Online Analysis of Disk I/O

By

Michael Thomas Marzullo
B.S., University of Louisville, 2019

A Project
Submitted to the Faculty of the
University of Louisville
Speed Scientific School
as Partial Fulfillment of the Requirements
for the Professional Degree of

Computer Science and Computer Engineering

December 2019

Online Analysis of Disk I/O

Submitted by

Michael Thomas Marzullo

A Project Approved on

_____

(Date)

by the Following Reading and Examination Committee:

_____

Dr. Nihat Altiparmak, Project Advisor

ABSTRACT

The move from hard disk drives to solid state drives has enabled a performance increase over an order of magnitude, and other state of the art technologies such as 3D XPoint have another order of magnitude of performance over SSDs. These massive advancements in hardware have left gaps in the software that optimizes their usage so that their full potential can be utilized. We have developed a solution to identify correlations in block device writes in the Linux kernel in a real-time manner to improve the total I/O performance on advanced devices.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Altiparmak, for his excellent guidance and instruction that have allowed the completion of this project. I would like to thank my family for always supporting me through my education and ensuring my success; my mother, Kristen, whom has always shown me the way when I have been lost, and my father, Peter, whom has inspired me to utilize my abilities to their whole potential. Finally, I thank my friends for their support, specifically Sadaf Fatima Ahmed for making me a better person.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
Related Work

For our analysis, we focus on collecting block correlations since they may be used to optimize parallel I/O performance, reduce the cost of Garbage Collection (GC), improve caching performance, and other tasks. To collect these correlations, frequent itemset mining (FIM) techniques can be used. FIM algorithms take a series of transactions as input, and output associated items with a support value greater than a specified minimum. Within the domain of FIM algorithms, there are offline and stream algorithms. An example of a basic offline FIM algorithm is the Apriori algorithm, which performs a scan of the transactions to first filter all items that are not frequent and then finds the associated items. This algorithm has significant exponential computational complexity, making it inefficient for large itemsets. Another offline FIM algorithm, ECLAT (Equivalence Class Clustering and bottom-up Lattice Traversal), provides a large performance enhancement over the Apriori algorithm based on a depth-first search of intersecting transactions. It can be executed in parallel and typically outperforms Apriori by an order of magnitude. One application of frequent itemset mining to the domain of block correlations has been explored by Li et al. in C-Miner: Mining Block Correlations in Storage Systems. C-Miner is an offline mining algorithm that can be used to find correlated blocks from sources such as file systems or databases. A "gap" measurement is defined to limit the maximum distance between frequent subsequences. This creates a sliding window in which C-Miner processes items, which limits the spatial locality. Temporal locality is not considered for the generated block association rules. Offline algorithms provide a basis for providing accurate correlation results, but they cannot be applied to perform optimizations in real-time for data such as live IO streams from a block device.

Stream based FIM algorithms have also been explored more recently to apply these data mining techniques to continuous data, which is applicable to many domains in the real world (network traffic, gps, sensor data, etc..). Shin et. al have created a tree structure that adapts to a configured memory size. They present the estDec+ algorithm which implements a Compressible Prefix (CP) tree to dynamically accept items from a stream source. The CP-tree merges and splits nodes to efficiently utilize the allocated memory space for the tree. The estDec+ algorithm's accuracy is determined by the proportion of the allocated memory of the tree versus the size/throughput of the input stream. However, the performance of streaming algorithms such as estDec+ is not applicable to extremely high throughput streams from high performance IO devices.

These stream algorithms also typically focus on generating the maximum frequent itemsets instead of frequently correlated pairs which we are attempting to find. They also must perform efficiently given a limited memory space and CPU resources to not reduce performance on the rest of the system.

Instead of focusing on data mining through advanced techniques, we focus on simpler methods used within computer architecture to achieve fast processing time and accurate results. Much research on caching methods has been performed for use with hardware and software. The Adaptive Replacement Cache (ARC) describes a cache that dynamically resizes based on access patterns. Optimal performance can be achieved without fixed user configuration of the cache size. The ARC consists of four separate LRU caches. The two main caches store recent items and frequent items (more than 1 hit). The two other caches are ghost caches which store items that have been evicted from the main caches. A fixed cache size is set for the two main caches, and the size of the two ghost caches is automatically tuned by the algorithm. In their experiments, ARC was shown to be superior to other caching algorithms such as 2Q.

CHAPTER II

Implementation

A. Efficient Monitoring of Disk I/O

The Linux blktrace utility is used to monitor all block layer IO events. A user level and kernel level component is used to enable this functionality. The user-level blktrace accepts parameters such as the type of request to filter and the block device to monitor. When the utility is run, it signals the kernel component to enable monitoring. The output produced by the utility is in a binary format, so to convert them to a human-readable format, the blkparse utility is used to parse this output into a format specified by the user. This output can either be stored in a file or output to stdout. Because of this, the utility can only be used for offline analysis of block events. Along with this, the tool monitors all system activity and can only be filtered by the block device. Each event trace includes information such as the CPU id, command type, process id, time stamp, sector number, and offset. At the kernel level, an ioctl handler is used to receive the input from the user-level blktrace utility to both enable tracepoints within the block layer and set the parameters and filters requested by the user utility. The events are then communicated from the kernel to the user-level with a debugfs relay.

To monitor storage I/O requests for specific applications, we developed a modification to the blktrace mechanism within Linux kernel to filter traces sent to the user-level blktrace program by process id. In the kernel, the blk_user_trace_setup struct was changed to include an additional integer array to store process ids to monitor. When this parameter is received by blktrace through the blk_trace_ioctl function, only IO requests from the

specified IO filter are forwarded to the userspace relay from the __blk_add_trace function. The user blktrace utility was then modified to accept a pid parameter so that the pid may be set dynamically to monitor requested processes. The binary output from blktrace is piped into blkparse, which acts as a writer to a named pipe containing the sector, offset, and timestamp of the request. This named pipe may then be read by another program to be used for online analysis.

A weakness in the current implementation is that these modifications cannot filter block level operations that are created through asynchronous I/O system calls. When these I/O operations are performed, the process id passed to the block layer from the kernel is 0 instead of the original process id, which means that they cannot be filtered through blktrace. An additional modification in the kernel would be required to either pass the original process id to the block layer or track the requests in another manner entirely.

B. Online Disk I/O Analysis Techniques

We developed an multi-level caching algorithm that takes a data stream of transactions as input and processes them in a near real time fashion to allow frequent block correlations to be queried. Multiple iterations of this algorithm were created to gradually improve performance. The first iteration of the algorithm (implemented in Python) is described below.

1. Algorithm 1

The basic architecture of the algorithm is a two-level cache with an additional eviction cache. The first level cache acts as either an LRU or LIFO cache with a configured maximum item count. This simple method is used as a first step in quickly filtering the large stream of items received. After an item count increases past a defined support

threshold, the item is then inserted into the second level of the cache. This second level represents all frequent pairs and uses another hash table to store and retrieve the data. The items may then be evicted based on either a maximum sequence distance or time threshold.

While the cache is running, a separate class generates a graph of all possible block associations to be queried by another application. This graph is created from the items in the L2 cache by applying DFS to output the connected components. The graph itself utilizes an adjacency matrix to store all block ids as individual vertices. These block associations are then assigned a stream to use through round-robin selection.

When items are received by the algorithm from the named pipe, all possible block pairs are generated to be inserted in the L1 cache, which is an O(N^2) operation. In our initial performance testing, this was found to be too slow to handle workloads such as large sequential reads in real time. Because of this lack of performance, more modifications were needed for a new algorithm.

2. Algorithm 2

In this modified algorithm, the received write operations from the tracing component are stored as intervals instead of individual pairs. The primary data structure used to store these items is an interval tree, which is a modification of red-black tree that stores intervals instead of values in the leaves of the tree. This allows high performance throughput for IO operations that contain a large range of blocks, with the added complexity of handling and adapting overlapping operations on existing intervals within the red-black tree. In addition, to further improve both the performance of the cache and the accuracy of the results to track both frequency and temporal locality, the first level cache structure from the first algorithm is replaced with the ARC algorithm previously described. Once the support from the frequent LRU cache reaches the minimum threshold, the item is moved into the L2 frequent cache, which is also an interval tree. The eviction policy for

the L2 cache is determined either by time passed or transactions processed.

Once the frequent itemsets are queried by another application, the algorithm generates the correlated block pairs from the L2 cache.

CHAPTER III

Experiments & Results

The experiments were run on an Intel Xeon E3-1230v5 with 64GB of RAM and a Samsung SM961 SSD. Fio was used as a synthetic benchmarking tool to demonstrate the performance and accuracy of our algorithm. Fio can generate workloads based on job configurations that can use a variety of different system interfaces and IO request types. For our evaluation, we focused on creating workloads that contain blocks that are overwritten over defined sequences and times. This requires precise control of the sectors and block sizes written to the IO device, which fio does not have any configuration to produce these types of workloads. However, it does have a method to replay the binary output from blktrace or its internal iolog format. We developed a tool to artificially create workloads in the form of the blktrace output, which is then ran by fio. This tool allows us to define a distribution of sequential and random writes along with specific block ranges to repeatedly perform writes. Through these workloads, the algorithm's effectiveness can be evaluated through multiple metrics. In addition to our synthetic workloads, real traces from different workload types (web, database, etc…) are also replayed through fio and evaluated.

The accuracy of our results is evaluated based on a comparison to offline algorithms, …

CHAPTER IV

Conclusions

CHAPTER V
Future Work

APPENDIX I

Drawings and Diagrams

APPENDIX II

Source Code

REFERENCES