Improving the SSD-based Cache by Different Optimization Algorithms

Chi Zhang, Anthony Harrell, Eric Wong

COEN 283 spring 2014
Table of Contents

1. Abstract........................................................................................................... 3
2. Introduction....................................................................................................... 3
   2.1 Objective....................................................................................................... 3
   2.2 What is the problem..................................................................................... 3
   2.3 Why is the project related to this class......................................................... 3
   2.4 Why the other approach is not good............................................................ 4
   2.5 Why our approach is better......................................................................... 4
   2.6 Statement of the problem............................................................................. 4
   2.7 Scope of the problem.................................................................................. 5
3. Theoretical Bases and Literature Review ................................................. 5
   3.1 definition of the problem............................................................................ 5
   3.2 Theoretical background of the problem..................................................... 5
   3.3 Related research to solve the problem......................................................... 5
   3.4 Advantage/disadvantage of prior research.................................................. 6
   3.5 Solution to solve this problem.................................................................... 6
   3.6 Where our solution differs from others...................................................... 6
   3.7 Why our solution is better......................................................................... 7
4. Hypothesis......................................................................................................... 7
   4.1 Multiple hypothesis..................................................................................... 7
5. Methodology..................................................................................................... 8
   5.1 How to generate/collect input data.............................................................. 8
   5.2 How to solve the problem.......................................................................... 8
   5.3 How to generate output............................................................................. 9
   5.4 How to test against hypothesis................................................................... 9
6. Implementation.................................................................................................. 10
   6.1 Code ........................................................................................................... 10
   6.2 Flowcharts .................................................................................................. 14
7. Data Analysis and Discussion................................................................. 16
1. **Abstraction**

Since the SSD-based cache has been used widely to speed up the data intensive application for hard disk (HDD). There are many algorithms proposed to enhance the existing solutions such as: FBR, LUR, MQ, etc. But, most of these algorithms introduce additional overload (maybe overhead?) as a tradeoff for the gained improvement. We analyze these algorithms to find out the limitations that each algorithm will be facing in certain (maybe different?) scenarios, so the users can have an adaptive solution based on their specific environment needs. In our research, we look into three algorithms, the LB-CLOCK (Large Block Clock), LARC (Lazy Adaptive Replacement), and AML. By analyzing the simulation results, we find that some are unnecessary and impractical in certain real work (maybe world?) environments. Our experiment will show that conventional algorithms could achieve the same goal without adding more workload. *(This can and will change should have been written at end of project)*

Index Terms-Flash; Solid State Drive; Cache Algorithm

2. **Introduction**

2.1 **Objective**

Our main objective is to see which of the three algorithms researched holds up best to different test case simulations that try to mimic real world situations. In doing so we will get a clearer representation of which algorithm, if any of the specific ones we researched, can outperform current replacement algorithms. To perform this test we will be trying to optimize the three researched algorithms the LB-Clock, LARC and AML algorithms to see if they can perform better than their initial implementations.

2.2 **What is the problem**

The limitation of SSD that may affect its direct application in an enterprise environment is the endurance problem. The count of erasures per block for the SLC flash memory is typically limited to 100,000 while that for the MLC flash memory is limited to only 10,000 [3]. Due to this inherent problem the SSD hasn’t been widely accepted because of the unique SSD feature of *erase-before-write* imposes a real challenge to the performance and longevity of flash memory SSD [2]. To help try and solve this problem and make SSD’s a viable option for a non-volatile cache or to improve the RAM caching done in SSD hardware at the FTL (Flash Translation Layer) level. Algorithms have been designed attempting to solve this issue, but many of these algorithms depend on specific data sets to have any real gain inefficiency.

2.3 **Why this topic applies to our class**

This topic applies to what we discussed in class because of how important caching is to the overall speed of computing. If there is a way to better leverage the newer technology of SSD’s by figuring out a good caching algorithm which will mitigate the “wear out” issue plaguing this technology;
it could feasibly be implemented as a Last Level Cache that is non-volatile resulting in increased speed, but with the reliability of standard HDD for large data storage. Resulting in cheaper faster computers.

2.4 Why the other approach doesn’t work

During our research we found a common theme between all the algorithms we were researching, every algorithm worked, but only during specific workloads. In realizing this we determined that each algorithm worked in some way, but of the three which one works best in all workloads? By analyzing the current implementation we discovered how we might be able to optimize the initial algorithms to work better on a variety of real world test cases. Below we talk about the weakness of each algorithm researched, and why they don’t completely solve the problem.

The first algorithm we researched is LB-CLOCK (Large Block Clock), the main issue with the LB-Clock algorithm is that it only accounts for writes to the cache, making it a purely theoretical algorithm, because a cache replacement algorithm should handle read and writes. Further investigation into a read from the cache is needed. So we will try implementing into the LB-Clock algorithm a read from cache using similar parameters to the ones presented in the paper to see if the algorithm can truly be considered for implementation as a SSD caching algorithm. Secondly, the LB-Clock algorithm is dependent on a non-volatile cache, which is fine when a server has it implemented, but what happens with the client side? If we want to apply our write caching algorithm in the client side, which usually uses volatile DRAM based cache, we have to address the non-volatility issue [1]. They discuss further research into this being needed, but it is hardware based issue and out of the scope of this paper and the purposed idea of adding a read from cache to the current algorithm implementation.

The second algorithm is LARC (Lazy Adaptive Replacement). The LARC algorithm gets a higher hit ratio by introducing an additional queue to store the candidate blocks. There will be extra overhead when lookup happens in that queue, and even the length of that queue is adjustable. The problem is, the lookup overhead could increase the time of data access. Another problem is the response is hysteretic when the hit rate changes dramatically. The replacement ratio still stays at a lower value, but the length of the \( Q_r \) will be increased dramatically until it finally responds. This case also causes the cost of time for look up to increase as well.

The third algorithm is AML (Adaptively Mixed List). Like LARC, the weaknesses of AML lie in the fact that it requires additional queues. In AML, this is an even more severe issue due to the fact that there are two additional queues, the mixed list and the ghost list, instead of just one. In addition, AML also adjusts the size of each queue periodically. All this extra overhead slows down page replacement when all three lists have to be accessed in a worst case scenario.

2.5 Why our approach will work better

Our approach will work better by trying to optimize the algorithms which through testing have already shown that they can improve the endurance, or “wear out”, of an SSD. So by optimizing their implementations and testing with a number of test simulations which reflect situations that happen in everyday computing, we can see which algorithm of the three researched is best for caching using SSD’s and show that they are a viable option as a last level cache, or a current storage option.

2.6 Scope of the problem
For this paper we will be scoping our project to the specific optimizations presented in the next section of the paper. For the LB-Clock the non-volatility issue is beyond the scope of this paper due to it being hardware based, and not relevant to comparison of the simulation against the other algorithms. The trace data which is used in the simulations of the researched papers, only covers specific situations, we are going to come up with some scenarios and simulation data which could prove or disprove the issues mentioned above.

3. **Theoretical Bases and Literature Review**

3.1 **Definition of the problem**

Many caching algorithms for SSD’s have been created, but all seem to have one flaw, they all only work best for one specific workload. Some work best in random access, or sequential writes but implementation causes a lot of overhead due to special data structures needing to be present. Some work best in strictly writing to the cache, and haven’t even implement reading from the cache, basically only making half of a caching algorithm. So, the need of an SSD specific caching algorithm that can perform well on all types of workloads is needed.

3.2 **Theoretical background of the problem**

The theoretical background of this problem has its roots in page replacement algorithms that help speed up caches. The implementation of these caches has made modern computing possible by increasing CPU utilization through multiprogramming. The basis for all the algorithms presented in this paper are based on some page replacement counterpart. Although SSD’s have particular hurdles that need addressing such as erase-before-write, or the difference in read versus write speed. The SSD caching algorithms try to use the tried and true page replacement algorithms as a starting point and conform them to particular needs of solid state drives.

3.3 **Related research to solve the problem**

Extensive prior research has been done on this topic, if not directly on SSD caching algorithms used in combination with normal HDD’s, then on implementing a better caching algorithm in the FTL layer which resides inside the actual hardware of the SSD. All research is aimed at bettering endurance, or slowing “wear out” on SSD’s by implementing algorithms that write less, evict less, or try to have less erase block hits. Our main research focused on the three papers mentioned below.

“Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks” [1]
Improving the SSD-based Cache by Different Optimization Algorithms

“Improving Flash-based Disk Cache with Lazy Adaptive Replacement” [4]

3.4 Advantage/Disadvantage of prior research

The advantage of prior research is that we have a basis to explore different algorithms to optimize. Prior research has proved that the algorithms we are trying to optimize work well for specific workloads, and gives us a base line to test our optimizations against. In doing so we hope to help further the research put forth by the authors of the researched papers, and all researchers looking into viable SSD caching algorithms.

3.5 Our solution to the problem

For our simulation we intend to optimize the three researched algorithms, the LB-Clock, LARC, and AML algorithms. Trying to find an implementation that works best across varying workloads that are made to mimic real world computing situations. Most of the purposed algorithms work well on a specific workload type. For example LARC results suggest that LARC is better to be used for read dominant applications or for read only disk cache [4], and the LB-Clock algorithm has the highest benefit in [the] case of the online transaction processing (OLTP) type Financial workloads. This is a very significant result as the write performance of the OLTP type applications on SSDs is one of the major concerns for the datacenters in widely deploying SSDs [1]. AML proves, that runtime is determined by hit ratio and write count. It is the standard to judge whether this algorithm is effective enough. The runtime of AML is reduced at most by 18% compared to APRA. Even if CFLRU has a good performance in the read-most applications, AML reduces the runtime on average by 14% compared to CFLRU [5]. Our purposed way of optimizing each algorithm is stated below. We will start with LB-Clock, then talk about LARC’s optimization and finally talk about how AML is purposed to be optimized.

3.6 Where our solution differs from others

Each researched algorithm has its own weakness, and in doing the research we have tried to pinpoint some areas that could feasibly lead to better optimization, and thus less evictions, erases or writes; which will lead to the increased longevity of the SSD being used. Because SSD’s are only starting to gain widespread use, research of these algorithms is in its infancy. So, any optimization that can be made to currently used algorithms will help in further research that others have started. Below is how our research of the LB-Clock, LARC and AML algorithms will be changed from their current implementations.

For LB-Clock, we will look into one specific issue that makes it weaker than the other two presented in this paper. The main issue with the LB-Clock is that it was created to be a write only caching algorithm. It was purposed that a read cache can speedup performance by servicing read operations from the read cache. Moreover, it helps to reduce the load in the flash data channel (i.e., bus) which is shared by both read and write operations [1]. Due to the need for further research, as stated by the authors of “Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks”. Firstly,
a separate second read cache will be implemented to see if their hypothesis of a second read only cache implementation will help the LB-Clocks overall performance. Secondly, a standard implementing of a read from cache scheme will be added in the current algorithm. This will be implemented to see if it helps or hinders the performance of the current LB-Clock implementation. We will then compare which caching algorithm will impede or advance the results found by the papers research.

The second algorithm is the LARC, there are two ways we are going to try to implement optimizations for this algorithm. First we will use a binary search algorithm to do the $Qr$ lookup, trying to reduce the time complexity from $O(n)$ to $O(\log n)$, so it could speed up the lookup in the ghost queue. On the other hand, since there is still additional overhead caused by the $Qr$, we are going to use the second chance replacement algorithm to prove it can archive the same result without implementing the extra queue that is talked about in their research.

For AML, there are a few optimizations that could be implemented. First, when looking through the mixed list, the algorithm will store the first cold dirty page it finds, this way there is no need to iterate over the list twice if a clean page is not found in the list. Another optimization that could be implemented is reducing the frequency of when the queue sizes are adjusted. Both these optimizations aim to reduce overhead without negatively impacting performance.

### 3.7 Why our solution is better

Our purposed solution will be better because we are testing three different algorithms against each other to find which optimization gives the best results. Where each paper we researched attempted a new algorithms based on old tried and true algorithms, our approach, like theirs, builds off their research to see if we can implement one, hopefully three, optimized SSD cache replacement algorithms.

### 4. Hypothesis

#### 4.1 Multiple Hypothesis

In this section we will describe the hypothesis of our solutions individually. Then we will move into which of the three algorithms will perform best of the three. Each optimization implemented could have a positive or negative affect depending on if our optimization is implemented correctly.

For the hypothesis of the LB-CLOCK, we hypothesize that the introduction of either read caching approach will more than likely show that the LB-Clock, although good in speeding up writing to the cache and helping the “wear out” problem inherent to SSD’s. LB-Clock will not be a viable algorithm to use because of its non-reading capabilities. We believe that once reads from cache are added that the reads will diminish any improvement that the LB-Clock had produced by being a write only caching algorithm.

For the hypothesis of the LARC algorithm, we will set the cache size to an extra-large value, and make the length of the ghost queue have a similar size, the data lookup will have a loop for each element in the ghost queue, and we give a low hit ratio for the ghost cache, so it will cause an increase in
the access time. There will be a breakpoint where the total cost of time equals the gain by the additional queue.

For the hypothesis of the AML algorithm, we theorize that storing the cold dirty page will ultimately lead to slightly faster performance. However, changing the frequency of the queue size adjustment will not have a positive effect on performance.

Finally, from our current research into these algorithms and our purposed attempts to implement optimizations into them. We conclude that because LB-Clock, by its inception, was a write only caching algorithm, that once the read is added it will diminish its efficiency. Taking it out of contention for a better caching algorithm. We hypothesize the LARC, or AML algorithms will prove to be the strongest depending on which one can incur less overhead by the purposed optimizations.

5. Methodology

5.1 How to generate or collect input

We will create a data file based on real world scenarios as our input. We will try to find a balance of real world scenarios that can test each algorithm to see which workload each one is best suited for. To do this we will look at what each paper used as its testing sets and put together data sets that represent a culmination of all tests used from the research. An example of a trace set taken from the University of Massachusetts Amhesrst Storage Traces [6], which is used in both the LB-Clock test data as well as the LARC test data, is below to give an idea of what our trace data will include. Not all fields are necessary for our simulation, so the following will need to be parsed to extract what we need as well as be able to handle what is not needed.

The following is an example of the first few record of a trace file:

```
0,20941264,8192,W,0.551706,Alpha/NT
0,20939840,8192,W,0.554041
0,20939808,8192,W,0.556202
1,3436288,15872,W,1.250720,0x123,5.99,test
1,3435888,512,W,1.609859
1,3435889,512,W,1.634761
0,7695360,4096,R,2.346628
1,10274472,4096,R,2.436645
2,30862016,4096,W,2.444803
2,30845544,4096,W,2.449733
1,10356592,4096,W,2.449733
```

Trace Data from UMass Trace Repository

5.2 How to solve the problem
The solutions are implemented in C language. Each solution will use the same simulation interface which is provided by the framework to output the test result into a CSV file. Each algorithm simulation will be based on specific timing parameters determined by SSD manufacturers to help more closely simulate SSD data transfer rates, of read, write, and erase operations.

A second option that we are debating on is using a program called FlashSim to help us simulate a flash-based drive. We will implement our algorithms into their codebase, or find a way to use their tool to use our coded algorithms. This is all dependent on time and how easy it will be to integrate into their codebase. If it is too difficult we will default to the standard C simulation. Research into the code integration has started, but hasn’t been evaluated fully to say what our final direction will be.

5.3 How to generate output

The program will output to a CSV file, which will be imported into Excel to get our visual results representation and compare that with the hypothesis result, as well as our papers’ graphs, within the same scale. If FlashSim is used we will have tables of output to support our graphs that will be created from Excel.

5.4 Design diagram / Flow chart (Conceptual)

Basic Flow Chart of Simulation Using C Framework Implementation
6. Simulation Implementation

6.1 Code (snippets)

We implemented each of our algorithms using Visual Studio as our IDE. All algorithms were written in C/C++. The first algorithm is the LARC algorithm. The second algorithm is the LB_Clock algorithm, and the third and final algorithm is the AML algorithm implementation.

The LARC algorithm implemented two algorithms, the LRU and the LARC to compare read-dominant test cases. A hash was implemented to search the ghost queue. By implementing a hash for the search of this queue, we hoped to improve search times and make the overall algorithm more efficient. Here is pseudo code of the hash implementation:

If B is in the Q then

Move B to the MRU end of the Q

#hash table created for the B

Addhash(B)

Findhash(B)

If B is in the Qr then

Remove the B from the Qr and insert B to the MRU of the Q

Else

Insert B to the MRU of the Qr

The hash took care of the reference calculation, and to take care of any collusions that might occur while hashing a chaining linked list was implemented. To keep this paper concise the code for the hash and the LARC algorithms, as well as the rest of the code not shown in the explanations of the other algorithms below. A link to the GitHub repository is in the appendix A, along with a link to all trace files used for this simulation.

The LB_Clock was implemented as close to how it was described in the paper, unlike the other algorithms the paper didn’t contain pseudo code to follow for the algorithm implementation. Also, the original implementation of the algorithm didn’t have read from cache capabilities. So the LB_Clock was implemented using an “LB_Clock” structure which held size of cache, size of blocks, and page sizes as well as accounting information for output, such as cache hits and cache misses. Here is what the structure declaration looked like.

```
typedef struct LB_Clock
{
    LB_Node* clockStart;
```
As you can see the “LB_Node” was the circular linked list used as the clock, and there was a second node pointer which was used as the clock hand. “InfoString” was a structure used to keep an integer value of the cache, block and page sizes and a string that help the type it was, i.e. kilobyte, megabyte or gigabyte. The timer was used to keep track of clock ticks and whenever a certain amount of ticks happened then it reset all reference bits in the clock list to 0. This is the basic concept of the CLOCK replacement algorithm, where LB_Clock differed was in is victim candidate set.

I didn’t know how the authors officially implemented there victim candidate set, but what I envision and implemented, below is the code for eviction of a block that is a write command:

```c
LB_Node* checkVictimSetWrite(LB_Clock*lb_clock)
{
    // variables
    int evictBlock;
    LB_Node* evictNode,*currNode,*nextNode;
    int reset;

    // so there is always something to evict
    evictNode=lb_clock->clockHand;
    currNode=lb_clock->clockHand;
    nextNode=currNode->next;

    while(currNode->next!=lb_clock->clockHand)
    {
        // think about logic here.....test
        if(nextNode==NULL&&currNode->currNumOfPages>=nextNode->currNumOfPages)
        {
            if(currNode->refBit==0&&currNode->currNumOfPages==lb_clock->pagePerBlock)
            {
                evictNode=currNode;
                break;
            }
        }
        currNode=currNode->next;
    }
    lb_clock->clockHand=lb_clock->clockHand->next;
    return evictNode;
}
```

This code checks reference bits, and size of the page table, if the reference bit is zero and the page table is close to full, then it gets evicted. The clock hand is then iterated to the next node, thus basically making this similar to LRU, but with a timer.
Improving the SSD-based Cache by Different Optimization Algorithms

There was a need to make a second function that was for read’s, because of how the original algorithm was supposed to be implemented, which was explained above, but they didn’t implement a read from cache, and thus didn’t have an eviction specific to the read. In our implementation we opted to evict the block with the least amount to half full page table. This was done because we thought that more reads could take place in blocks that were full unlike the write which evicts blocks that are full. So the read looked at reference bits of zero and where the clock hand was then compared the page table, evicting the one with the least pages in it.

For the AML portion of the code, we first set the global variables. There are 3 vectors to represent the lists L1, L2, and L3. They are created as vectors because their sizes change throughout the similar and vectors accommodate this functionality the best. Also, the first element is the LRU page while the last element is the most recently used. There is a constant integer variable to represent the boundary where a page goes from hot to cold and a constant integer variable to represent to maximum cold flag value. There are also integer variables to represent to max number of pages L2 can grow up to and current size of L2.

Once that is done, the AML function is called. AML starts by creating a new page based on the inputs given to it. Then it iterates over L1 and L2 to see if there’s a cache hit. If it does, then that page is moved to the end of the vector. Otherwise AML will check if L1 is full or not. If it is then the page is appended to the end of L1. If not, AML will call the findPageToReplace function. findPageToReplace first checks to see if L2 is empty. If it is not then it calls the replace function on the first element of L2. If it is empty, the adjust_window function is called to see if the size of L2 needs to be changed. Afterwards, findPageToReplace will check to see if there’s a clean page in L1. If there is, then it calls the replace function on the first instance of a clean page in L1. If no pages satisfy these conditions, then another loop iterates over L1 looking for a cold page and calls the replace function on it. Lastly, if no pages fit any of these conditions then replace is called for the first element in L1.

```c
void findPageToReplace(page target, char* mode)
{
    if(!L2.empty())
        replace(2,0,target);
    else
    {
        adjust_window(mode);
        bool found = false;
        page tempPage;
        for(int i=0;i<L1.size();i++)
        {
            tempPage=L1.at(i);
            if(tempPage.clean)
            {
                replace(1,i,target);
                found = true;
                break;
            }
            else
            {tempPage.coldFlag++;
            }
        }
        if(!found)
        {
```
Improving the SSD-based Cache by Different Optimization Algorithms

```cpp
for(int i=0; i<L1.size(); i++)
{
    tempPage=L1.at(i);
    if(tempPage.coldFlag>=BOUNDARY)
    {
        replace(1,i,target);
        found=true;
        break;
    }
}
if(!found)
    replace(1,0,target);
}
```

The replace function does the actual evicting. This is separate from the findPageToReplace function in order to reduce duplication of code and to be more object oriented. Replace simply takes the target page and removes it from either L1 or L2, depending on what value is passed to it. It then adds the evicted page to the end of L3. After the replacement functions are called, AML will iterate through L1 and L2 in order to increase their coldFlag value. Here is what the code to the replace function looks like:

```cpp
void replace(int targetList, int targetIndex, page target)
{
    page replacePage;
    if(targetList==1)
    {
        replacePage=L1.at(targetIndex);
        L1.at(targetIndex)=target;
    }
    else
    {
        replacePage=L2.at(targetIndex);
        L2.at(targetIndex)=target;
    }
    L3.push_back(replacePage);
    if(!replacePage.clean)
    {
        timestamp+=(DATA_TRANS+MEM_ACCESS_LATE);
        timestamp+=BLOCK_ERASE;
    }
}
```

### 6.2 Flowcharts

Flowchart of LARC with Hash Algorithm
Flowchart of LB_Clock Algorithm
Improving the SSD-based Cache by Different Optimization Algorithms

Start

intiClock()

traceParser()

checkTraceMode()

WriteToCache

checkInCache()

readFromCache()

Read/Write

Create Block/Evict Block

addPages()

Hit/Miss

checkVictimSetRead

checkVictimSetRead

Read/Write

WriteToCache

checkVictimSetRead

Read/Write

Check Victim Set Read

Check Victim Set Read
7. Data Analysis

7.1 Output Generation

Our simulations ran with multiple trace files and varying block and page sizes. The different sizes was hard to lock down due to the different algorithms having different size granularity needs. For example AML and LARC only used cache size, and cache line size, which they considered blocks, whereas the LB_Clock needed, cache size, block size, and page size. So to compare data we decided to compare at the block level using 4 blocks, 8 blocks, 16 blocks and 32 blocks. With a total cache size of 4MB. The trace files are from UMass Trace Repository, all the files we used were from the storage section of the repository. We chose these trace files as test cases because two of the three research papers used one of these trace files in their algorithm research. Combining the different type of trace files would give a better understanding of which algorithms worked best under which conditions. Two of the I/O traces came from the OLTP applications running at two large financial institutions, and three of the trace files came from the Search Engine I/O traces, used to trace web searches from a popular search engine. We didn’t use the trace files in their entirety because of the sheer size of each file. We used a sample size of 500 traces to 1000 traces from each file as our
Improving the SSD-based Cache by Different Optimization Algorithms

sample size. We tried running on entire trace file and the program ran for over 20 minutes without finishing, so to save time and get a variety of results we scaled down to our sample sizes.

Of the fields specified in the documentation of the trace files, we only really needed four of them for our simulation. To make parsing the files easier and less time consuming we pulled out four field from the trace files to use below is which fields we used as input:

<table>
<thead>
<tr>
<th>Field1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical block address (LBA)</td>
<td>Size</td>
<td>Transfer mode</td>
<td>Time stamp</td>
</tr>
</tbody>
</table>

Below is the structure used to hold our input from the trace files, the names in the structure are pretty much one to one of what the input fields are in the above table:

```c
struct trace {
    char* addr;
    int size;
    char* mode;
    float time_stamp;
};
```

Our output was generated from each algorithm outputting to a specified output CSV file. We used the time stamp from our trace file as our X axis in certain depictions of our results. The second output field was the page faults and the third and final was out cache hits. The structure used to store our output data is listed below, one thing to mention is when we were initially implementing this structure we thought we would need a hit rate, or hit ratio, on every trace line, but it turned into hits in cache instead of hit rate every line, but the name remained hit rate due to time constraints:

```c
struct output_entry {
    float time_stamp;
    int page_fault;
    float hit_rate;
};
```

7.2 Output Analysis

LARC with Hash Results(4 blocks)
Graph 7.0

Page Faults of LARC Simulation (LRU, LARC, LARC-Hash)
Improving the SSD-based Cache by Different Optimization Algorithms

**Graph 7.1**

Hit Ratio of LARC simulation (LRU, LARC, LARC-Hash)

**Graph 7.2**

LB_Clock (1 large cache 4 blocks)
Improving the SSD-based Cache by Different Optimization Algorithms

Graph 7.3

LB_Clock version 2(2 smaller independent caches 4 blocks)
Graph 7.4

LB_Clock&LB_Clock V2 Hit Ratio Comparison
Improving the SSD-based Cache by Different Optimization Algorithms

All Algorithms Page Fault Comparison

All Algorithms Page Fault Comparison 2
7.3 Output to Hypothesis Comparison

The LARC algorithm has similar hit ratio when the block size is small, it does not have significant improvement, the reason is we set the length of the ghost queue to 9, it is close to the magnitude of the length of the cache size, so the overhead of the extra queue affect almost counteract the improvement. Compared with LRU, LARC reduces the page fault about 12%.

Since we implement the hash search to speed up the search in the ghost queue, it could process more references at same period, so its hit rate could be lower than the others, it is about 10% lower than the LARC and LRU when the block size is 32, and page fault is higher than the others.

For the LB_Clock there wasn’t a base line algorithm to test against. Since implementing two version of the read part of the algorithm, we can see that the author of “Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks” [1] prediction of having separate caches for read and write would be the best route of implementing a read to the caching algorithm. We see from the results that there was an increase in hit rate with version 2 of the LB_Clock algorithm. Page faults stayed about the same between both algorithms, and there wasn’t a marketable difference in hit ratio only about 5 to 10%, but the hit ratio seemed to increase over time. This could be due to the implementation of the algorithm, and needing to fill the blocks with the page granularity before it could receive constant hits. This is reflected in the graphs of 7.3 and 7.4.

There was no surprise when it came to comparing the LB_Clock against the other algorithms it doesn’t compete once the read is introduced. This could be due to many reasons, but more than likely due to the implementation of the algorithm against the other two. In the discussion section and the abnormal cases explanation sections the reasons for this are discussed.

For the AML algorithm tests were inconclusive, there was no real gain or loss for the implemented algorithm. Due to this our hypothesis was disproven.

Finally for the overall algorithm that had the best optimization our hypothesis was partly correct. LB_Clock underperformed compared to LARC and AML, which we hypothesized would happen. So
between LARC with hashing and AML with varying queue sizes, LARC won out as the better of the two algorithms. This was proved by AML’s lack of increase in hit ratio across the tests.

### 7.4 Abnormal Cases Explanation

The abnormal cases that we encountered were more on how each algorithm dealt with evictions, AML was a page based eviction algorithm, where LB_Clock was a block based algorithm. So finding and comparing data output was difficult because sizes weren’t consistent across each algorithm. For example both LARC and AML only used cache size and cache line size, whereas LB_Clock used a different break down of sizes. It used cache size, and block size (cache line size), but it had another granularity to account for which was the page size. These differences lead to cases of different hit ratios that might not truly represent how effective one algorithm is against the other. This leads to the discussion below of what we feel needs to be explicitly emphasized when talking about caching for SSD’s.

Secondly, we ran into some issues with parsing the data, something in the parser code was causing weird loops in multiple algorithms, sometime leading to erroneous results because some trace lines got looped twice, or in one instance it looked like the files were read 4 times in a row on the last algorithm.

### 7.5 Discussion

We feel that research into this area needs to be broken down into categories of research. In all the research we did it was never clear which cache was being tested, LARC was the only algorithm that was distinct in informing the reader that it was being used as a hybrid cache, whereas LB_Clock talked about being a cache for large data servers, but then talked about being implemented above the FTL, and seemed to be a caching algorithm that was for the RAM in an SSD not used as a last level cache. We could never tell which AML was meant to be used for. So, if anything was gained from this research it is that SSD caching algorithms need a clearer definition of their intention, whether it be as a last level cache or a FTL style cache written to help the RAM in the SSD. Overall, it was a good simulation to see if caching algorithms that were meant to be at the FTL or RAM level could compete with page replacement algorithms that were meant as a last level, or hybrid, cache.

### 8. Conclusion

#### 8.1 Summary

In conclusion based on our simulations all optimizations purposed had some gains as compared to their initial implementation. LARC saw and increase in of around 10% when using a larger block size. LB_Clock performed better as a multi-cache read and write caching algorithm compared to the version of one large cache for both read and writes by about 5 to 10%. It was also shown that the LB_Clock shouldn’t be used as a caching algorithm against the other two algorithms presented in this paper.

AML didn’t show conclusive results in either direction the implementation of the varying queue added overhead, but didn’t increase or diminish how the caching algorithm performed.

The only draw backs to these implementations is the overhead necessary to optimize them, so you have a trade of in performance but with the addition of extra overhead. Further research is
purposed below, and once tested in an actual kernel the performance gains could possibly outweigh the
drawbacks of added overhead.

8.2 Recommendations for Future Study

As stated in section 7.5 there needs to be a clearer distinction between what SSD caching
algorithms are intended for, and the study of these algorithms could greatly help the speed and cost of
current day computers. The next step to fully realizing these algorithms as a true last level cache on a
system is to implement them into a kernel. We would recommend implementing these or other
algorithms into software such as Flashcache, created by Facebook, or EnhanceIO driver, which can be
updated and tested in the linux kernel. We think these algorithms have potential against the basic ones
used in the stated software, but didn’t have time to implement our algorithms into them. Also,
another area of research into these could be multi-threading these algorithms to increase their
throughput. LB_Clock could see positive gains from threading, especially in the case of separate caches
for read and write. One thread could be used specifically for the read cache, and one specific to the
write cache. As for the LARC and the AML, more research needs to be done into fixing the overhead
issues of these algorithms, this holds true for the LB_Clock as well. As a final note, as solid state drives
becomes more mainstream in there use, these algorithms could help shape the way they are
implemented into current systems, or how they interact with the FTL, but as we stated earlier in the
paper the related research needs to be specific in which area is trying to be researched.

9. Bibliography

[1] Large Block CLOCK (LB-CLOCK): A Write Caching Algorithm for Solid State Disks, Biplob Debnath, Sunil
Subramanya, David Du, David J. Lilja. MASCOTS 2009

[2] PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD
Jian Hu ; Dept. of Comput. Sci. & Eng., Univ. of Nebraska, Lincoln, NE, USA ; Hong Jiang ; Lei Tian ; Lei Xu

http://www.smxrtos.com/articles/mlcscl.htm

[4] Improving flash-based disk cache with Lazy Adaptive Replacement
Sai Huang Wuhan Nat. Lab. for Optoelectron., Huazhong Univ. of Sci. & Technol., Wuhan, China Qingsong Wei ; Jianxi Chen ; Cheng
Chen ; Dan Feng

Hao Zhu ; Sch. of Comput. Sci. & Technol., Nat. Univ. of Defense Technol., Changsha, China ; Huadong Dai ; Yuejin Yan

[6] “University of Massachusetts Amhesrst Storage
10. **Appendix**

GitHub Repository of our code base to download current code and what was used for version control: [https://github.com/ssd-cache/term_project](https://github.com/ssd-cache/term_project)

UMass Trace Repository File can be found here and are in a particular zipped format. Use Excel to open the file and use comma's as delimiter. Our sample size traces are in the trace directory in our solution: [http://traces.cs.umass.edu/index.php/Storage/Storage](http://traces.cs.umass.edu/index.php/Storage/Storage)