How Block Cache Eviction Policies affect the I/O Performance of Applications with Structured Data Access Patterns

Gurer Ozen

A Thesis in the Field of Information Technology
for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

February 2018

Abstract

The Block Cache layer of an operating system stores a subset of recently accessed disk blocks in the main memory to avoid slow disk operations. The performance gain is relative to the likelihood of subsequent accesses for the cached blocks. The eviction policy decides which cached blocks are evicted, and has a huge impact on the system performance.

I have developed a testing framework to evaluate different eviction strategies under a cache simulation by using the real world pre-cache access records. The framework also replays post-cache accesses on a real solid state disk to get a more realistic measurement of the performance.

I have compared the 2Q, ARC, BRRIP, CAR, CLOCK, CLOCKPRO, LRU, Random2, RRIP, SpatialClock algorithms under various test captures. In addition to classic sequential and random categories, I have focused on database access patterns for B-Tree structured disk files.

Testing with disk replay was time consuming but revealed that the ordering of evicted blocks have a significant performance impact. The reduction of number of I/O requests is not the sole metric for the block cache performance, and the comparison studies must include the operating time on real hardware.

The randomized algorithms did not perform well for this reason, and the 2Q turned out to be the most competitive eviction strategy for the tested workloads.

Acknowledgements

I would like to thank my thesis director Stratos Idreos for his support and guidance on this work, and his help in focusing my general curiosity about the file system performance into a very specific and important subject.

My research advisor Jeff Parker provided great feedback during the proposal period, which equipped me with clear direction and a well though out plan. I could not finish this project on time without all this planning.

My wife Damla Nur Ozen encouraged me to begin this masters program and supported me at every step.

The music of the 70s rock-soft group Blue Jean Committee deserves credit for getting me through late night debugging sessions.

Contents

Chaper 1: Introduction

Storage

Storage retains the digital information being used by the computation, and is a fundamental part and function of computers. There are many different storage devices and technologies, but they can be categorized by the following major characteristics:

Capacity The total amount of information that can be stored on the device at any given time. The capacity of a device is measured in bytes. It is generally divided into granular units where each unit has a location (address) and can be read or written individually. Devices which use a recording media might provide a read-only capacity. Also, certain devices such as Solid State Disks (SSDs) might have a limit on how many times information can be overwritten on each location. The SSDs also have to erase the unit before writing the new content. The granularity for erase is generally larger than the write operation.

Throughput The rate at which information can be read from or written to the device. The throughput of a device is measured in bytes per second. The write throughput is slower than the read throughput as it involves physical state changes. If the locations are not accessed in a sequential order, certain devices, such as the physical disks which need to move the magnetic head to a different location, could fail to reach their maximum throughput.

Latency The time it takes to access a particular location. The latency is usually measured in nanoseconds or milliseconds. Similar to the throughput, it could be different for the reads and the writes, and it could be affected by the access order.

Energy use The energy use for the physical storage is generally based on the accesses as they cause physical movements. The energy use of the electronic circuit based storage is generally dependent on the operating frequency of the circuit. Operating at a higher frequency requires more power. Certain volatile devices, such as the computer memory, might need to draw power constantly for retaining the information whereas others might hold the information until power comes back.

High capacity, high throughput, low latency and low energy use are desirable. Since these characteristics are often in conflict with each other, it is very hard to accomplish. Low latency requires circuits to remain active to avoid power up delays, and this might be costly for large capacities. A device optimized for large data transfers to gain high throughput might lose latency when handling a large number of small data transfer requests, because the operation overhead exceeds the amount of actual work.

This conflict resulted in a hiearchical storage model commonly found in computers. The primary storage is the main memory (CPU registers, CPU cache and the RAM) which has low latency, high throughput but also a high energy use and low capacity. The secondary storage is the disks (Physical spinning disks, SSDs, CD-ROM) and other persistent mediums which have high capacity, lower energy use (relative to capacity) but also have a higher latency and lower throughput.

Software support is critical for making the best use of this hiearchy for data sets larger than the main memory and obtaining the optimum performance.

File Management

Storage hardware could be used by many different applications and it is the Operating System's (OS) responsibility to share storage resources efficiently. The OS also provides additional storage services such as space management, isolation, redundancy, compression, and encryption.

Modern OSes have a File Management subsystem to manage the Storage and it is implemented as a series of layers:

Filesystem This layer presents the storage to applications as a tree-like namespace where each node is a directory and each leaf is a file. Files can be arbitrarily large and information can be read from or written to any position in the file. A filesystem implementation translates this namespace into linearly addressed fixed size blocks. How files are laid on the block addressing space and what kind of metadata is stored is not visible to the applications.

Block Cache Disk storage has a higher latency compared to the main memory. It is beneficial to use the available free memory as a cache for the recently accessed disk blocks. This will save time if that block is used again by the filesystem layer. Delaying the writing of modified blocks can also be beneficial as it might allow combining write operations to adjacent blocks or to the same block. Modern operating systems have unified implementations which combine the general memory management and the block cache. The blocks are mapped into the memory pages and managed together with the other application and kernel data.

I/O Scheduler The blocks which are going to be written into or read from a storage device are entered into a queue. A scheduler might sort these requests into a more efficient order for devices which has a slower non-sequential access such as physical disks. Time savings from improved order is usually greater than the extra delays introduced by the scheduling operation for such devices.

Device Driver At the bottom of the subsystem, there is the device specific driver which turns block requests into the actual hardware commands. The hardware can also have its own layers, either as a firmware or implemented as a part of the circuitry, which does internal caching, scheduling or even mapping. SSDs, for example, internally map blocks to different addresses when that location is over its overwrite lifetime.

This abstraction is highly desirable for robustness and flexibility. A new type of device can be used by applications simply by providing a device driver. A new algorithm for caching or scheduling can be implemented as a simple module.

Yet the evolution of the storage hardware and the new software optimization techniques often break this abstraction by requiring the knowledge of other layers' internal operation. The block cache would be more efficient if it has the knowledge about which blocks will be needed next by the applications. If a device features some internal optimizations, repeating them at the filesystem or the scheduler level can be unnecessary or even detrimental to the performance. A filesystem can layout files into the blocks in a more efficient way if it has the knowledge about how the underlying device deals with the addressing.

There are two general ways to solve this problem: the interfaces between layers could be extended to pass more information or the missing information can be predicted from previous operations. Assuming the applications are trustworthy, extending the interface provides highly reliable information. However it also requires modifications in all applications, and designing such an interface involves a trade-off between generality and usefulness.

The prediction needs to be done in real time with limited computational resources. It is usually based on heuristics. Results are less reliable and often require fine tuning for expected workloads. However, they are also quick to compute and they do work well in practice. Such heuristics are commonly used in production OSes.

Block Cache Eviction

The world of the storage is extremely rich, and each component briefly described here have their own research areas.

This thesis focuses only on the Block Cache layer and further limit its scope to the selection process for eviction of blocks. Despite being a very small piece in the whole subsystem, eviction has a major impact on the performance. Keeping the block that is going to be needed in the memory or combining multiple writes within a short period before the eviction, can prevent hardware operations which would be several orders of magnitude more expensive.

The Cache has a very minimal interface. It takes requests to read or write some blocks, and hopefully sends a smaller number of requests to the I/O layer. Eviction could happen under three scenarios:

1. When the cache is full, and we are reading a new block from the I/O layer. In this case, the eviction policy might decide to throw away an old block and keep the newly read block in the memory. The modified blocks must be written back to disk when they are being evicted.

2. When there is a memory pressure, and the OS forces the cache to reduce its size.

3. When there are modified blocks in the cache. Delaying the writes certainly improves the performance, but leaves the system vulnerable to data loss in case of a power failure. To prevent this, modified blocks are periodically

flushed to the disk based on conditions such as the time after the last flush, the age of the blocks, percentage of the modified blocks in the cache, the current I/O load and others. A flush can also be forced by a user or an application using a system call like fsync().

The challenge here is the difficulty of predicting when a given block will be accessed next, if ever. This information can be guessed from the previous access patterns or might be given as a hint to the cache by the applications themselves.

There are some common patterns:

Sequential The application reads or writes a file from beginning to end. A videoplayer playing a media file, or a file compression tool which read a file and write the compressed version of it are some examples to this pattern. Note that even if the file is accessed sequentially, blocks are usually not accessed in a linear order at the underlying device. Optimizing that part is the job of the filesystem layout, I/O scheduler and sometimes the device itself. From the perspective of the block cache, there are two common optimizations: Subsequent blocks could be fetched from the I/O layer without being asked for (this feature is called readahed), and become already available when the application asks for them. In the second optimization method, the fetched blocks could be prevented from caching or associated with a higher eviction priority if the application is not going to use them in the near future.

Random The application is expected to read or write into a file in random order. While this category is traditionally named random, a specific distrubition such as uniform distribution is not assumed here. It is more like a non-sequential category which cover many different patterns. A web browser or a desktop manager application are some examples.

B-Tree It is common amongst the database applications which manage huge datasets to use a B-Tree structure for the on-disk represantation of the database. This is clearly a distinct pattern where generally the root branches are accessed more frequently than the leaf nodes.

Hashtable Some data store application prefer this structure because the lookup and retrieval usually involve one access to the file. A good hash function should create access patterns very close to a uniform distribution.

There is a huge body of research on detecting access patterns and optimizing them for underlying hardware. Many proposed techniques are used in major OSes. However, the focus is only on a small subset of patterns. As an example, the POSIX fadvise() system call, or the Windows CreateFile() system call, which create a file with a declared access pattern, only support the sequential and the random flags. The effect of the random flag is usually just limited to disabling the read-ahead.

Research Questions

In this thesis, I answer the following questions within this context:

1. What is the performance of the applications with various data access patterns under various eviction policies?

2. Is there a performance difference amongst the random access applications like a general desktop application, or a B-Tree database with respect to the eviction policy?

3. How much performance is gained for SSDs by using special algorithms exploiting the device IO characteristics? Would they offer significant benefit over LRU based or adaptive eviction algorithms?

Chapter 2: Prior Work

Testing

The Block Cache is deeply embedded in the operating system. There is usually a unified memory management system which handles block caching, memory mapping of files, swap management and general memory management. That makes implementing and trying various algorithms difficult, because many code changes and interaction with unrelated subsystems are required. Collecting application traces and running them under a simulation with various algorithms is a much more feasible method. However, the trace-driven simulation may not emulate storage hardware precisely and may produce slightly different results than real world usage. A good solution to improve the results is to capture simulation after-cache disk access traces and run them on the real storage device using a work load player (Kim et al., 2012).

LRU Heuristics

Principle of locality states that the data accesses within a process tend to bunch together. The Least Recently Used (LRU) policy selects the block for replacement which has not been accessed for the longest time. The blocks are maintained in a list sorted by the last access time. Every time there is an access, that block is moved to the front of the list. The blocks are then evicted from the back of the list.

The implementation overhead for a true LRU is high, because every access causes a change in the ordering of blocks, and there are also synchronization costs when the accesses are concurrent.

A common feature amongst the Memory Management Units (MMUs) of CPUs is the access bit associated with each memory page corresponding to a block. That bit can be cleared by the operating system, and automatically set by the hardware when there is a read access. There is a similar dirty bit for the write accesses, set when the block is modified. When the dirty bit is clear, we can remove a block from the cache. When the dirty bit is set, we must write the block back to the secondary store. These hardware features are used to implement LRU approximations with very low overhead.

CLOCK

The CLOCK is one of the earliest LRU approximations. In this algorithm, all blocks are arranged in a circular list resembling a clock. The hand of the clock points to the oldest block in the cache. When there is a need to evict a page, the MMU access bit of the first pointed block is checked. If it is not set, that block is reclaimed and a new block is brought in its place with a zero access bit. If the bit is set, it is reset to zero and the hand is advanced to the next block until a block with a zero bit is found. This algorithm approximates the LRU with the least amount of overhead.

However, this basic LRU approximations don't work well when the application scans, reading or writing a large number of sequential blocks, larger than the cache size. All blocks are evicted before a second access in this scenario.

2Q

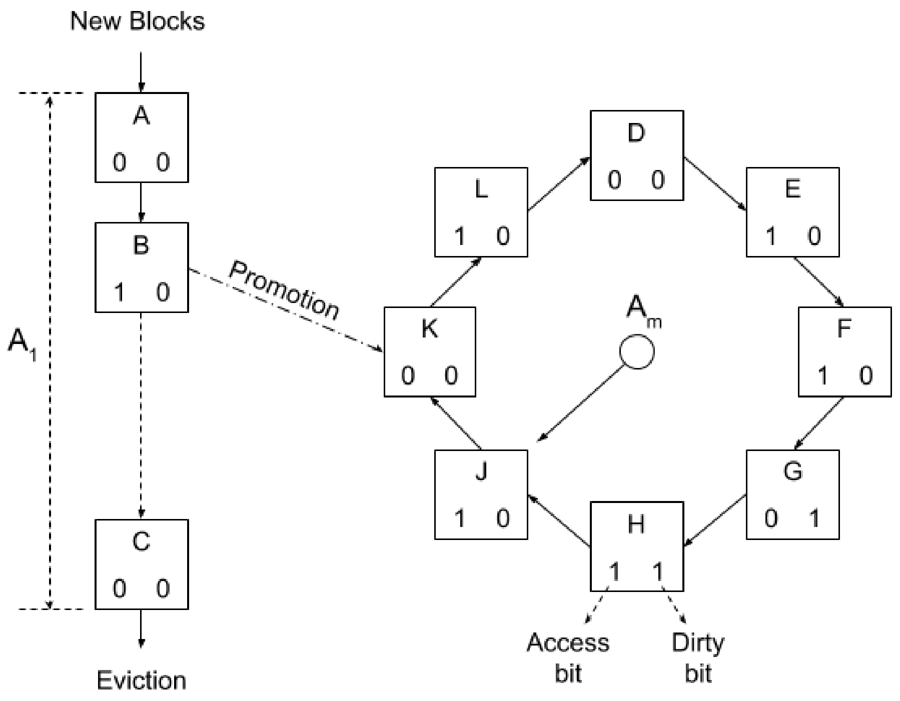

Figure 1: The two queues of the 2Q algorithm.

An important improvement over the CLOCK is the 2Q algorithm. There are two queues in this algorithm: The A_1 queue contains the blocks which are accessed once, and maintained in the First In First Out (FIFO) order. The A_m queue contains the blocks which are accessed multiple times, and maintained as an approximate LRU queue using the CLOCK algorithms (figure 1). A newly accessed block is first put into the A_1 queue, and evicted if it reaches the head of the queue without being accessed again. If the block has been accessed again, it is moved into the A_m queue. This prevents transient blocks from evicting frequently accessed blocks (Johnson and Shasha, 1994).

Linux kernel uses a custom variant of the 2Q algorithm.

CLOCKPRO

The CLOCKPRO algorithm is another improvement over the original CLOCK. Blocks are similarly organized in a circular list. Each block has a type of a Hot block (recently accessed), a resident Cold block (content is in memory), or a non-resident Cold block (just a reference to the content). Non-resident Cold blocks serve as the recent access history. The list can hold items two times the cache size with a half of them dedicated to the history information. Each block also has a Test flag indicating if the block is in test period.

There are also three pointers. Hand_{hot} points to the block with the largest recency. $\text{Hand}_{\text{cold}}$ points to the last resident Cold block. $\text{Hand}_{\text{test}}$ points to the last cold block in test period.

$\text{Hand}_{\text{cold}}$is used to search for a resident cold block for replacement. It moves until it finds an eligible block to evict, and then stops at the next resident cold block. If the pointed block is accessed since last time; if it is in testing period, it is made a Hot block, moved to the tail of the list and $\text{Hand}_{\text{hot}}$is moved, if it is not in testing period, it is just moved to the tail of the list. If the pointed block is not accessed since last time, content is evicted, and entry is also evicted if the block is not in testing period.

$\text{Hand}_{\text{test}}$ cleans up the non resident blocks as it moves.

Hand_{hot} resets the access bits of hot blocks it sees, and turn unaccessed blocks into cold blocks. If number of non resident clock blocks is larger than cache size, $\text{Hand}_{\text{cold}}$ is also moved. Hand_{hot} also cleans up any non resident blocks just like $\text{Hand}_{\text{test}}$ as it moves over them.

CLOCKPRO has no operation for cache hits other than updating the access flag. A cache miss could move $\text{Hand}_{\text{cold}}$ to reclaim a block if cache is full. If block is not seen before, it is inserted as a Cold block. If it is seen before, it is inserted as a Hot block in testing period, and Hand_{hot} is moved (Jiang et al., 2005).

RRIP

The Re-Reference Interval Prediction (RRIP) algorithm considers the block order as the order of re-reference. The last block to evict is predicted to have a near-immediate re-reference whereas the first block to evict is predicted to have a distant re-reference. Each block has an M-bit Re-Reference Prediction Value (RRPV). A zero RRPV value indicates that this block is predicted to be re-referenced in near immediate future, and a maximum value of $2^M - 1$ indicates that the block is predicted to be re-referenced in distant future.

The RRIP inserts new blocks with a long but not distant re-reference interval. The suggested value is $2^M - 2$. The intuition behind this is to prevent transient blocks from polluting the cache but also to give time to algorithm for improving the prediction.

The RRPV value is set to zero upon access. The eviction process scans the block list from left to right until a block with the maximum value is found. If there is no such block, all RRPV values are incremented by one and the search continues.

In practice, $M = 2$ bits gives good results and scan resistance (Jaleel et al., 2010).

CFLRU

The solid state disks must erase the entire sector before updating a single block in the sector. The write operation is expensive for this reason. The sectors also have a limited lifetime before they worn out from writes. Therefore avoiding the write operations can be more beneficial than avoiding the reads.

One of the earliest policies for SSDs is the Clean First LRU (CFLRU) algorithm which prefers clean pages over dirty pages for the eviction. Since this could fill the cache solely with dirty blocks and reduce the read performance, the block list is divided into a working region of most recently used blocks, and a clean-first region of least recently used blocks. The algorithm first evicts clean blocks from the clean-first region in the LRU order, and then continues eviction with the dirty blocks from the clean-first region in the same LRU order. Selecting a good partition size is important for the performance (Park et al., 2006).

LRU-WSR

The LRU Write Sequence Reordering (LRU-WSR) algorithm is similar to the CFLRU, but instead of always preferring the clean blocks, it gives dirty blocks a one time chance to avoid eviction. A cold bit is associated with each block. During the scan, if the least recently used block is clean, it is immediately evicted. If the block is dirty and the cold bit is zero, the block is moved to the head of the list, and the cold bit is set. If the block is dirty and the cold bit is set too, it is evicted (Jung et al., 2008).

FAB

The Flash Aware Buffer (FAB) algorithm cluster adjacent blocks as a group to mimic to physical sector layout of the SSDs. These groups are maintained in an LRU list. The eviction scans the list from the tail to the head until a group with the full number of blocks is found. If no such group is found, then the group with the maximum number of blocks seen during the scan is selected for the eviction. This strategy makes the best use of physical erase-before-write cycle of SSDs by avoiding erasing an entire sector to just write a single block (Jo et al., 2006).

Spatial Clock

The Spatial Clock is another algorithm specifically designed for SSDs. It uses classic CLOCK algorithm and memory reference bit for the LRU approximation, but the blocks are sorted by their logical sector numbers rather than by access time. Therefore spatial adjacency is also taken into account for the eviction decision. Maintaining blocks in the sorted order is an extra cost, but it is done only when inserting new blocks, and relatively rare when the cache hit ratio is high (Kim et al., 2012).

Adaptive Algorithms

ARC

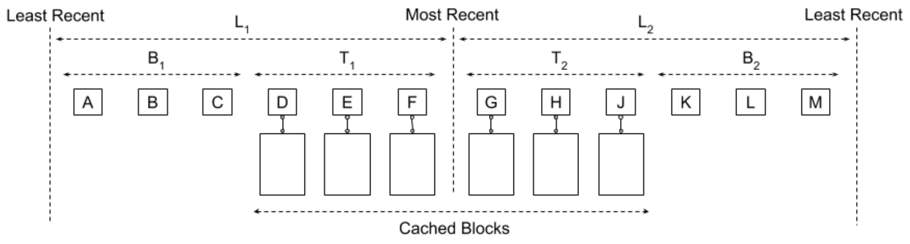

Figure 2: The LRU lists of the ARC.

The Adaptive Replacement Cache (ARC) algorithm maintains two lists called L_1 and L_2 in the LRU order (figure 2). The L_1 maintains references to the blocks that have been seen only once, and the L_2 maintains references to the blocks that have been seen at least twice, recently. It can be said that the L_1 captures recency while the L_2 captures the frequency. The L_1 is further divided into the top half T_1 which are the most recent blocks, and the bottom half B_1 which are the least recent blocks. The L_2 is similarly divided into the T_2 and B_2 halves.

The blocks in the T_1 and T_2 have their block contents cached in the memory.

The B_1 and B_2 only maintain references. The total size of the two lists is $2c$ if there is space for c number of blocks in the cache. If there are p blocks cached in the L_1, then the L_2 contains $c-p$ cached blocks. ARC dynamically adjusts the ratio of caching in each list to adapt to the changes in recency and frequency of the workload (Megiddo and Modha, 2004).

If x is the requested page, the algorithm,

1. If $x \in T_1 \cup T_2$ (hit): Moves x to the top of T_2.

2. If $x \in B_1$ (miss, seen once before): Adapts $p = \min(c, \max(|B_2|/|B_1|, 1))$, replaces a block using the algorithm below, moves x to the top of T_2 and stores the contents of the block in the cache.

3. If $x \in B_2$ (miss, seen more than once before): Adapts $p = \min(c, \max(|B_1|/|B_2|, 1))$, replaces a block using the algorithm below, moves x to the top of T_2 and stores the contents of the block in the cache.

4. If $x \notin L_1 \cup L_2$ (unseen miss): Checks conditions below, and then puts x at the top of T_1 and stores the contents of the block in the cache.

   (a) If $|L_1| = c$:

      i. If $|T_1| < c$: Deletes the LRU block of B_1, replaces a block using the algorithm below.

      ii. Else : Deletes the LRU block of T_1 and removes its contents from the cache.

   (b) If $|L_1| < c$ and $|L_1| + |L_2| \geq c$: Deletes the LRU page of B_2 if $|L_1| + |L_2| = 2c$, and replaces a block using the algorithm below.

The replacement algorithm is:

1. If $|T_1| \geq 1$ and $((x \in B_2$ and $|T_1| = p)$ or $|T_1| > p)$ then move the LRU block of T_1 to the top of B_1 and remove its contents from the cache.

2. Else, move the LRU block in T_2 to the top of B_2 and remove its contents from the cache.

CAR

Inspired by ARC, the Clock with Adaptive Replacement (CAR) algorithm uses two circular "CLOCK" lists, T_1 and T_2 for blocks which are in the cache, and two FIFO lists, B_1 and B_2 for the records of the recently evicted blocks from the corresponding T lists. The goal is to move data structure updates of ARC from the read/write access time to the eviction time (Bansal and Modha, 2004).

1. A cache hit in CAR just updates the access bit as in the original CLOCK algorithm.
2. A cache miss sets the reference bit to zero, and:
   (a) If the block is in B_1: Moves block to the tail of T_2, increases the target size of T_1.
   (b) If the block is in B_2: Moves block to the tail of T_2, decreases the target size of T_1.
3. The cache eviction replaces from T_1 if its size is larger than the target, otherwise replaces from T_2. This eviction is similar to the original CLOCK algorithm except that the record for the evicted block is pushed to the relevant B list.

Randomized Algorithms

Random2

A randomized way to implement an LRU is to take two random choices for eviction and use the LRU between them. This approach provides a performance close to an LRU, and surpasses it for large caches. The same idea also holds with LRU approximations (Luu, 2014). The mathematical concept behind this is that randomly distributing n balls into m bins will result in maximum number of balls in any bin of $O(\log n)$ with high probability, but if we choose least loaded of k random bins, the maximum is now $O(\log\log n/\log k)$ which is $O(\log\log n)$ even for $k = 2$ and each increment of k reduces the load only by a constant factor (Mitzenmacher et al., 2000).

BRRIP

The Bimodal RRIP (BRRIP) algorithm is an improvement over the RRIP which inserts majority of new blocks with $2^M - 1$ prediction value and some of them with $2^M - 2$ prediction value with a low probability to guard against cache thrashing patterns. Since this is degrading performance for normal patterns, a method called set dueling is suggested to compare two small cache sets with competing policies to dynamically change this behaviour (Jaleel et al., 2010).

Chapter 3: Methodology

I have developed a new test infrastructure (figure 3) for measuring and comparing the performance of eviction algorithms. All of the source code of this framework can be found on web at github.com/meduketto/blockcacheperf and available for general public use under the GNU GPLv2 license.

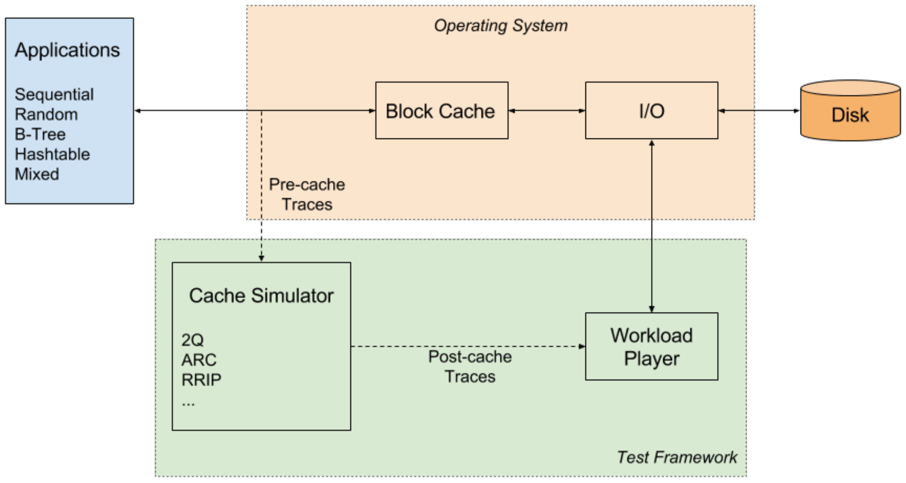

Figure 3: Test Framework.

Collection

I use Systemtap for collecting the application I/O access patterns. Systemtap lets the user write small scripts which could then be loaded into kernel as modules which can tap into any symbols such as functions or variables and take actions before and after their execution. This is a very flexible way to intercept application calls at any kernel subsystem and collect information. Systemtap script accepts parameters to limit its collection to a certain application or a certain directory.

For each test, a well defined set of operations are executed while the Systemtap script is collecting the Filesystem and Block I/O layer access information. A separate program then processes the output of script and generates a list of read and write operations to the storage sectors as if there is no page cache layer in Linux kernel and the application went directly to disk for each operation. This could result in non sequential accesses even for reading sequentially from the same file depending on how the filesystem lays out the file blocks into the disk blocks. This is more realistic as almost always there is a filesystem between the application and the storage unit. Sectors here are defined as 512 byte blocks as that is what Linux kernel chooses as the basic I/O unit.

I have developed some scripts to simulate an Sqlite database for different workloads. This is used for generating B-Tree access patterns.

Simulation

I have developed a cache simulator which mimics a block cache environment and generates the post cache accesses for 2Q, ARC, CAR, CLOCK, CLOCK-PRO, FAB, LRU, RRIP, BRRIP, Random2, and SpatialClock algorithms. The simulator takes the output of collection phase and generate a post cache output for each of the selected algorithms. The reduction of the number of read and write accesses can be seen in this phase.

Linux kernel uses a unified memory management and block cache system called page cache which uses 4 kb page sizes. The simulator maps the sector size to page cache size so the algorithms produce similar results as if they are within the kernel.

The simulator also has many internal checks and some unit tests to catch any problem with the implementation of the algorithms.

Player

I have developed this tool to run any post cache access trace on the raw storage device. This tools avoids any high level layers or caches of the Linux, and goes directly to Block I/O layer to send commands. It tries to send commands in batches to allow for any internal device driver optimization.

The benefit of this tool is to be able to look behind the reduction of requests, and determine the exact runtime of a given post cache output to measure the effects of block ordering if there are any. Since both spinning and solid state disks have performance variances based on the location and the type of access, this step is very important to understand the real performance.

A great care is required for this measurement. It is run directly on the bare metal without any virtualization. All other OS services and applications are stopped during the run with the Linux single user runlevel. Test repeat count is increased until timings numbers are stable. Then each test set is also repeated five times and the lowest number is taken as it conveys the best throughput of the device for this post cache pattern.

Since running such tests create potentially billions of write requests, a spare disk must be used for testing purposes as the tool can easily wear out the lifetime of many blocks.

Hardware

The I/O cache player tests are done with a Samsung 850 EVO 500GB 2.5" SATA III (MZ-75E500B/AM) SSD drive. All tests are run on a AMD Ryzen5 1600 6-core processor with 16GB ram using Debian 9.3 - Linux 4.9.0 kernel.

Chapter 4: Test Results

Full set of input files captured during these tests can be found on web at github.com/meduketto/blockcacheperf/tree/master/data

Each result includes a description of test input, information about cache size and number of I/O operations generated by the test, a table of algorithms with the number their reduced I/O operations and the duration for their replay on the disk, and a figure showing their performance.

The I/O duration is normalized to a single iteration. Each algorithm output is replayed several iterations for each test to measure an error free duration. The whole tests took a little longer than a week to run.

Test 1 - Kernel Compile - 10k blocks

A bash script is executed which downloads the Linux Kernel 4.15.4, unpacks it, configures with default config and compiles with -j8 option for eight thread parallellism.

Cache size is 10k (40 MB), 39526517 sectors read, 2443430 written.

	Read	Written	Time
ARC	218102	284603	4.864
CLOCK	199849	287966	4.933
CLOCKPRO	208026	289128	6.256
2Q	204673	292278	4.870
LRU	222473	284602	5.046
R2	249772	289559	6.905
SPATIAL	207518	286570	6.025
RRIP	209661	285915	4.848
BRRIP	249304	286636	5.419

CAR	207868	285431	5.027

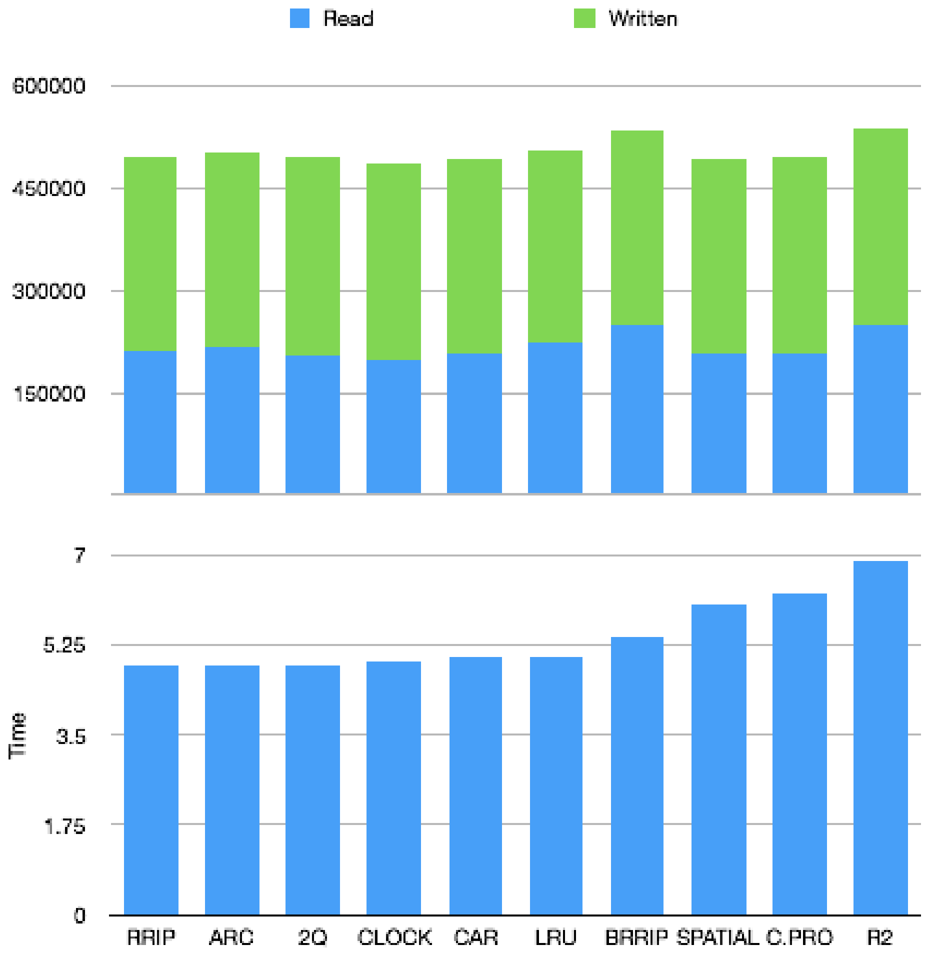

Figure 4: Test 1 - Kernel compile - 10k blocks

The parallel compilation generates an interleave of short sequential patterns as the compiler reads input files and generates the output files which are then feed into the assembler and the linker.

We can immediately observe that the two performance graphs are not directly correlated. Post-cache I/O operation numbers are very close to each other for this relatively simple access patterns. However, SPATIAL and CLOCKPRO algorithms came up last in the time test, almost dropping to the performance of the R2 algorithm which has the highest number of cache misses. Also, BRRIP algorithm which has similarly high cache misses was still faster than them.

This result indicates that the post-cache I/O patterns could have a significant impact on the performance.

Test 2 - Kernel Compile - 2k blocks

Test 1 is simulated and played for 2k cache size.

Cache size is 2k (8 MB), 39526517 sectors read, 2443430 written

	Read	Written	Time
ARC	445644	294163	7.477
CLOCK	451792	294453	7.623
CLOCKPRO	529762	295091	10.094
2Q	488947	295398	7.916
LRU	457784	293662	7.653
R2	1080066	297306	17.220
SPATIAL	458587	295172	8.983
RRIP	443881	293780	7.396
BRRIP	496553	295713	8.192
CAR	444338	295903	7.555

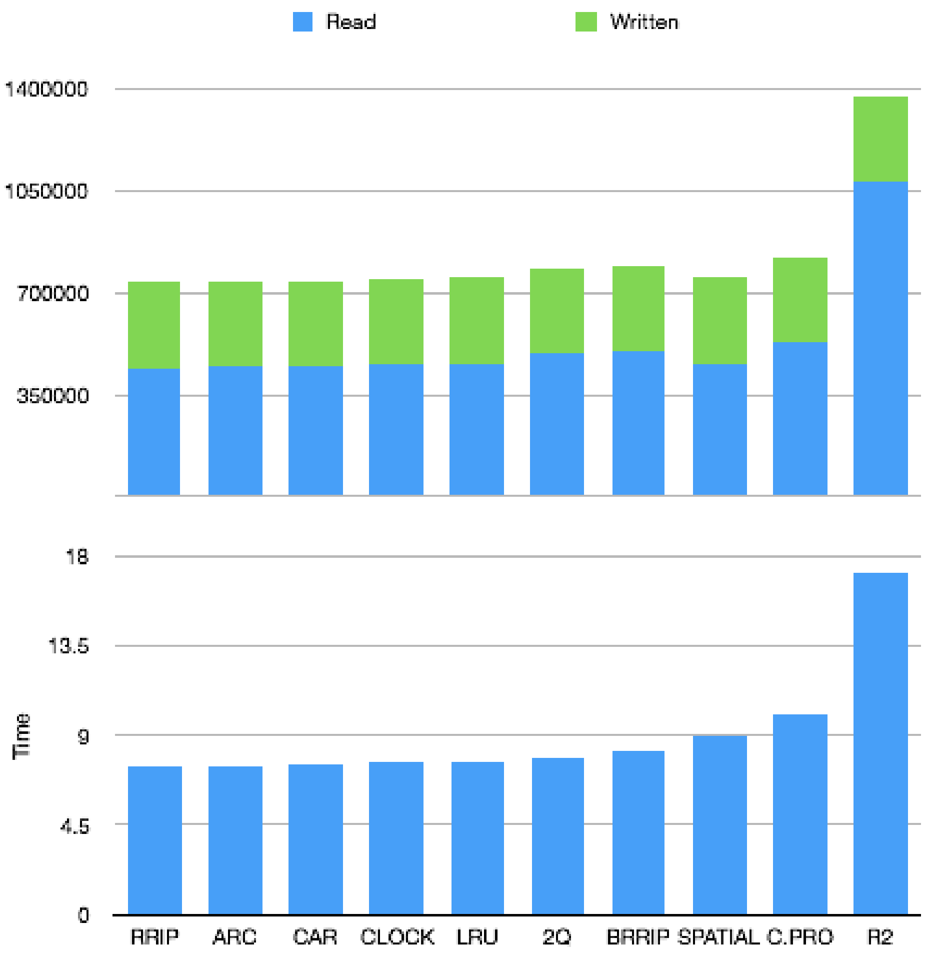

Figure 5: Test 2 - Kernel Compile - 2k blocks

This is basically test 1 with a tighter cache size. The results are very similar too. We can see SPATIAL and CLOCKPRO failing in the time test despite good results for the cache hits.

Test 3 - Sequential File Operations

A large file is copied and then several text search queries are applied on it with the 'grep' command.

Cache size is 10k (40 MB), 5472132 sectors read, 1460956 written.

	Read	Written	Time
ARC	675594	181911	6.718
CLOCK	675605	181944	6.721
CLOCKPRO	675625	181939	7.538
2Q	675594	181910	6.850
LRU	675594	181910	7.015
R2	672383	181832	8.201
SPATIAL	675612	181928	7.618
RRIP	675618	181935	6.574
BRRIP	680852	187171	7.046
CAR	675611	181926	7.045

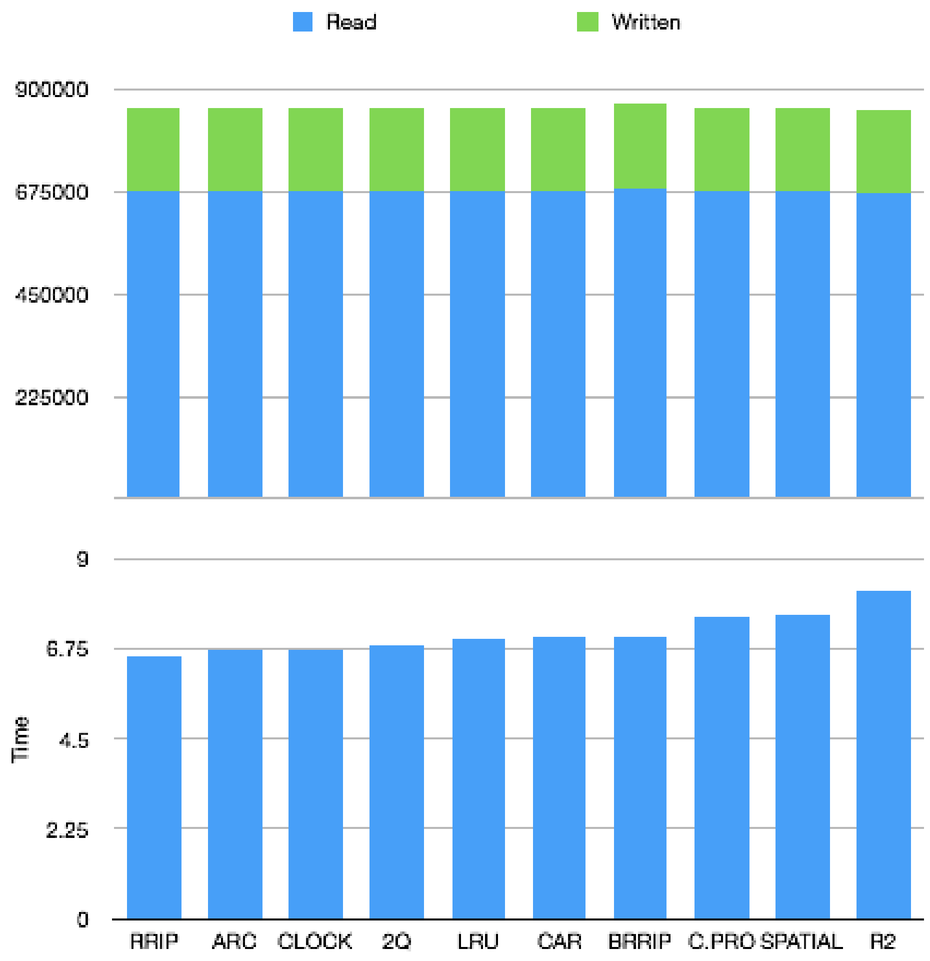

Figure 6: Test 3 - Sequential File Operations

This is a more sequential operation where we scan a large file several times for read and write. This simplest pattern resulted in very close cache miss numbers for all algorithms. Yet CLOCKPRO, SPATIAL and R2 are distinctly separated from others in the time graph.

R2 approximates LRU for recency, but the output ordering is very different than the input order. SPATIAL basically outputs in physical proximity order. CLOCKPRO skips around the CLOCK with different hands. All other algorithms are essentially LRU/CLOCK based, with the optional second chain for frequency, and doesn't alter the output order from access order much. We can speculate that this ordering change might be causing the slowness on the SSD disk.

Test 4 - Desktop Use

Firefox web browser, LibreOffice Writer and some other desktop applications are used for five minutes.

Cache size is 16 (64 MB), 52310 sectors read, 14124 written.

	Read	Written	Time
ARC	7217	1942	0.093
CLOCK	7183	1948	0.092
CLOCKPRO	7164	1949	0.096
2Q	7193	1974	0.094
LRU	7221	1966	0.093
R2	7352	1979	0.098
SPATIAL	7200	1972	0.096
RRIP	7214	1954	0.094
BRRIP	7277	1990	0.094
CAR	7224	2000	0.093

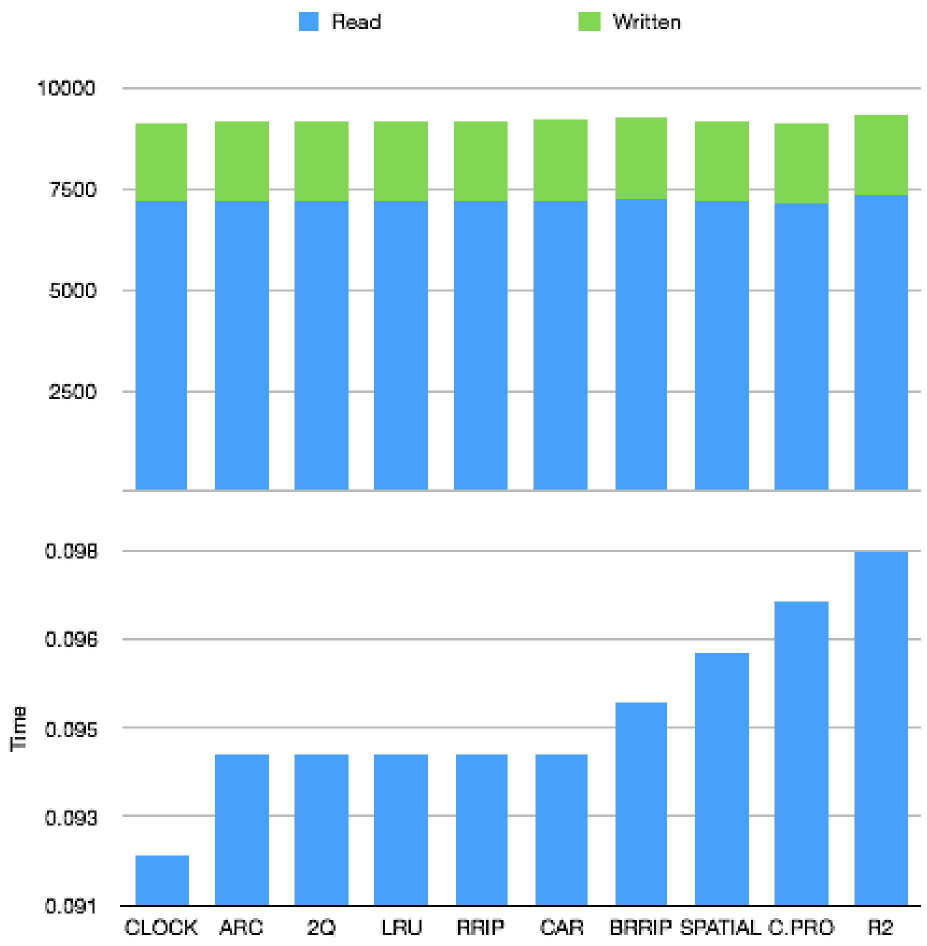

Figure 7: Test 4 - Desktop Use

This is a basic desktop user pattern. We can still see similar performance graphs like the previous tests. However, relative time differences are very small in test to provide a strong evidence.

Test 5 - Sqlite - 90% reads - NoAppCache - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 10k (40 MB), 180019987 sectors read, 4930829 written.

	Read	Written	Time
ARC	9477451	1085441	127.789
CLOCK	9470505	1083163	130.360
CLOCKPRO	9650216	1085733	133.140
2Q	9456862	1076526	127.288
LRU	9467084	1082375	127.453
R2	11110855	1086677	152.493
SPATIAL	9464288	1082660	129.893
RRIP	9464738	1078797	126.283
BRRIP	9469541	1075925	127.158
CAR	9466048	1082338	128.873

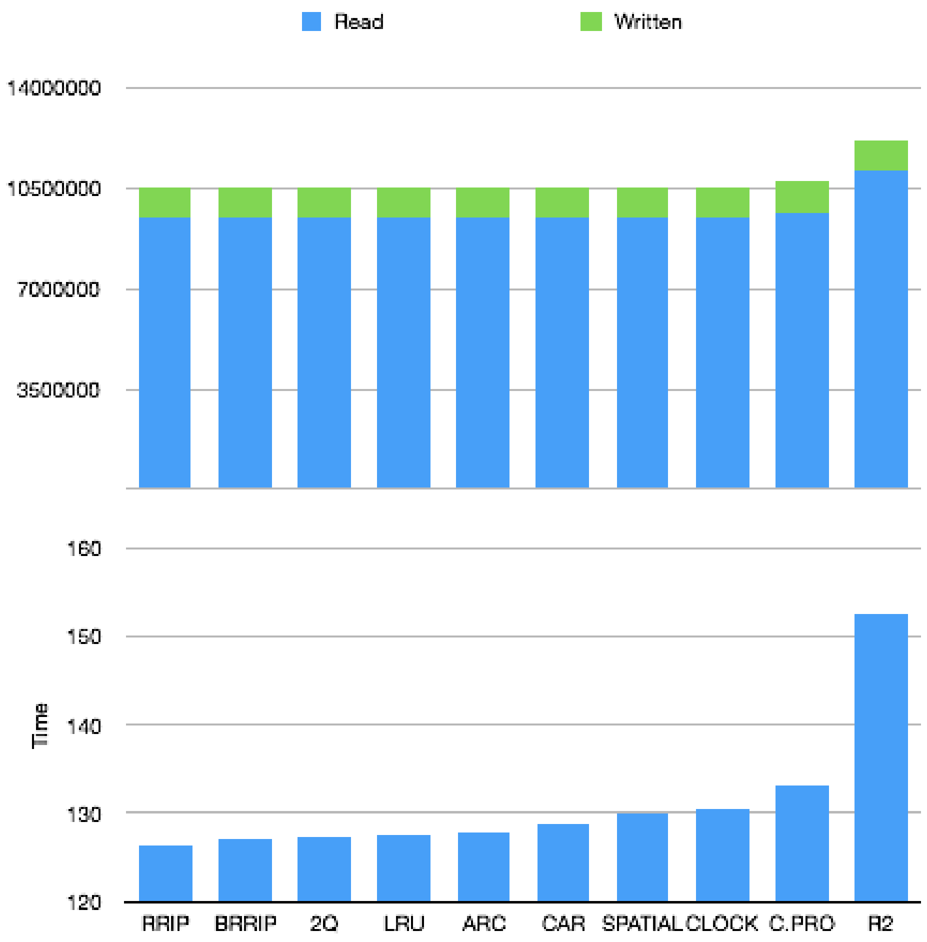

Figure 8: Test 5 - Sqlite - 90% reads - NoAppCache - 10k blocks

Now we are looking at the BTree access patterns of the Sqlite. Since Sqlite's own application cache is disabled, we can see a lot of read operations. For this read heavy load, most algorithms are in the same range. CLOCKPRO is still slightly slower but this time its cache miss count is high as well. The R2 performed badly both in cache misses and time.

Test 6 - Sqlite - 90% reads - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 10k (40 MB), 84264936 sectors read, 4913090 written.

	Read	Written	Time
ARC	9480535	1080772	128.388
CLOCK	9475894	1082137	131.232
CLOCKPRO	9671038	1083504	134.037
2Q	9460568	1049441	127.606
LRU	9477130	1079151	126.588
R2	11173377	1085350	152.967
SPATIAL	9467956	1081361	129.184
RRIP	9469578	1077424	128.361
BRRIP	9467682	1077818	130.833
CAR	9527420	1087077	131.183

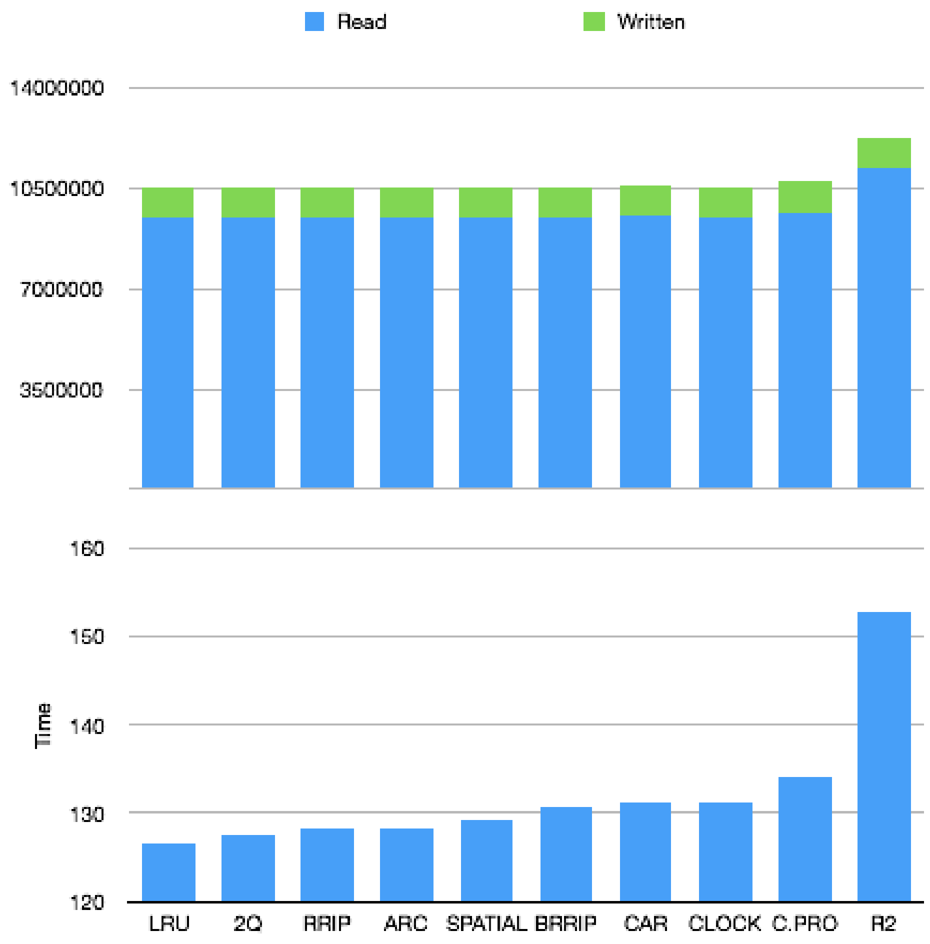

Figure 9: Test 6 - Sqlite - 90% reads - 10k blocks

This is test 5 with Sqlite's application cache enabled which reduced the number of reads. We can observe a similar result. The outliers CLOCKPRO and R2 also have high cache misses in line with their bad performance.

Test 7 - Sqlite - 90% reads - NoAppCache - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 2k (8 MB), 180019987 sectors read, 4930829 written.

	Read	Written	Time
ARC	11639730	1092052	152.981
CLOCK	12588826	1092689	166.402
CLOCKPRO	13091141	1092653	175.624
2Q	9921201	1088064	132.258
LRU	12664434	1092701	165.201
R2	15148560	1093018	209.044
SPATIAL	12769153	1092642	168.950
RRIP	12177127	1092324	160.696
BRRIP	13880125	1091681	184.622
CAR	10758844	1092788	148.213

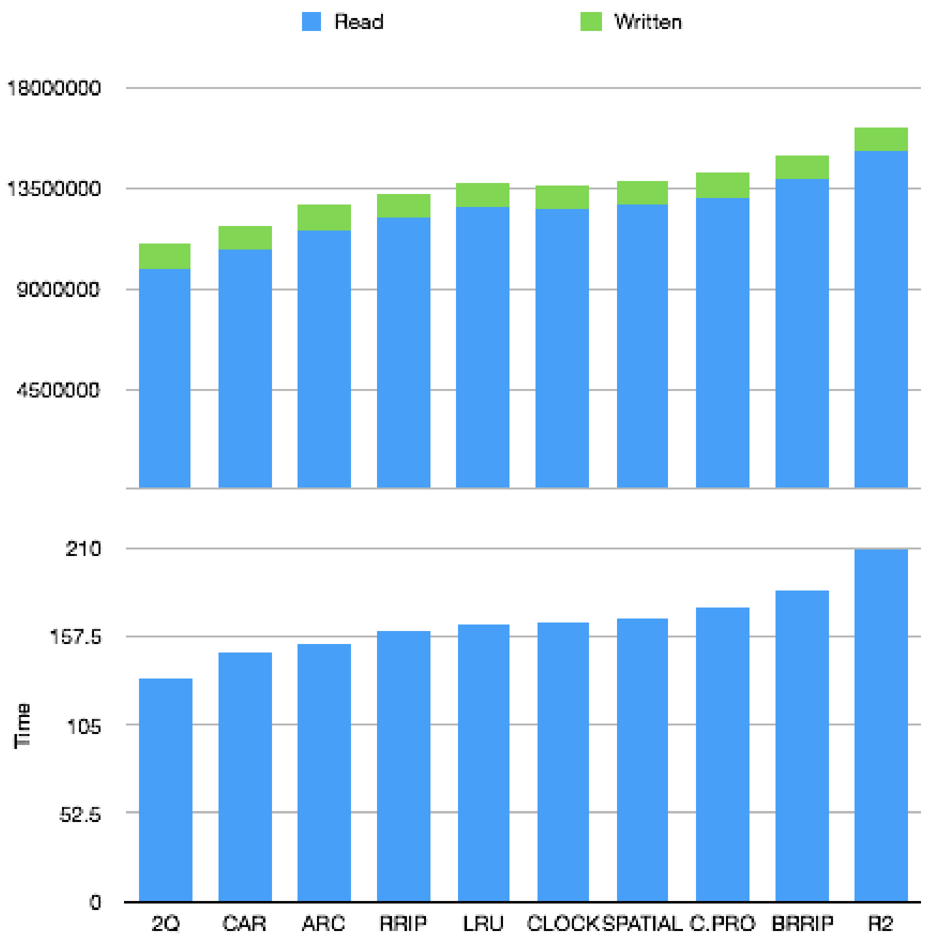

Figure 10: Test 7 - Sqlite - 90% reads - NoAppCache - 2k blocks

This test evaluates the read heavy load with a smaller cache size. We can now start to observe the relative cache hit performance of the algorithms. Top performers are the algorithms which can deal with frequency in addition to the recency. Since application cache is disabled, frequently accessed root nodes of the BTree are most likely kept in the cache for a longer time compared to the recency only algorithms.

Cache miss and time graphs are well aligned. The cache miss differences are high enough to hide the effect of the output patterns.

Test 8 - Sqlite - 90% reads - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 2k (8 MB), 84264936 sectors read, 4913090 written.

	Read	Written	Time
ARC	11701194	1089496	157.961
CLOCK	12896862	1089516	171.841
CLOCKPRO	13284963	1089612	178.785
2Q	9903823	1083760	132.345
LRU	12984687	1089273	167.280
R2	15054238	1089448	200.183
SPATIAL	13173075	1089580	176.240
RRIP	12505026	1089240	169.087
BRRIP	13634034	1089675	184.569
CAR	10801379	1090471	144.444

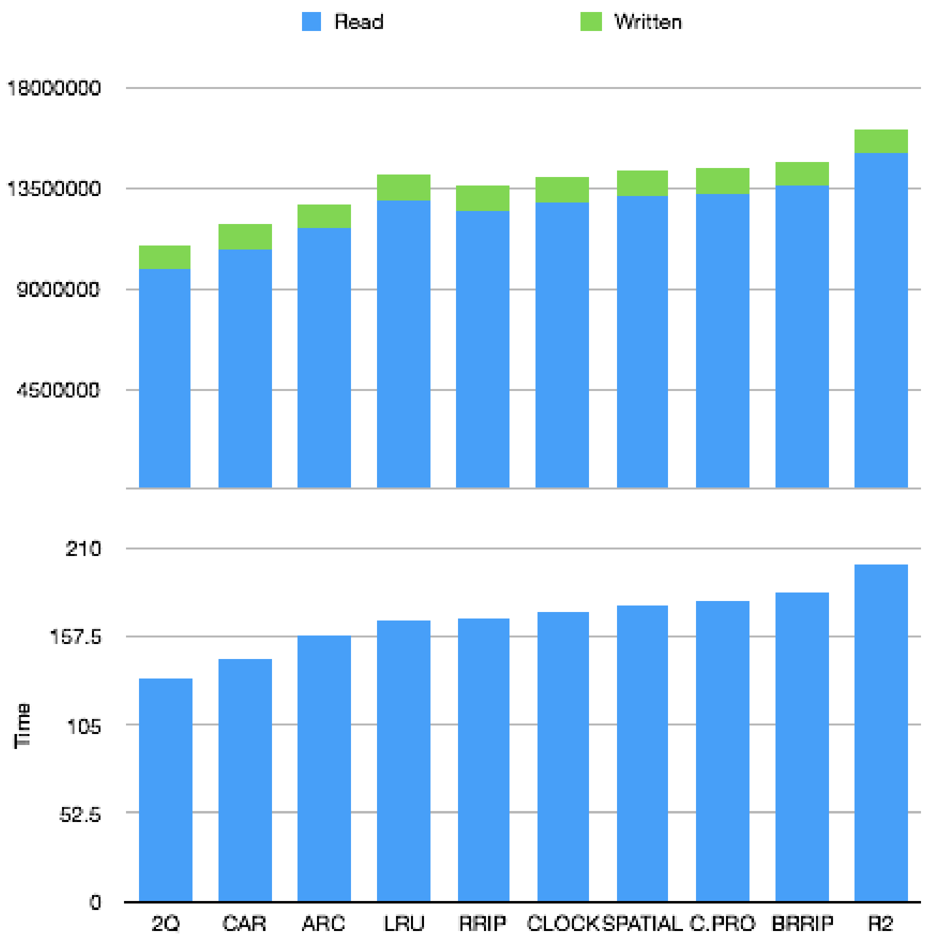

Figure 11: Test 8 - Sqlite - 90% reads - 2k blocks

With the application cache disabled, we can still observe the advantage of frequency based algorithms, and the alignment of cache miss and time graphs as the miss counts are far apart.

Test 9 - Sqlite - 66% reads - NoAppCache - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %66 of the transactions are queries and %33 are updates.

Cache size is 10k (40 MB), 180056706 sectors read, 15674525 written.

	Read	Written	Time
ARC	9493880	3421141	167.037
CLOCK	9472682	3396620	180.331
CLOCKPRO	9649310	3406751	180.673
2Q	9458606	3366934	164.008
LRU	9469031	3395257	168.773
R2	11104795	3408930	199.190
SPATIAL	9465422	3394760	177.777
RRIP	9467448	3376988	169.934
BRRIP	9475382	3363940	168.152
CAR	9468689	3395181	176.479

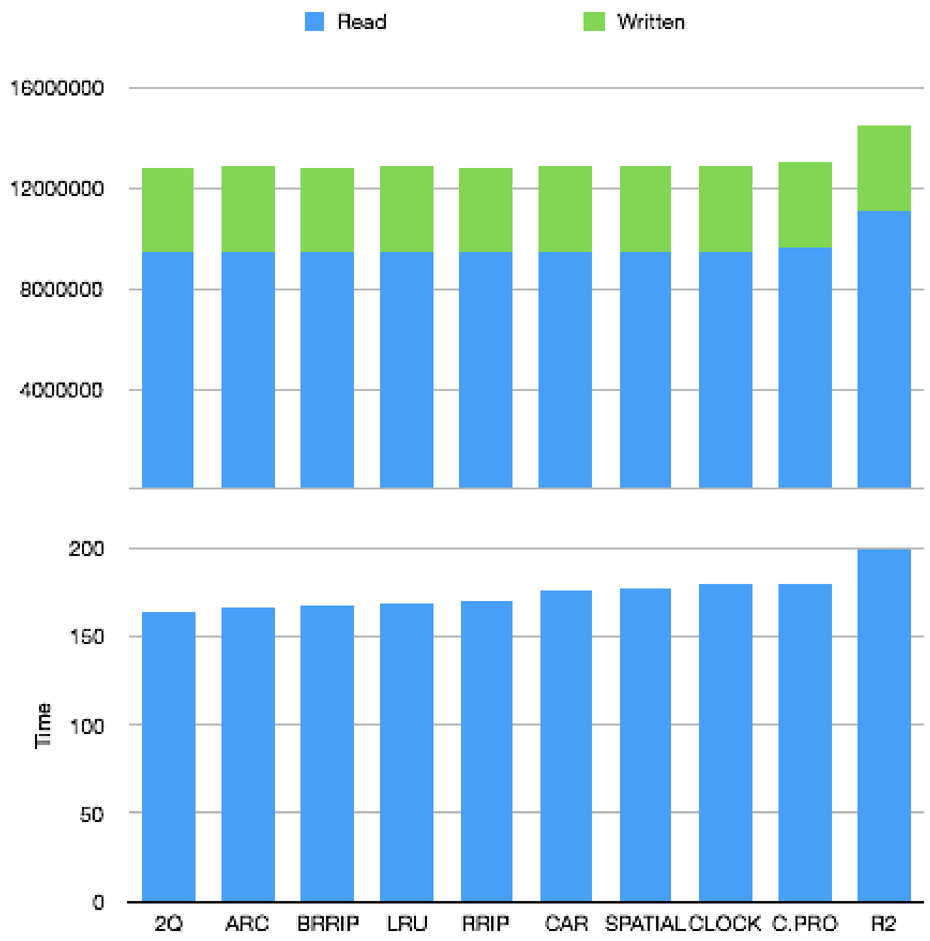

Figure 12: Test 9 - Sqlite - 66% reads - NoAppCache - 10k blocks

This time we have more updates on the database. Note that even though the amount of updates are increased, Sqlite still needs to do a lot of reads to locate and read the record which is going to be updated, so the number of reads is still high.

Again, apart from CLOCKPRO and R2, cache miss numbers are very close. We also observe the advantage of frequency algorithms. The CAR algorithm is interesting because although it has miss counts in the low end, still ended up in the slower group. Similarly ARC ended up in the second place despite having higher miss counts. We are seeing the effect of output patterns again.

BRRIP having better performance than RRIP suggests the existence of some cache trashing patterns.

Test 10 - Sqlite - 66% reads - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %66 of the transactions are queries and %33 are updates.

Cache size is 10k (40 MB), 86574084 sectors read, 15617024 written.

	Read	Written	Time
ARC	9514558	3413724	165.085
CLOCK	9479804	3396238	179.203
CLOCKPRO	9678869	3401259	178.280
2Q	9466830	3370172	165.067
LRU	9478952	3389903	168.182
R2	11238355	3421023	202.024
SPATIAL	9471743	3392066	176.305
RRIP	9475244	3372653	169.410
BRRIP	9468380	3381504	178.008
CAR	9480038	3401101	181.300

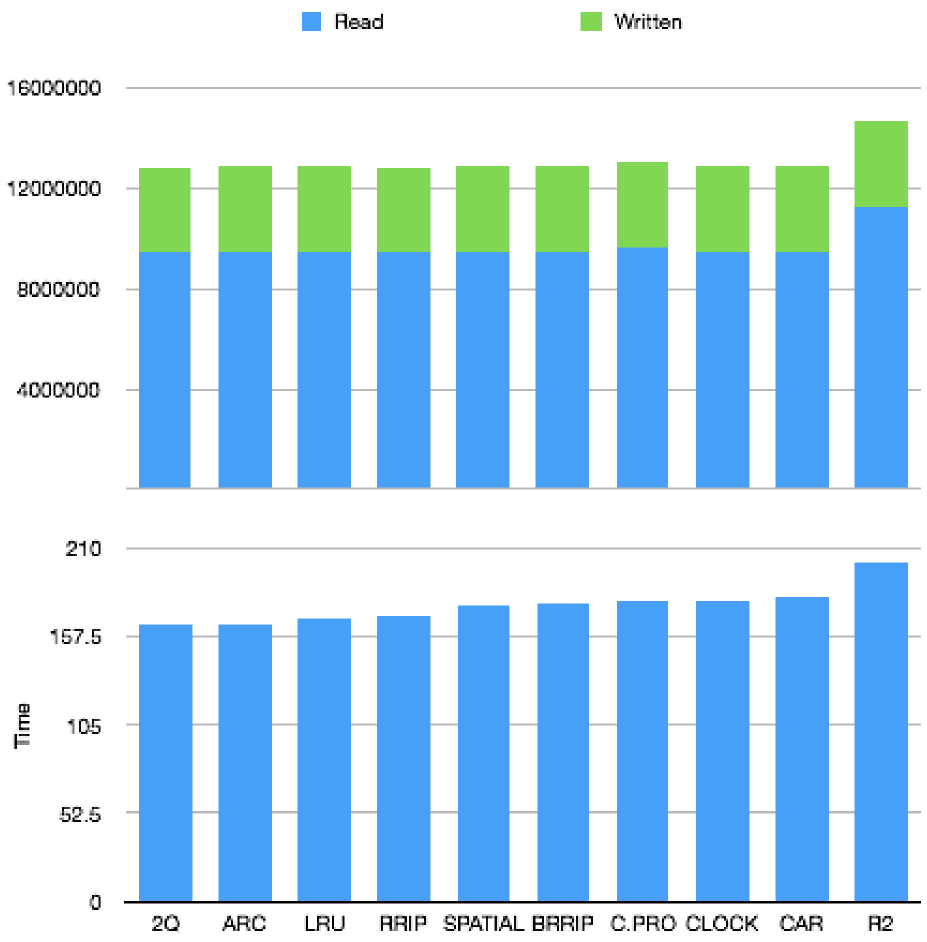

Figure 13: Test 10 - Sqlite - 66% reads - 10k blocks

The application cache version has very similar results to the test 9 except the CAR is performed even worse.

Test 11 - Sqlite - 66% reads - NoAppCache - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %66 of the transactions are queries and %33 are updates.

Cache size is 2k (8 MB), 180056706 sectors read, 15674525 written.

	Read	Written	Time
ARC	12104223	3464405	198.353
CLOCK	12567234	3468177	217.442
CLOCKPRO	13070870	3467804	225.588
2Q	9924782	3453642	171.744
LRU	12641982	3468101	202.997
R2	15138753	3466971	250.772
SPATIAL	12742142	3467963	216.766
RRIP	12469770	3465647	203.180
BRRIP	14419616	3461717	230.309
CAR	10748832	3469053	195.259

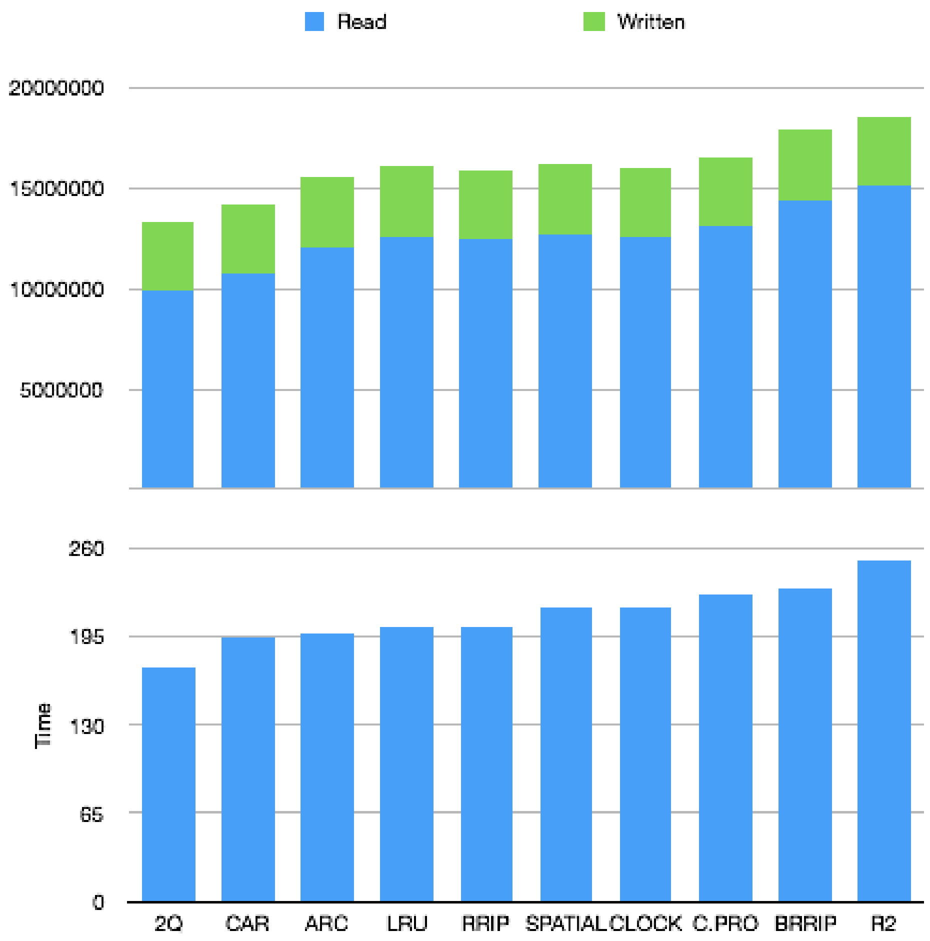

Figure 14: Test 11 - Sqlite - 66% reads - NoAppCache - 2k blocks

Reduced cache size realized the advantage of frequency based algorithms again. The graphs are aligned due to the big cache miss differences.

Test 12 - Sqlite - 66% reads - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %66 of the transactions are queries and %33 are updates.

Cache size is 2k (8 MB), 86574084 sectors read, 15617024 written.

	Read	Written	Time
ARC	12814173	3462407	205.558
CLOCK	13164549	3464528	214.614
CLOCKPRO	13542387	3461983	225.858
2Q	9905103	3440557	170.114
LRU	13486978	3462868	214.977
R2	15410358	3462806	248.135
SPATIAL	13411545	3463034	219.331
RRIP	12998507	3462209	221.310
BRRIP	14242082	3459539	238.372
CAR	11519113	3459256	205.019

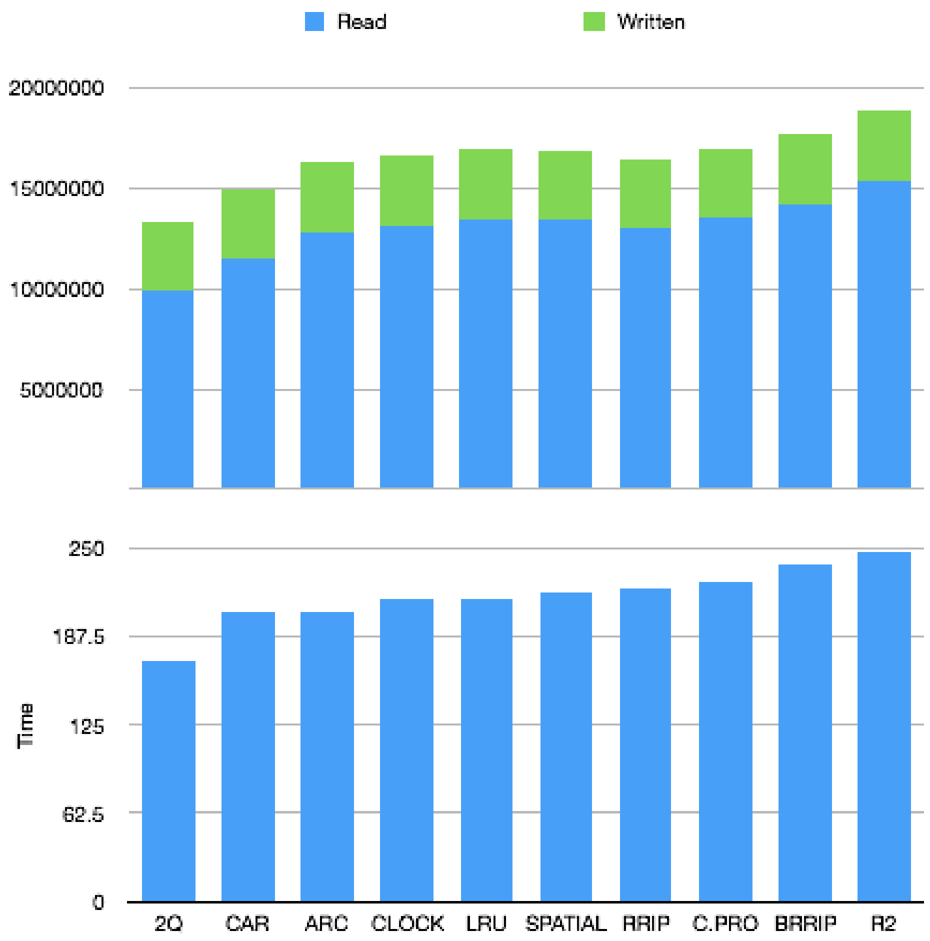

Figure 15: Test 12 - Sqlite - 66% reads - 2k blocks

Application cache is not showing any difference compared to the test 11.

Test 13 - Sqlite - 10% reads - NoAppCache - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 10k (40 MB), 180189834 sectors read, 40875389 written.

	Read	Written	Time
ARC	9468506	8679587	267.013
CLOCK	9465074	8571702	285.360
CLOCKPRO	9596896	8600294	278.265
2Q	9456607	8547651	244.293
LRU	9465722	8572023	252.460
R2	10917201	8611326	298.261
SPATIAL	9459656	8567633	279.661
RRIP	9466617	8559252	264.476
BRRIP	9482313	8555522	272.280
CAR	9464081	8570733	277.364

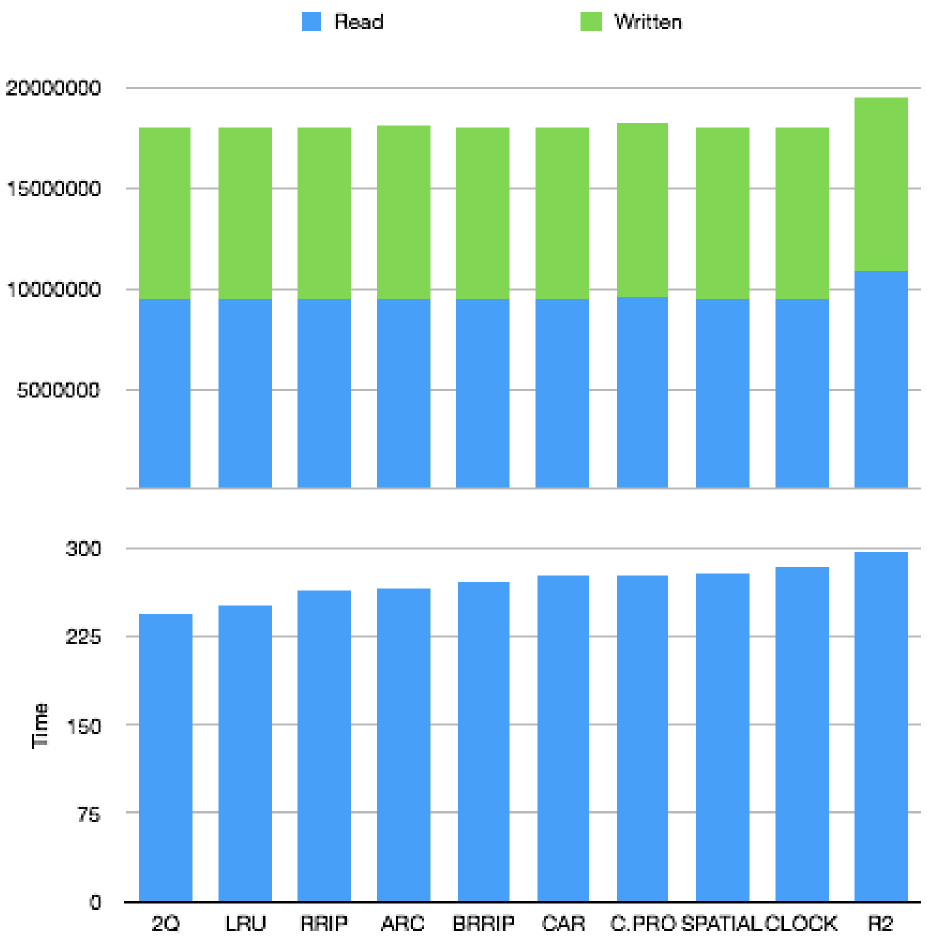

Figure 16: Test 13 - Sqlite - 10% reads - NoAppCache - 10k blocks

Here the majority of the database operations are updates. Graphs are mostly aligned. We can see the CLOCK group with the exception of 2Q performing slightly worse than the others.

Test 14 - Sqlite - 10% reads - 10k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 10k (40 MB), 87323955 sectors read, 40556357 written.

	Read	Written	Time
ARC	9514098	8691159	252.642
CLOCK	9483961	8576192	298.013
CLOCKPRO	9676427	8596752	277.321
2Q	9469002	8588565	246.178
LRU	9482378	8573839	276.202
R2	11265184	8667172	304.161
SPATIAL	9477041	8569309	278.124
RRIP	9483345	8562202	280.658
BRRIP	9474203	8550724	272.982
CAR	9482640	8578284	299.438

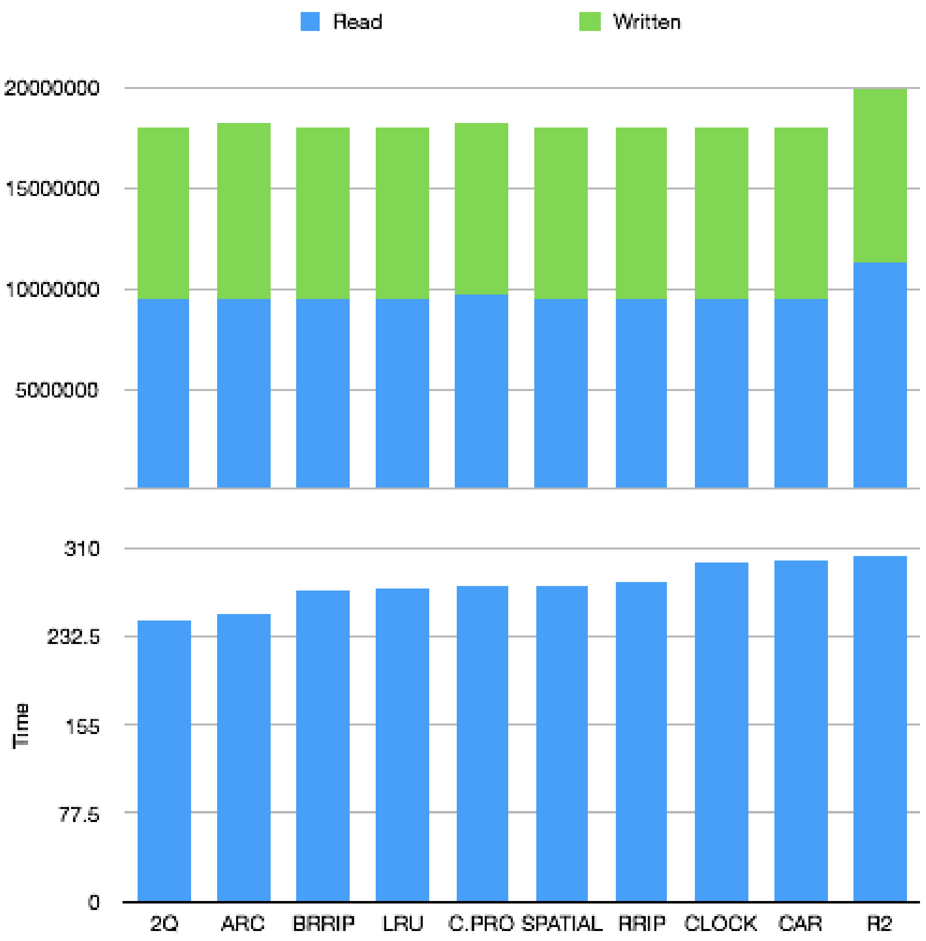

Figure 17: Test 14 - Sqlite - 10% reads - 10k blocks

Application cache did not change the results much from the test 13.

Test 15 - Sqlite - 10% reads - NoAppCache - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is disabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 2k (8 MB), 180189834 sectors read, 40875389 written.

	Read	Written	Time
ARC	12368269	8994393	297.951
CLOCK	12004692	8998002	313.078
CLOCKPRO	12595160	8995274	327.659
2Q	9932803	8962015	265.430
LRU	12083070	8996795	286.205
R2	14805402	8988068	356.623
SPATIAL	12059078	8997296	311.778
RRIP	12489098	8993964	294.418
BRRIP	14876617	8983083	328.119
CAR	10352762	9000539	299.742

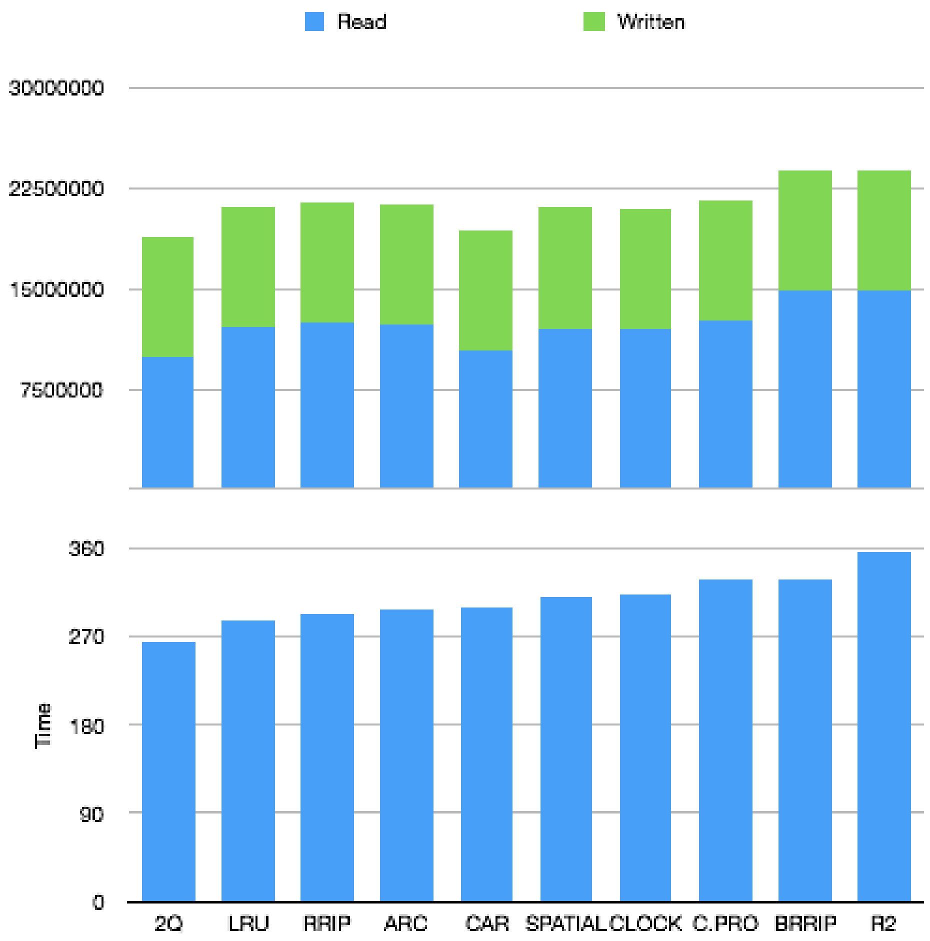

Figure 18: Test 15 - Sqlite - 10% reads - NoAppCache - 2k blocks

The graphs became very different with the smaller cache size. We can see the CAR reaching a lower miss count but still slightly slower for the actual duration.

Test 16 - Sqlite - 10% reads - 2k blocks

Five million transactions are executed on an Sqlite database of ten million entries, roughly 600MB in size. Sqlite application cache is enabled. About %90 of the transactions are queries and %10 are updates.

Cache size is 2k (8 MB), 180189834 sectors read, 40875389 written.

	Read	Written	Time
ARC	14245071	8940217	307.986
CLOCK	13189318	8958468	334.696
CLOCKPRO	13907809	8963171	339.034
2Q	14594322	8928127	316.901
LRU	14473906	8984360	317.887
R2	15752067	8978034	366.425
SPATIAL	13744973	8962013	329.944
RRIP	13864215	8949789	349.234
BRRIP	14475038	8935423	349.741
CAR	13517720	8978588	334.544

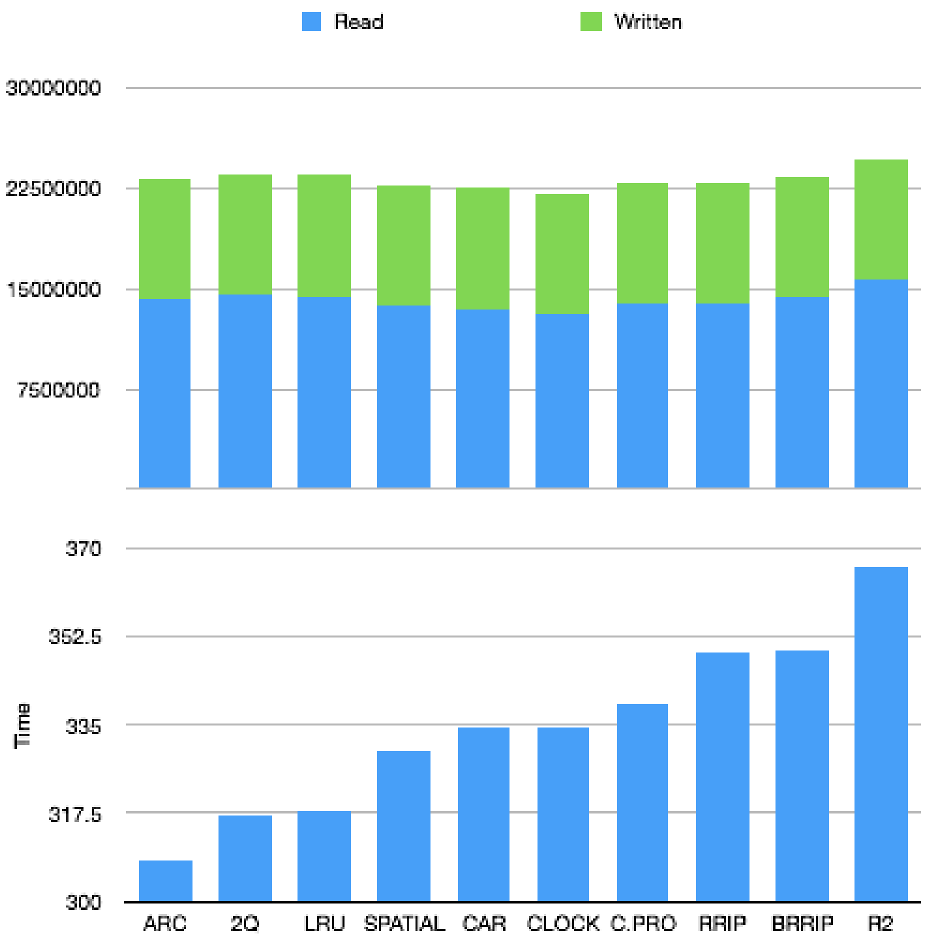

Figure 19: Test 16 - Sqlite - 10% reads - 2k blocks

This graph with the heavy writes, smaller cache size and application cache is very interesting. We can see that the first fastest entries are not the first lowest cache miss entries. If we ignore the R2, graphs are completely unaligned. Cache misses are not very close to each other, but the performance is dominated by the output order.

Chapter 5: Conclusions

The tests show that the total duration of the operation is not directly correlated with the amount of reduction of the I/O requests. Certain algorithms, such as SPATIAL, CLOCKPRO and R2 show a performance difference even under simple sequential patterns. Smaller cache sizes or certain BTree access patterns cause a difference in other algorithms.

This has important consequences for the algorithm development. The minimum amount of cache misses is not necessarily the optimum cache performance. Tests on the real hardware is as important as the simulated results, and must accompany every performance evaluation. Algorithms must accomodate the unusual performance characteristics of the SSDs.

But what are those characteristics? If we can understand and model them, we could improve simulations and design better algorithms.

The SPATIAL algorithm was designed to avoid evicting single pages in larger SSD blocks as the write operation erases the entire block. This effect can be verified with the workload player on the device by comparing the duration of clustered and separated writes. However, this optimization did not yield a comparative advantage to other algorithms in tests.

SSDs may have multiple internal channels. If a sequence of reads or writes are mapped into different channels, they could be executed in parallel. This effect can be observed by comparing a long sequential read with same blocks read in shuffled order. However, the optimum pattern is not as simple as the most sequential pattern. What matters is how the initial mapping is done from logical order to the physical order inside the device's translation layer, and how close the second access is to the first access. It is difficult to model this behaviour without knowing the exact internal layout, mapping algorithm, wear leveling algorithm and the internal caching of the SSD controller. This information is very specific to each device and usually not publicly available.

Simple metrics for patterns, such as the number of cache misses, reads, writes, the locality or sequentality of accesses do not correlate directly with the resulting operation time. Further study is necessary to create a generic model.

The 2Q algorithm is the winner in nine tests, and very close to the winner in all others. A good real life performance is not suprising given that the Linux eviction algorithm is based on 2Q. It deals with BTree patterns particularly well due to its care for frequency as well as recency, but the real advantage is how SSD friendly its output patterns are. Without the SSD manufacturer information or the further studies, we can only speculate that might be because the SSD controllers are designed to perform well with 2Q output due to its use in Linux and other OSes.

Further Studies

An accurate and generic model for the SSD performance of access patterns would be very useful to design and test eviction algorithms.

How can we take an unknown SSD and find out its internal structure such as number of channels, block size and mapping algorithms by measuring the I/O operation times?

I have observed small changes with and without the Sqlite application cache. How much difference would it make to use larger application caches, or to use other database products' application caches? Some products can use rawdisk, ignoring the OS block cache layer altogether. Does that still make sense with latest SSDs? Do they use any optimizations specific to SSDs and their workloads which could give better performance than the OS cache?

We think about finding the optimum eviction algorithm for storing BTrees on SSDs. How about the other way? Could there be a more efficient data representation which better fits with the SSD architecture?

References

Bansal, S. and Modha, D. S. (2004). Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA. USENIX Association.

Jaleel, A., Theobald, K. B., Steely, Jr., S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71.

Jiang, S., Chen, F., and Zhang, X. (2005). Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 35–35, Berkeley, CA, USA. USENIX Association.

Jo, H., Kang, J.-U., Park, S.-Y., Kim, J.-S., and Lee, J. (2006). Fab: Flash-aware buffer management policy for portable media players. *IEEE Trans. on Consum. Electron.*, 52(2):485–493.

Johnson, T. and Shasha, D. (1994). 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Jung, H., Shim, H., Park, S., Kang, S., and Cha, J. (2008). Lru-wsr: Integration of lru and writes sequence reordering for flash memory. *IEEE Trans. on Consum. Electron.*, 54(3):1215–1223.

Kim, H., Ryu, M., and Ramachandran, U. (2012). What is a good buffer cache replacement scheme for mobile flash storage? *SIGMETRICS Perform. Eval. Rev.*, 40(1):235–246.

Luu, D. (2014). *Cache Eviction: when are randomized algorithms better than LRU?* `https://danluu.com/2choices-eviction/`.

Megiddo, N. and Modha, D. S. (2004). Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65.

Mitzenmacher, M., Richa, A. W., and Sitaraman, R. (2000). The power of two random choices: A survey of techniques and results. In *in Handbook of Randomized Computing*, pages 255–312. Kluwer.

Park, S.-y., Jung, D., Kang, J.-u., Kim, J.-s., and Lee, J. (2006). Cflru: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 234–241, New York, NY, USA. ACM.