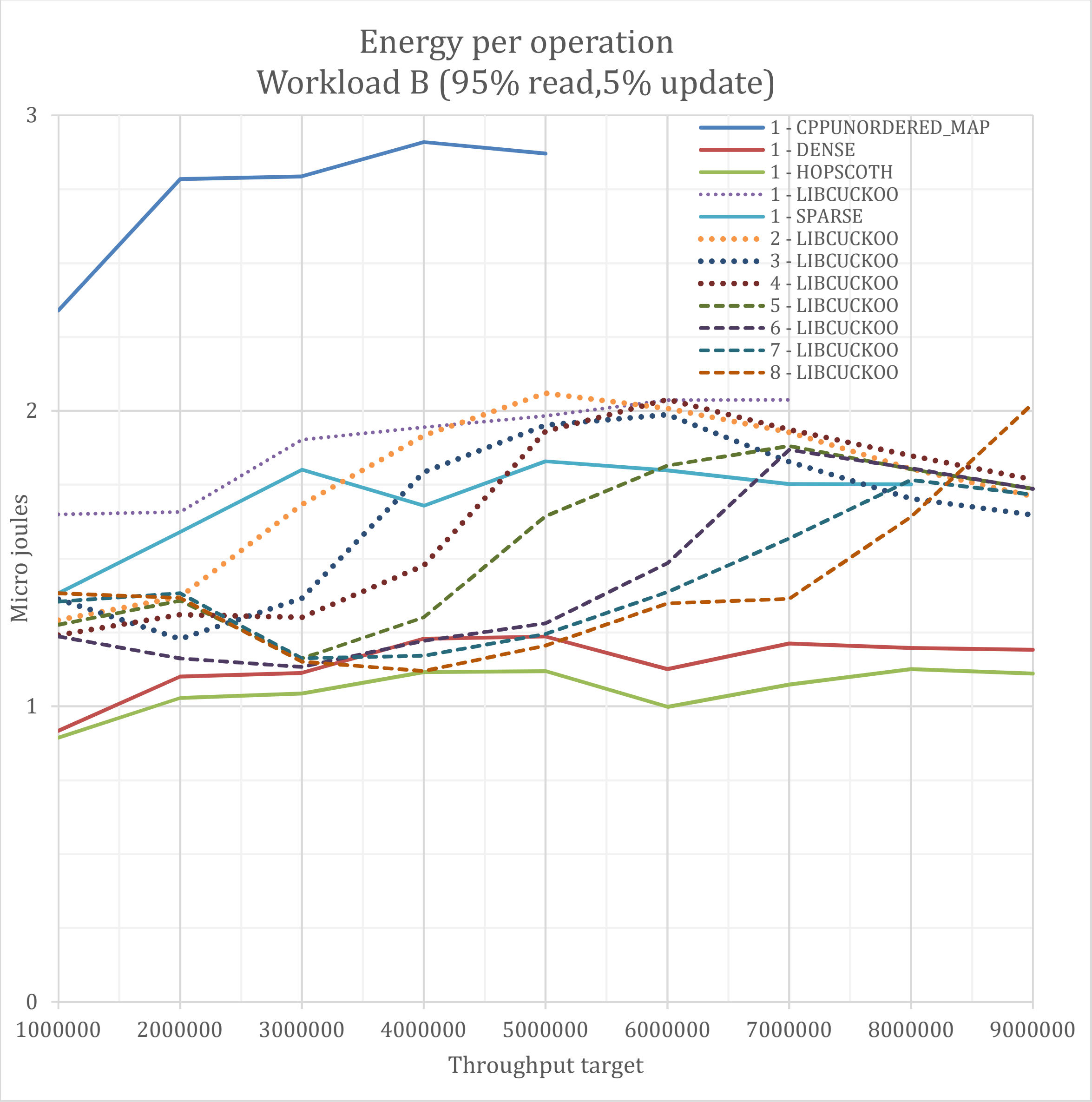


Figure 9-11 The energy use per operation for throughput targets up to 9 million operations per second. It includes Libcuckoo with more than four threads. Number prior to interface name is the number of threads used.

Figure 9-11 shows an interesting behaviour of Libcuckoo. Libcuckoo with more than 4 threads outperforms Libcuckoo with four threads on the hardware which has 4 cores that are not hyper- threaded. Likely because of lower throughput, the CPU has time to context which between threads to hide latency. For the rest of the results, Libcuckoo with more than four threads will not be shown, as this is the only interesting insight their results contributes.

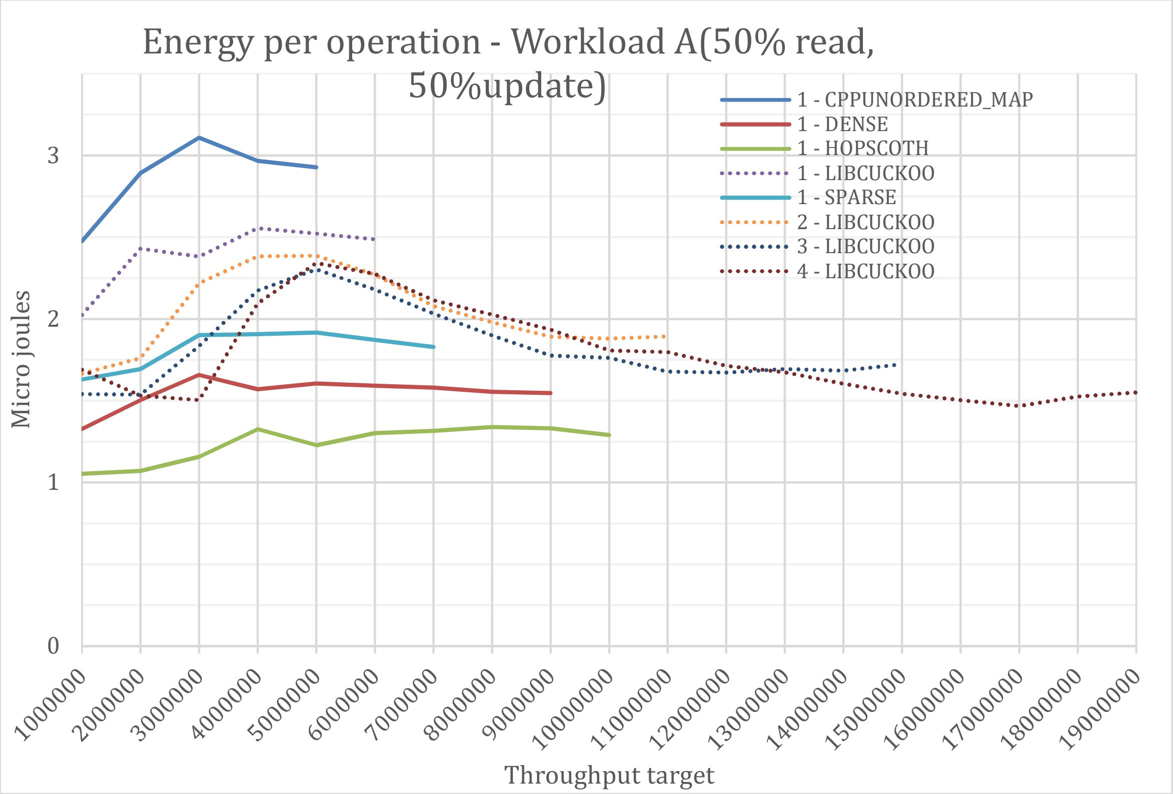


Figure 9-12 The energy use per operation for all throughput targets with workload A. Number prior to interface name is the number of threads used.

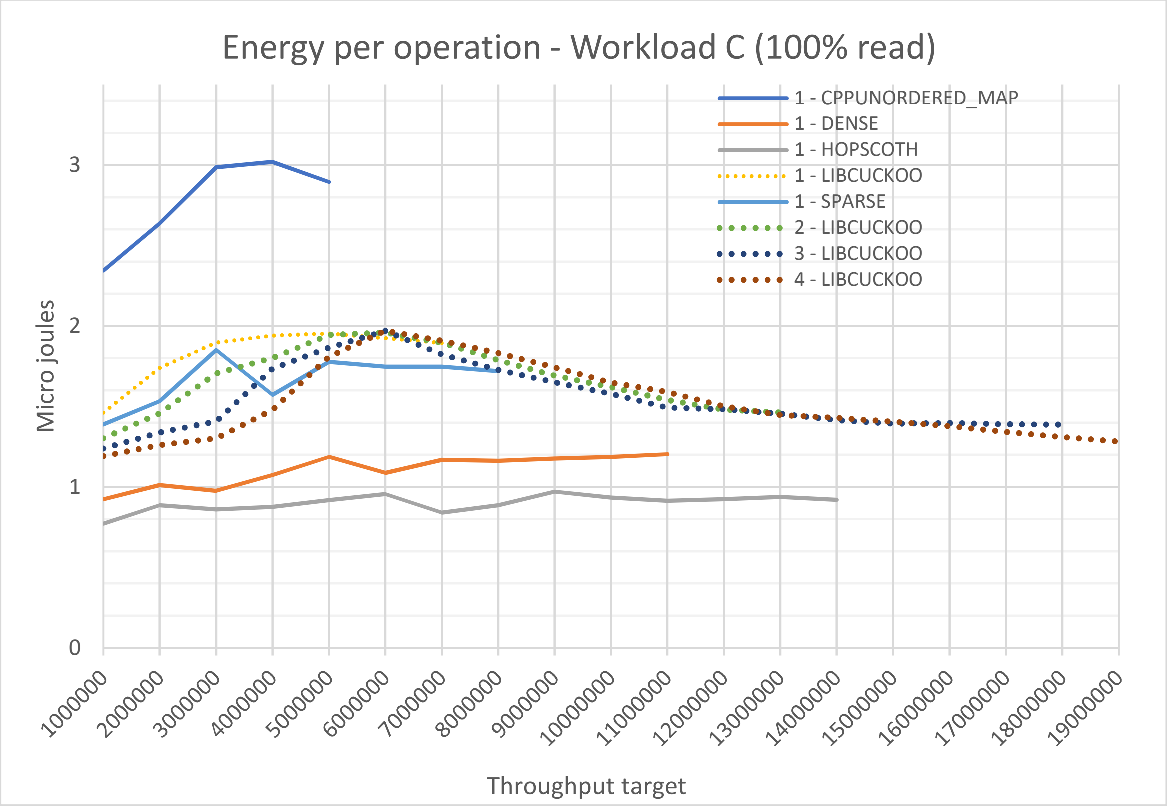


Figure 9-13 The energy use per operation for all throughput targets with workload A. Number prior to interface name is the number of threads used.

The results for both workload A and C are quite similar. Hopscotch is slightly better than Google's dense hash. Libcuckoo performance better the more threads it uses, but still not with large variations, with the exception for with one thread. Moreover, there is an interesting point at the throughput target of between five and 6 million, where the results of the libcuckoo interfaces are almost identical, and after which its results steadily improve.

It is important to be aware that the line where these plots end, which is highest throughput target reached the interface. This point does not reflect the average maximum throughput, but rather the single highest target throughput reached for the given interface. This result should be seen in conjunction with Figure 9-14 which gives the average maximum throughput results.

9.3.3 Maximum throughput

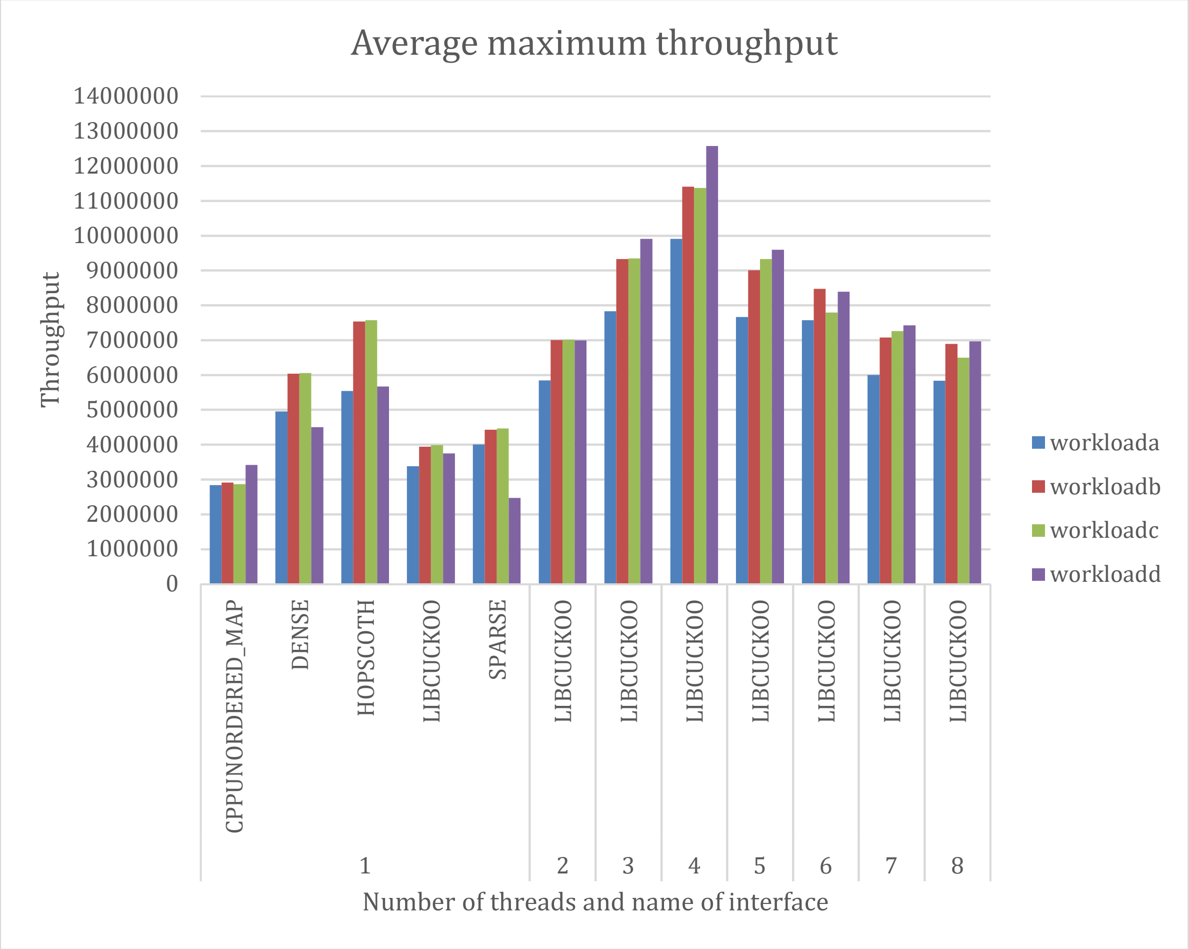


Figure 9-14 This graph shows the average maximum throughput measured with different workloads. The number beneath the interface name is the number of threads used.

With one thread, hopscotch performs best with all workloads. The concurrent Libcuckoo does performs best with four threads which is the number of cores in the system.

9.3.4 Latency

The latency is measured in nanoseconds and the resolution of the clock on the system is 1 nanosecond. Please note that all drafts and plots in the section start at 600 ns, no observations were as low as 600 ns. When percentile is used in the following graphs and plots, it means the percentage of operations were faster than that. For example, if the 70th percentile is 700 ns, 70% of operations had latency lower than 700 ns.

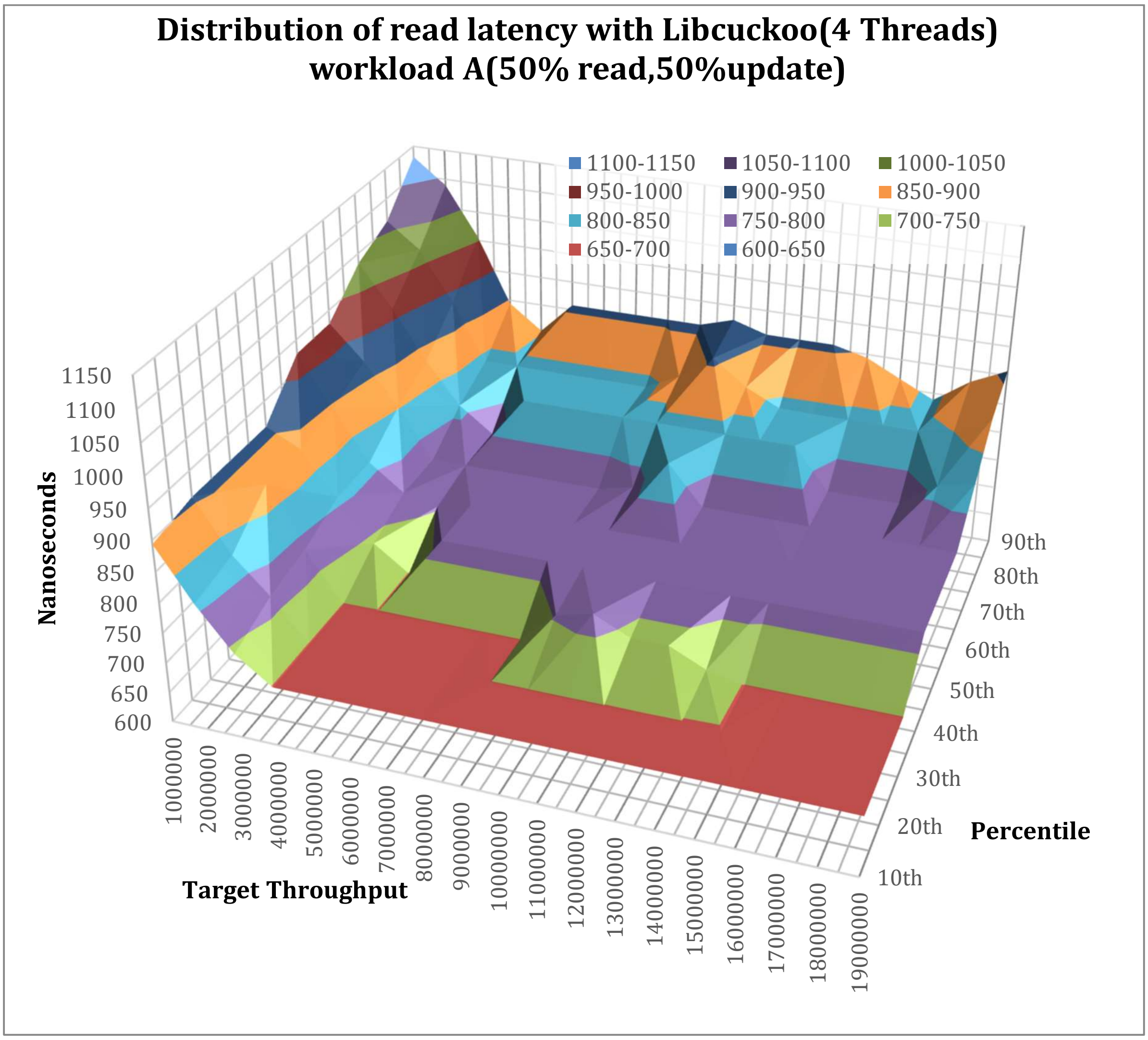


Figure 9-15 This plot illustrates how that latency percentiles are distributed across all the throughput targets.

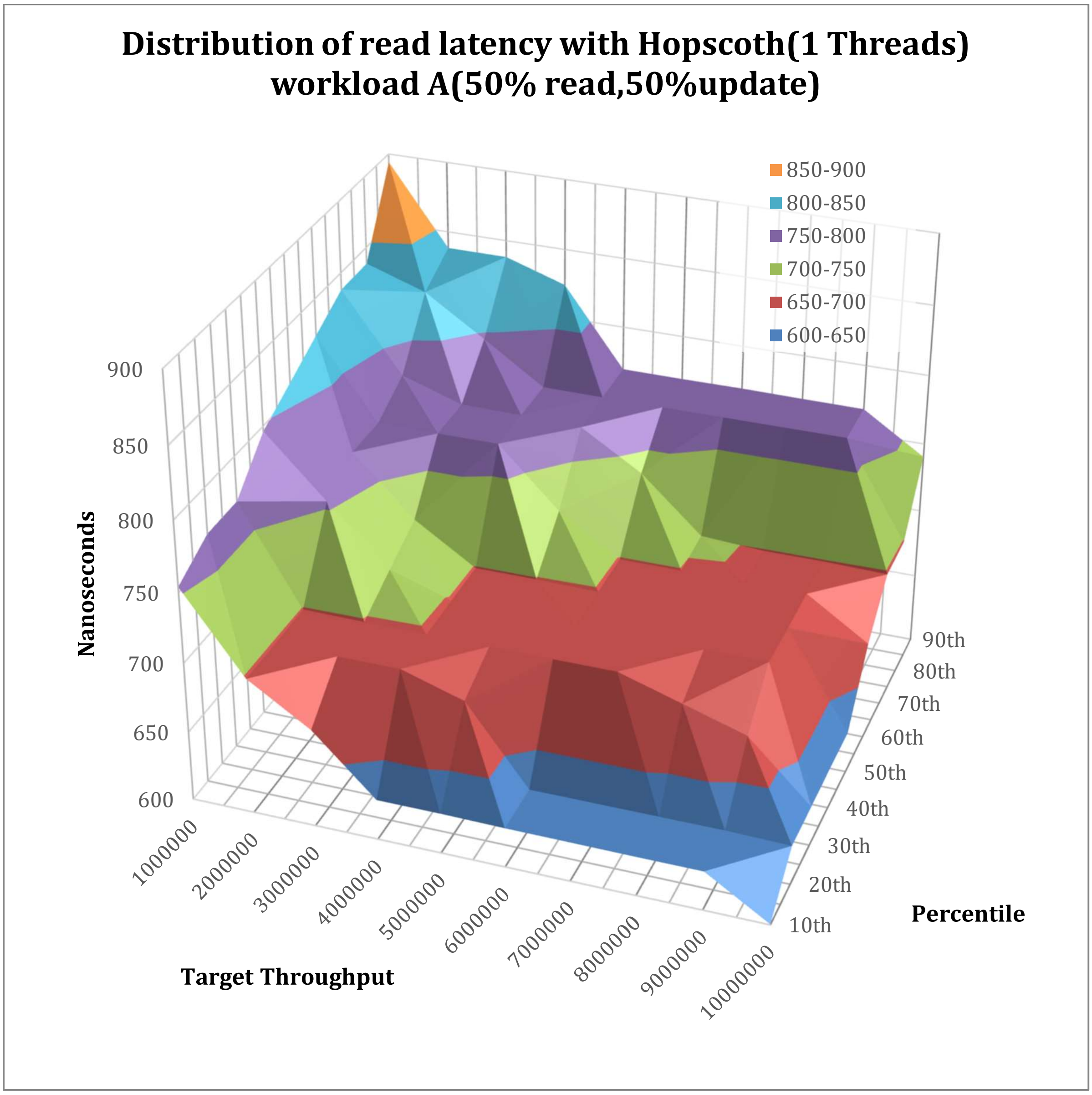


Figure 9-16 This plot illustrates how that latency percentiles are distributed across all the throughput targets.

The evaluation framework makes it possible to get a multi-dimensional insight into the latency distribution. Figure 9-15**Feil! Fant ikke referansekilden.** and Figure 9-16 illustrates this. This makes it possible to identify variations in latency more precisely at different throughputs. Figure 9-15 illustrates this best; it has a rapid decrease in latency across all percentiles when the target throughput is from 1 million to 5 million operations per second from where it flattens out with a slight peak again at 11 million operations per second. The same drop and flatting out can be seen in Figure 9-16, at the same target throughput of 5 million as in **Feil! Fant ikke referansekilden.**. When all the latency distribution plots are examined (not show in rapport to due to lack of space), the relative distribution of percentile does not vary a lot with target throughput. Therefore, averaging the percentile distribution of all target throughput is representative for the different interfaces see Figure 9-17. This graph shows that serial interfaces have the best latency results, all of them outperforming the concurrent libcuckoo. Also note that the standard C++ unordered map which has the worst results overall in other performance metrics, are among the best in latency.

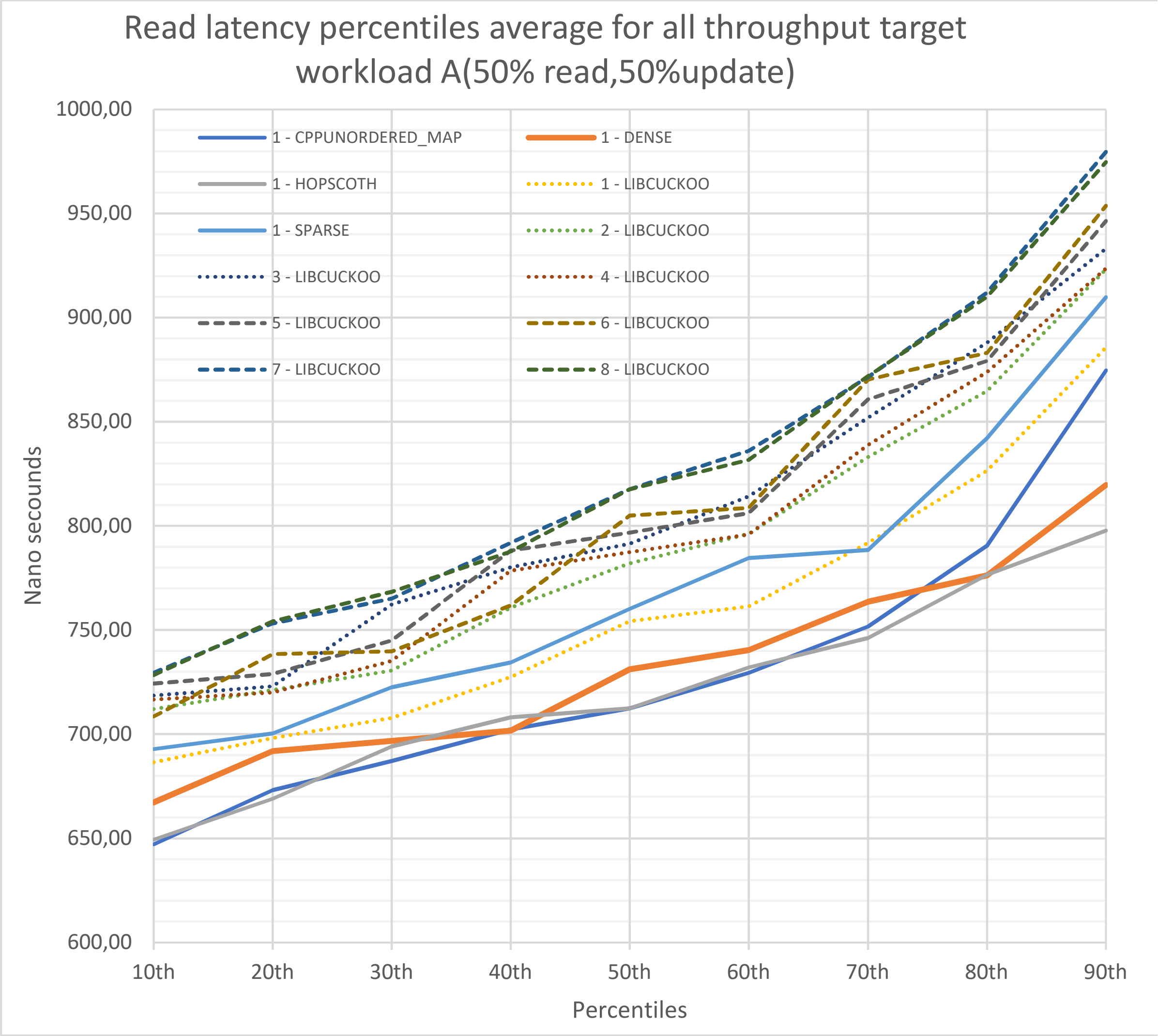
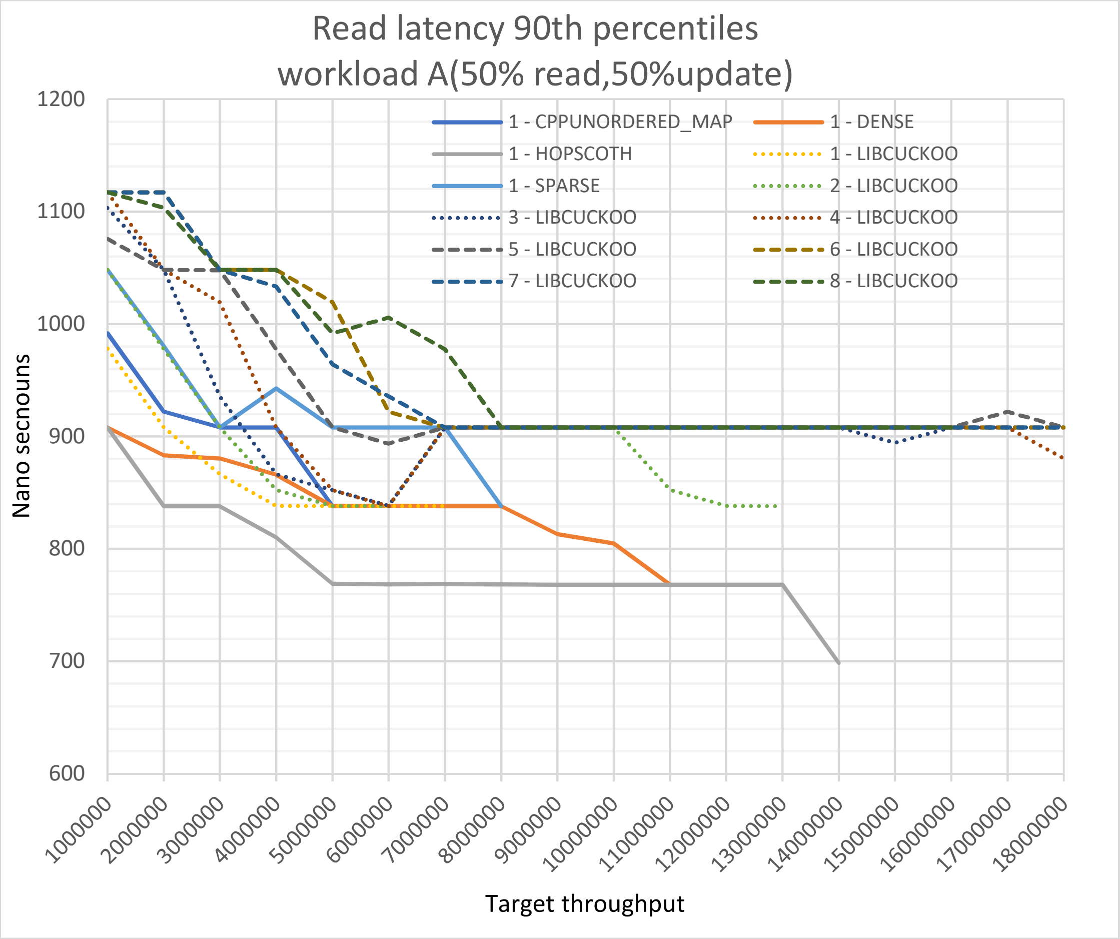


Figure 9-17 This graph shows the results average percentile distribution across all throughput target rates with workload A.



z

Figure 9-18 this graph shows the 90th percentile for all throughput targets with workload A.

The 90th percentile is an interesting metric of latency, as it reliably informs us of what latency you can expect 90% of the time. Figure 9-18 shows some interesting results for latency across the different target throughputs. The best performers are again hopscotch and Google dense hash, the unordered map is not performing well in this percentile. However, Libcuckoo with up to 4 threads, matches Google dense hash for some target throughputs. Before Libcuckoo latency increased to around 900 ns, it remained constant for the remaining throughput targets. In general, the serial interfaces has the best latency, with hopscotch as the best flattening out at around 760 ns. It is important to note that the latency drop both Google dense hash and hopscotch have when approaching their maximum throughput, is likely due to outliers in the data set and should not be considered reliable.

There is ,however, a general trend all interfaces have in common. They have a relatively constant decrease in latency from the lowest throughput target to around 5 to 6 million operations per second. The initial latency for the lowest throughput also seems to increase the more threads are in use. This will be discussed further in section 10.2.

9.4 The theoretical use cases

The results will now focus around one of the intended use cases for the evaluation framework, evaluating an application' specific interaction characteristics with the key-value store. As I have no real world application data to take from, there are created two theoretical use cases based on the YCSB core workloads. From the characteristics of these theoretical use cases, the best interface for that application can be determined based on the performance metrics.

9.4.1 Application A

Theoretical application A is an application that conforms to YCSB core workload A, and has an average throughput that varies between eight and 10 million operations per second.

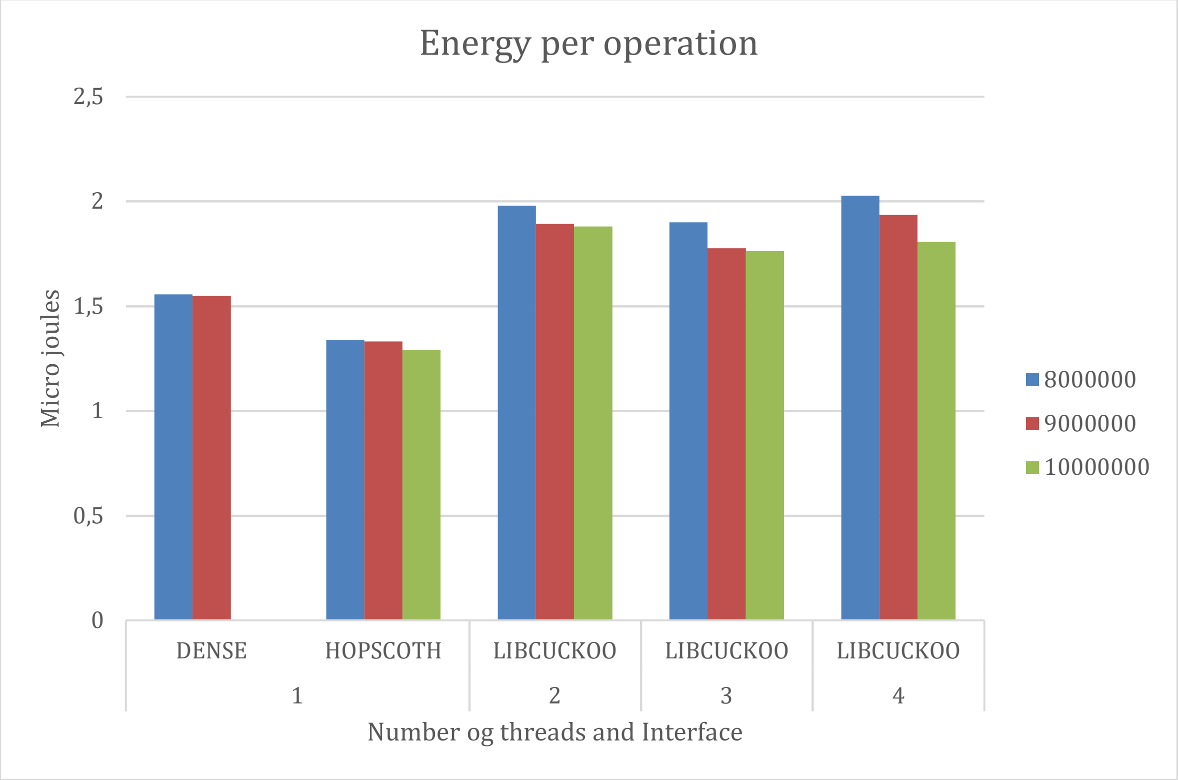


Figure 9-19 This graph shows energy per operation with all the interfaces that can deliver the performance required by application A. The colours represent different throughput targets. The dense interface does not have a grey (10000000) graph as its average maximum throughput is lower than 10 million operations per second.

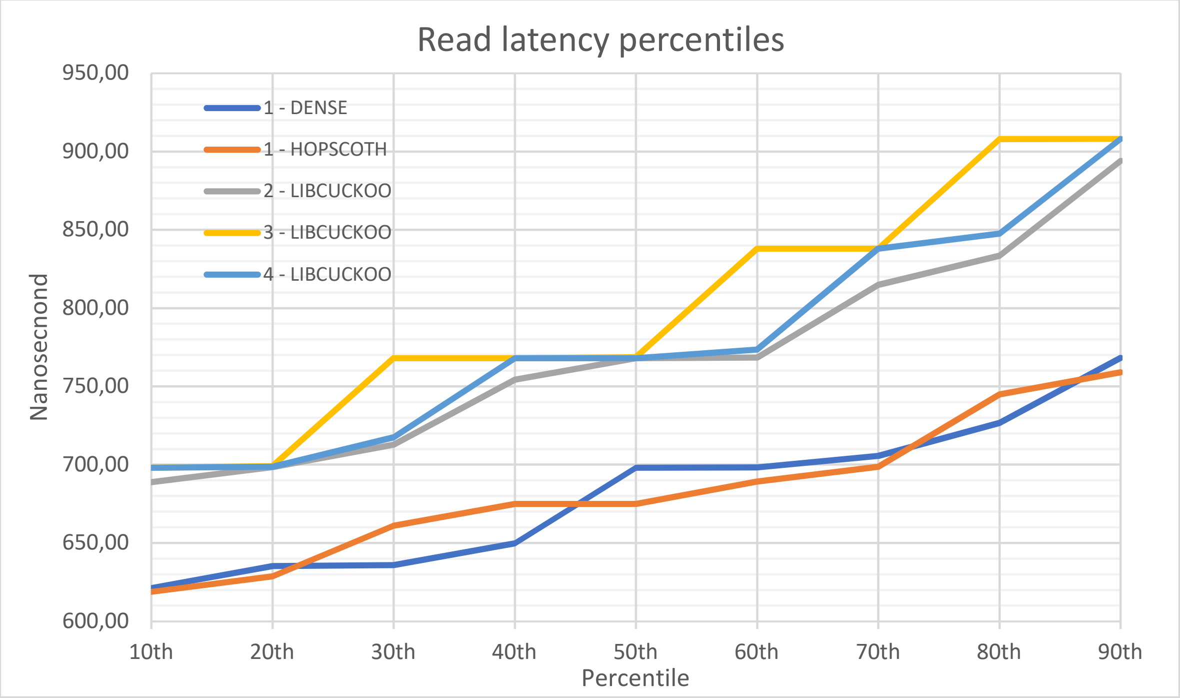


Figure 9-20 This graph shows read latency percentiles with all the interfaces that can deliver the performance required by application A.

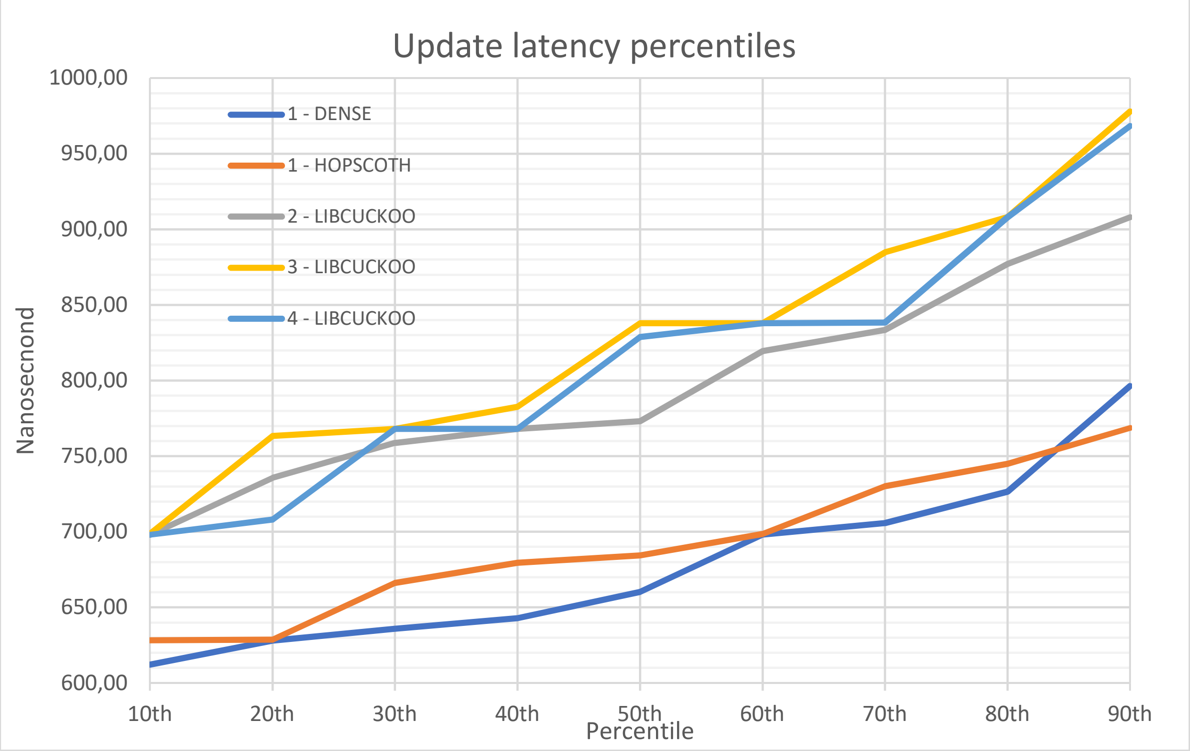


Figure 9-21 This graph shows Update latency percentiles with all the interfaces that can deliver the performance required by application A.

Figure 9-19, Figure 9-20 and Figure 9-21 show the different interfaces which can perform the throughputs that application A need, with the exception of Google dense hash which cannot reliably perform 10 million operations per second. On all performance metrics, hopscotch performed the best. However, if application A and some point can expect to need to run at higher throughputs, only the Libcuckoo can achieve that.

9.4.2 Application B

Theoretical application B is an application which conforms to YCSB core workload C, and has an average throughput that varies between 2 and 3 million operations per second.

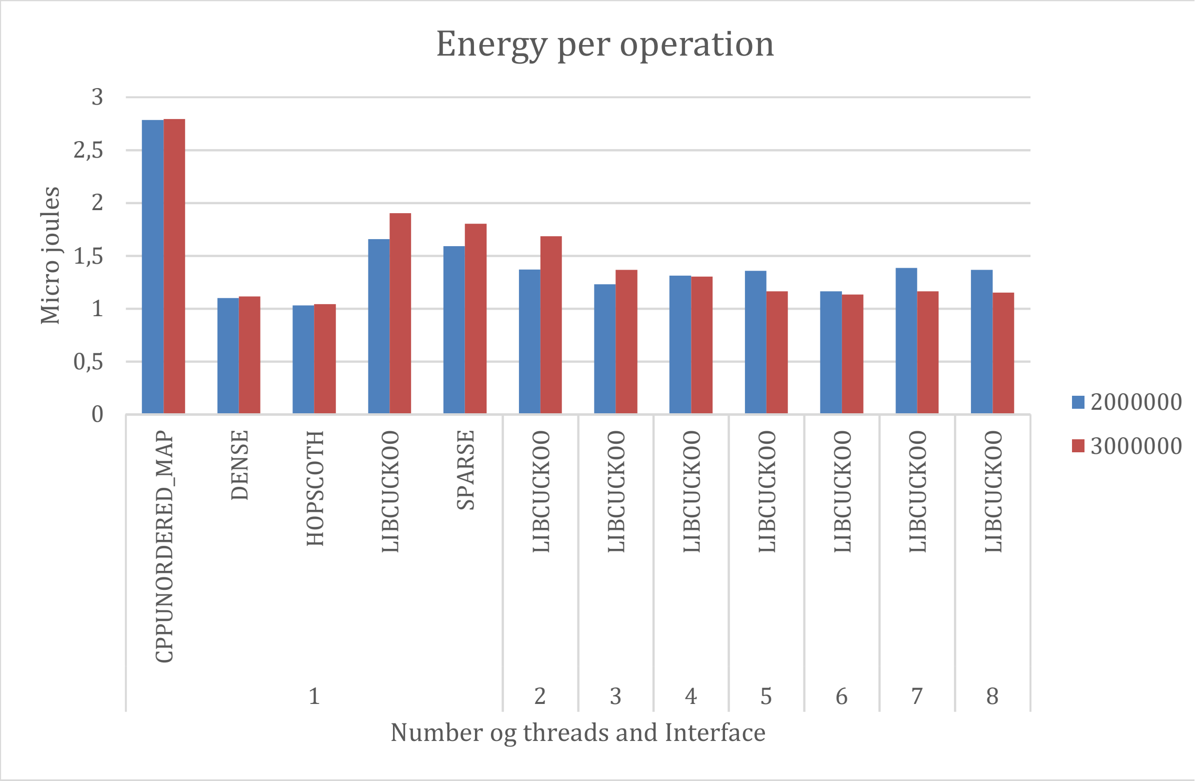


Figure 9-22 This graph shows energy per operation with all the interfaces that can deliver the performance required by application B.

Figure 9-22, Figure 9-23 and Figure 9-24 show all interfaces that meet applications B throughput requirements. At this low throughput, the difference between interfaces is quite small, hopscotch is slightly better than Google's dense hash for energy efficiency. As for latency the unordered map is slightly better than hopscotch, especially for updates. However, overall hopscotch is the best interface for application B needs.

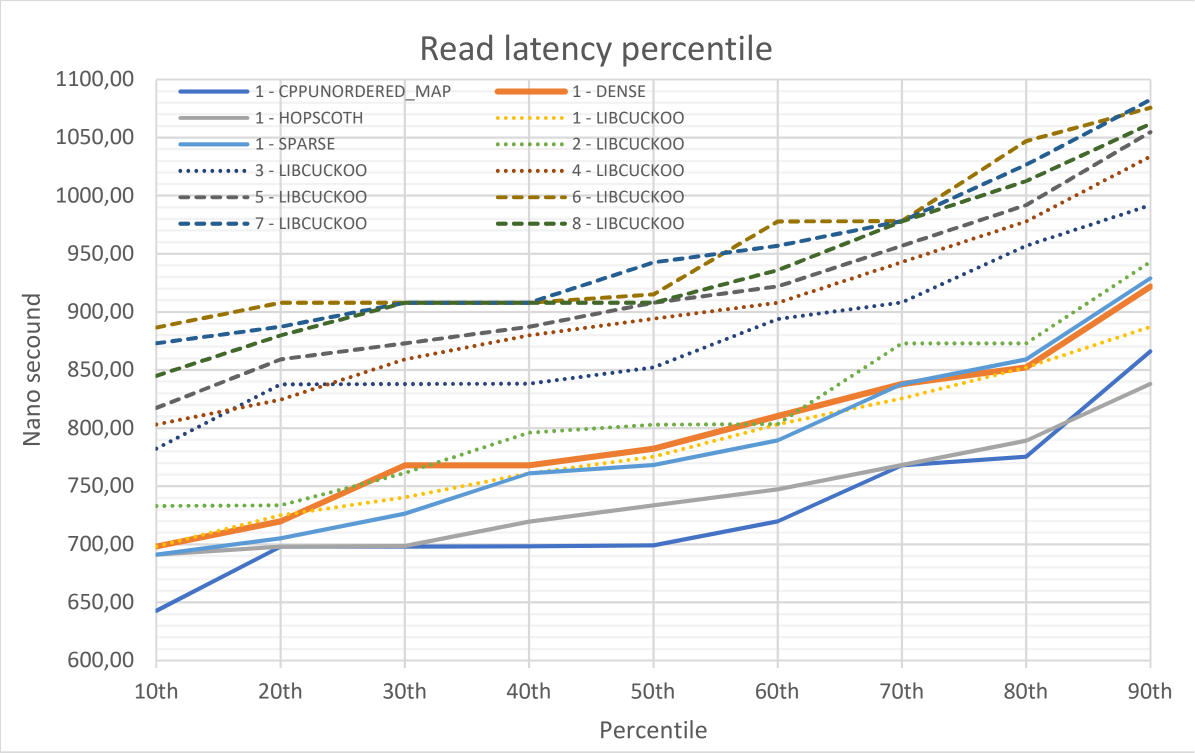


Figure 9-23 This graph shows read latency percentiles with all the interfaces that can deliver the performance required by application B.

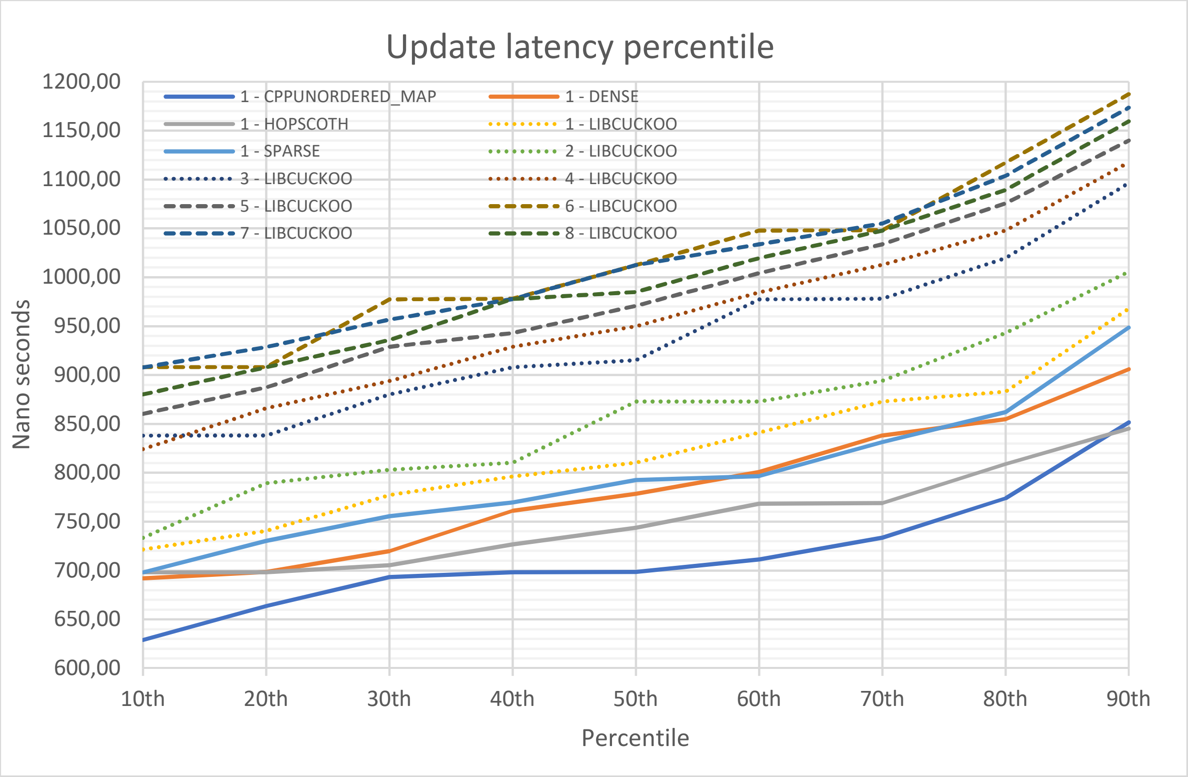


Figure 9-24 This graph shows Update latency percentiles with all the interfaces that can deliver the performance required by application B.

10 Discussion

10.1 The evaluation framework

In this section, the pros and cons of the implementation and the different aspects of the value for evaluation framework, is discussed. Starting with the design and implementation of the throughput control, the issues with using YCSB the trace workloads, and how these issues affect the implemented method of latency measuring.

10.1.1 Throughput control through intervals.

As the results show, the implemented method of controlling the throughput worked well. Deviation from the target throughput is relatively constant at approximately 1%, and when deviating from this main, within 0% to 2%, up to the point where the target throughput and the maximum throughput of the interface meet and the interface was unable to meet the target throughput, and as expected deviating below 0%.

10.1.1.1 Problems with the target throughput termination criteria

There was however, one problem with the implementation, and that was the maximum throughput criteria. The idea was that up until the maximum throughput, each thread would have lesser and lesser time at the end of the interval to sleep after the interval target is reached, at which point the threads would not sleep for any duration. Indicating the maximum throughput was reached, however, this only holds true for single threaded executions. When the number of threads increases, this assumption fails when the number of threads were equal lower to a number of cores in the CPU. The framework attempted a few throughput target iterations past maximum throughput. However, if the number of threads exceeded the number of cores on the CPU, the framework attempted several more throughput target iterations beyond the maximum throughput.

This effect is believed to be due to the context switching between threads, and that some threads managed to complete their intervals and therefore sleep whilst other threads were unable to reach their target throughput. The reason the sleep time was used, was that it may be an interesting metric to keep track of, and that by using it as a termination criteria, the maximum throughput measured would not be affected. In retrospect, keeping track of sleep time give no interesting insights, and the termination criteria should probably have been based on the average throughput of all samples in a configuration.

This does not affect the results in any other way than that there are some measurements that need to be discarded. In addition, the execution time of the framework is unnecessarily increased, something that should be avoided as the execution time is already extensive.

10.1.2 Issues with using YCSB for trace generation

One assumption of the design was that the data set would be large enough to run for the entire test duration. This assumption underestimated that the speed of modern key-value stores, using YCSB to generate a trace of 100 million operations, was not enough to run for 60 seconds. It is only enough operations to run for 60 seconds at the throughput of 1 million operations per second. Generating a larger file is possible but impractical. Each workload takes an hour to generate, and the largest one is over 11 GB in size. There were also concurrency issues which will be discussed in the next section, considering that to execute for 60 seconds at close to the highest maximum throughput, measured at 20 million operations per second would require a data set of 1.2 billion operations, which is completely unfeasible in a pre-generated file.

The usual solution to this problem is simply to loop through the trace file several times. The distribution and the access pattern will be the same. This is true, but with one very important exception. The pattern will be the same as long as there are no insert operations in the pattern. The following example illustrates this: The YCSB core workload D is 95% reads and 5% inserts. The first time this trace is iterated through 5% of the keys, will be insert operations. However, the second time the trace is iterated through, the 5% of keys that are set to be inserts have already been inserted. In other words they are now effectively updates, and this would be true for every consecutive iteration through the trace. So, insert and update operations are effectively simplified to generic put operations. Insert implies that the access pattern will increase the size of the key-value store over time.

Using the example given earlier with 20 million operations a second as the throughput running for 60 seconds, a 100 million operations trace will have to be looped through 12 times. If that trace has 5% inserts they will only actually perform insert operations for first iteration of the loop. That is 8.3% of the total number of operations that reflect the access pattern, the remaining 91,7% will perform 5% update operations. For this evaluation framework, which primary purpose is to correctly simulate applications interaction characteristics with the key-value store, this is an unacceptably large deviation, and I would generally argue that it is incorrect to perform experiments with insert operations in this way.

10.1.2.1 Concurrency issues

The by far the biggest benefit of using YCSB is that it can produce access patterns with different distributions. A necessity when trying to simulate an application’s interaction characteristics with the key-value store and was the primary reason it was chosen to generate traces from the access patterns. The initial implementation divided the operations of trace between the number of threads so that each thread executes its own segment of the trace. This did however create some concurrency issues for access patterns with insert operations. As the trace is designed as a sequential execution of operations, threads starting in later sections of the trace would attempt to read keys that have not been inserted yet. As this insert operation is in an earlier section which another thread has not yet reached in its execution, this will cause a read failure. In itself this might not be such a large issue, read failures can after all occur, but these are unspecified read failures, and they would occur more frequently the more threads that are used. It would incorrectly represent in applications interaction characteristics with the key-value store. It could be argued that the access pattern should also contain failure operations see section (3.2), but operation failures are probably rare and I’m not aware of any work on the subject.

10.1.2.2 Possible solution with YCSB.

If YCSB must be used, the best solution for this issue will be to generate one trace files generated for each of the threads in any configuration. So, if you have one thread you only require one trace file, when you have three threads you would require three trace files. With this solution the concurrency issue's would have been solved, as each thread has its own segregated trace to perform. However, this is still not a perfect solution. It is reasonable to believe that a key inserted by one thread might be updated or read by other threads. Therefore, this solution could theoretically create less contention than is realistic in a real word scenario. The reason this method was not implemented, it's that it is cumbersome and would require a reimplementation of the trace generation input of the evaluation framework. It would also require the generation of more YCSB trace files, testing with 1 to n number of threads would require $T_n = \frac{n(n+1)}{2}$. For example, testing with up to 8 threads with the 4 workloads used here, would require 36 separate files, of a total size 944 gb. which is unfeasible. So it would likely have to be implemented by creating a pool of multiple trace files, so small that they could be divided evenly among the current amount of threads in the configuration by some form of trace management solution. There was not enough time to do this. If there had been time, it would probably be better to create an internal trace generator.

10.1.2.3 Implemented solution

The solution actually implemented was to have each thread the use the same trace, so each thread reads sequentially from the same thread, which is stored sequentially in memory. The inherent consequence of this is that the number of operations to be executed, increases for each extra thread that is used. That they all use the same trace, has some important ramifications. Each thread will use the same keys in the same order, which, if it hadn't been for the fact that each threads execution is offset from each other, would have caused contention issues as all threads access the same keys simultaneously. These contention issues would have unfairly negatively influenced on the performance of a concurrent implementations. However, since they are upset it would have the opposite effects, as the first thread which is not offset would have caused each entry to be loaded in the CPU cache. So, all subsequent key upset threads can read the entries from CPU cache instead of main memory which in all likelihood benefits its performance, possibly to a large degree. Minimizing memory operations is an important factor in the development of key-value stores, for hash tables in particular the amount of cache misses is a significant part of the retrieval time of entry.

Solution also suffers from the problems discussed in section 10.1.2, only to slightly lesser degree. The first thread which is not offset is the only one that performance inserts, all the other upset threads will effectively perform updates. It is therefore clear that YCSB is unsuited for this evaluation framework.

10.1.3 Latency

The benefit of measuring each type of operation individually, is that the latency differences between CRUD operations can be observed. However, latency measurements was implemented under the previously discussed assumption that the execution time would be fixed. Latency measurements samples can be take no more then each interval depending on configuration. Since the execution time is much shorter than assumed it would be, the amount of latency samples collected is reduced as the target throughput increases. If execution time can be extended by using a different trace generation solution than YCSB, this Would not be a problem. However, it is obvious that latency samples should be collected every interval regardless. It also has the added benefit of reducing the complexity of the execution loop. But in my opinion latency should be sampled at a fixed rate per interval, and not at a fixed rate per operation. Per interval because the amount of work each interval, and thereby each thread, uses to measure latency is constant over the execution time, and not a variable of throughput. This minimizes the impact the implementation of the operation framework has on the results.

10.1.3.1 Results handling

Latency results are collected and calculated at the completion of each sample in a configuration, which means that the latency results are an average of 5 samples. For example, the method read latency displayed in the results is the average of 5 mean calculations from 5 sets of latency samples. In other words, instead of summing up all the latency samples, it uses a subset of five smaller latency samples which could negatively affect the accuracy of the results. However, if the number of samples is great enough this is not a big issue. Even so, this is an implementation oversight that can and should be fixed.

For the results of this experiment, this problem might be an issue for the YCSB core workloads where one operation type consists of 5% of the total. Because in this case, only 5% of the latency samples will on average be from this operations and that can make the sample size too small to be statistically relevant, especially when the execution time is shorter see figure (placeholder)

10.1.4 The evaluation framework conclusion

To summarize and conclude the evaluation framework discussion: Using a pre-generated file to describe the access pattern does not scale for higher throughput targets. YCSB provided the very important aspect of being able to generate different distribution; it is also its only benefits. It makes the workflow of the framework more complicated, and it cannot truly support insert operations in an adequate matter. If a custom solution for generating traces can be designed and implemented, it could feasibly natively support different types of keys and length and provide scalability at higher throughput. This should not be up to a complex problem, however making it support different distributions might be. This will also resolve the issues with latency measurements.

10.2 Results observations

Reviewing the results there were a few observations in the data that warrant further discussion. Particularly the latency results see section (todo), but also in the energy results the section (todo). In both of these performance metrics, libcuckoo's performance changes at the throughput targets of around 5 to 6 million operations. The energy per operation increasing up to this point, and decreasing afterwards, a trend not observed in the serial key-value stores which had a more constant energy per operation. This observation is not very significant, however if this peak is correlated with the latency observations, a pattern can be seen. All the key-value stores has a drop-in latency from the initial throughput target of 1 million operations per second, to 5 million operations per second after which it remains almost constant see (todo). Not the graph you might expect when measuring throughput and latency. At higher throughput should expect the latency to increase, as contention and the available hardware capacity decreases, not what is observed here.

10.2.1.1 Possible cause

My hypothesis are that this is caused by the CPU cache. The reason the latency drops the higher the throughput rate is, is likely caused by that at lower throughput rates it takes more time for the CPU cache to be loaded with entries from the key-value store. When the CPU cache mostly contains entries from the key-value store, the likelihood of finding an entry in cache reaches a maximum percentage. Which supports that the latency will flatten out at a certain throughput target, as seen in (todo). The counterargument to this is that all entries were preloaded in phase 2 and should therefore already reside in cache. However, that would only be the last entries inserted in the preload phase, and the distribution is Zipfian (see section 0), which means that some entries are more popular than others, and over time these entries will gradually dominate the entries in cache. As they are the most frequently accessed, which at a certain point will depend on the cache size, they will constitute some maximum percentage of the entries that reside in cache, at which point the increase in performance will end, and remain constant for the rest of the execution.

10.2.1.2 Possible solution

If this is correct it means that the latency measurements at lower throughput are incorrectly skewed from what you would expect in real world conditions and these are the conditions the evaluation framework is meant to simulate. To improve upon this, a warm up phase could be introduced. After preloading and prior to execution, the experiment could run a set number of operations to ensure that the cache is adequately populated with representative entries from the key-value store.

10.3 The theoretical use cases

The theoretical use cases presented in section (TODO!) serves as a proof of concept for the evaluation framework. Selecting an applications specific interaction characteristic with the key-value store, and then running experiments at these settings, resulted in a set of graphs that shows the performance trade-offs of the different key-value stores. This evaluation framework could be developed further to be a useful benchmarking and decision-support for developers.

10.4 Is concurrency better

The initial hypothesis was as follows:

- My hypothesis is that depending on applications' throughput demand, there can exist a point at which nonconcurrent key-value store outperforms a concurrent key-value store on some or all performance metrics.

The results of the experiments confirm this hypothesis, naturally limited to the key-value stores tested in this implementation. The hopscotch and Google dense hash key-value stores did generally outperform Libcuckoo on all performance metrics measured, up to their respective maximum throughputs. Libcuckoo did, at its best energy per operation metric, not outperform hopscotch. Libcuckoo did of course outperform all serial key-value stores when it came to maximum throughput, but at the throughput demands lower than 10 million operations per second, hopscotch outperform all other key-value store implementations on all performance metrics.

10.5 Limitations of simulation

Any benchmark or evaluation framework have some limitations in how realistic they simulate real world use. This evaluation framework has taken great effort, to as closely as possible simulate a real world applications use of a key-value store. However, since the evaluation is separate from the application, in that the applications memory and CPU use is subtracted from the evaluation, the evaluation framework cannot truly represent how a key-value store will perform in use. This can only be achieved by testing a key-value stores implementation within the application. The reason for this is best illustrated by an example: Take the observations that link the number of entries loading cache and the latency discussed in section 10.2. Where the size of the cache was relevant to the minimum latency, if an application was running this key-value store, it would reduce the amount of available cache as it would naturally use some portion of it, for the work it does that is not related to the key-value store. However, taking this into account, the evaluation framework still gives the best possible representation of a key-value stores performance, given these limitations.

11 Future work

11.1 Additional interfaces

This framework presents many future research opportunities. It would be very interesting to test even more key-value store implementations, to test the performance of lock free[6][9][10] versus lock based[7][5] concurrency implementations, and a tree based key-value stores[3] to see if it performs well on some performance metrics.

11.2 Space efficiency

Space efficiency measuring of the different key-value stores is a performance metric that can be valuable especially when evaluating Google SparseHash[13] which showed relatively mediocre results in the experiments. However, since Google SparseHash is designed to be primarily space efficient, its true benefits cannot be evaluated until the space efficiency metric is added.

11.3 Custom trace generator

To improve the evaluation framework, a custom trace generator is needed, which randomly can generate any key type at the scale needed to test for longer durations to get better accuracy and generating different distributions which might be a difficult algorithmic problem.

12 Conclusion

This implementation of the MELT key-value store evaluation framework has some limitations and issues that require improvement. However, the concept of a multidimensional evaluation of key-value stores have provided interesting results and has potential to be a relevant framework for evaluating different key-value stores for specific applications. In addition to this, the results of the experiments support the claims of my hypothesis. There seems to be a range of throughputs where serial key-value stores outperform concurrent ones.

13 References

[1] C. Imes, L. Bergstrom, and H. Hoffmann, “A Portable Interface for Runtime Energy Monitoring,” in *International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 968–974.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” *Proc. 1st ACM Symp. Cloud Comput. - SoCC ’10*, pp. 143–154, 2010.

[3] Y. Mao, E. Kohler, and R. Morris, “Cache Craftiness for Fast Multicore Key-Value Storage,” in *european conference on Computer Systems (EurpSys)*, 2012, pp. 183–196.

[4] “Java ConcurrentMap.” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentMap.html . [Accessed: 01-Jun-2017].

[5] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang, “Mega-KV: A Case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores,” in *Vldb*, 2015, pp. 1226–1237.

[6] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, “Algorithmic improvements for fast concurrent Cuckoo hashing,” *Proc. Ninth Eur. Conf. Comput. Syst. - EuroSys ’14*, pp. 1–14, 2014.

[7] M. Herlihy, Maurice; Shavit, Nir; Tzafrir, “Hopscotch Hashing,” in *Proceedings of the 22nd International Symposium on Distributed Computing*, 2008, pp. 0–15.

[8] O. Shalev and N. Shavit, “Split-ordered lists: Lock-free extensible hash tables,” *J. ACM JACM*, vol. 53, no. 3, pp. 379–405, 2006.

[9] D. Zhang and P.-Å. Larsen, “LHlf: lock-free linear hashing (poster paper),” in *PPoPP*, 2012, pp. 307–308.

[10] Y. Liu, K. Zhang, and M. Spear, “Dynamic-sized nonblocking hash tables,” *Proc. 2014 ACM Symp. Princ. Distrib. Comput. - Pod. ’14*, pp. 242–251, 2014.

[11] C. Hashing, “Cache-Oblivious Hashing,” in *PODS’10,* 2010, pp. 297–304.

[12] S. Richter, V. Alvarez, and J. Dittrich, “(P1) A seven-dimensional analysis of hashing methods and its implications on query processing,” in *Proceedings of the VLDB Endowment*, 2015, vol. 9, no. 3, pp. 96–107.

[13] “sparseHash GitHub,” 2017. [Online]. Available: https://github.com/sparsehash/sparsehash. [Accessed: 19-May-2017].

[14] “Intel performance counter monitor.” [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor. [Accessed: 29-May-2017].

[15] “Heartbeats-simple Github.” [Online]. Available: https://github.com/libheartbeats/heartbeats-simple. [Accessed: 23-May-2017].

[16] V. M. Weaver *et al.*, “Measuring Energy and Power with PAPI,” in *Parallel Processing Workshops (ICPPW)*, 2012.

[17] “Heartbeats Github.” [Online]. Available:

https://github.com/libheartbeats/heartbeats. [Accessed: 29-May-2017].

[18] C. Imes, D. H. K. Kim, and M. Maggio, "POET : A Portable Approach to Minimizing Energy Under Soft Real-time Constraints," 2015, pp. 75–86.

[19] "YCSB Github." [Online]. Available: https://github.com/brianfrankcooper/YCSB/wiki. [Accessed: 22-May-2017].

[20] J. Ousterhout *et al.*, "The RAMCloud Storage System," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 1–55, 2015.

[21] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51[1] R. P, no. 2, pp. 122–144, 2004.

[22] "Libcuckoo Github," 2013. [Online]. Available: https://github.com/efficient/libcuckoo. [Accessed: 21-May-2017].

[23] "GSL - GNU Scientific Library." [Online]. Available: https://www.gnu.org/software/gsl/. [Accessed: 25-May-2017].

[24] A. Measday, "csoft general purpose library," 2016. [Online]. Available: http://www.geonius.com/software/#liberal. [Accessed: 25-May-2017].

[25] "The System Interfaces volume of POSIX.1-2008," 2016. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html. [Accessed: 24-May-2017].

[26] "Energymon Github." [Online]. Available: https://github.com/energymon/energymon. [Accessed: 24-May-2017].

[27] "CityHash Github." [Online]. Available: https://github.com/google/cityhash. [Accessed: 25-May-2017].

[28] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3 : Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013, no. April, pp. 1–14.

[29] "Hopscotch-map Github." [Online]. Available: https://github.com/Tessil/hopscotch-map. [Accessed: 01-Jun-2017].