ANALYSIS OF AN IMPROVED DISTRIBUTED CACHE

A REPORT

Anirudh S Narayan

Prof. Dr. Hans-Peter Bischof
# Contents

INTRODUCTION .......................................................................................................................... 4

PRIOR WORK ............................................................................................................................... 4


   Introduction .............................................................................................................................. 4

   Traces and simulations ......................................................................................................... 4

   Benefits ................................................................................................................................ 5

   Proposed cache and results ................................................................................................. 5

Paper: Optimizing Data Popularity Conscious Bloom Filters ...................................................... 6

   Introduction .............................................................................................................................. 6

   Traces and Simulations: ....................................................................................................... 7

   Results .................................................................................................................................. 7

Paper: Theory and Practice of Bloom Filters for Distributed Systems ......................................... 8

   Introduction .............................................................................................................................. 8

   Content ................................................................................................................................ 8

Paper: A Multi-attribute Data Structure with Parallel Bloom Filters for Network Services .......... 10

   Introduction .............................................................................................................................. 10

   Proposal .................................................................................................................................. 11

   Results .................................................................................................................................. 13

PROBLEM STATEMENT ............................................................................................................. 14

   Expectation .............................................................................................................................. 14

DESIGN ....................................................................................................................................... 15

   Data Representation ............................................................................................................. 15

   Protocol .................................................................................................................................. 16

   Traces .................................................................................................................................... 16

   Warm Up ............................................................................................................................... 16

   Measures ............................................................................................................................... 17

   Replacement Algorithm ....................................................................................................... 17

   Masters ................................................................................................................................. 17

   Clients .................................................................................................................................. 18
An Analysis of an Improved Distributed Cache

IMPLEMENTATION.................................................................................................................. 18
  Configuration File.................................................................................................................. 18
  Data Engine.......................................................................................................................... 18
  Master and Client.................................................................................................................. 19
  Post-simulation.................................................................................................................... 19
  Logging.................................................................................................................................. 20
  Utility.................................................................................................................................. 20

RESULTS .................................................................................................................................. 20
  Effect of varying cache........................................................................................................ 22
  Effect of varying clients....................................................................................................... 26
  Replacement Policy.............................................................................................................. 28

CONCLUSION....................................................................................................................... 30

FUTURE WORK..................................................................................................................... 30

REFERENCES....................................................................................................................... 31

ACKNOWLEDGEMENT........................................................................................................ 32
INTRODUCTION

Caches are employed in many systems to facilitate easy access to data. In case of web cache, caches can store URLs in them to make data access faster. They do this by holding on to data sources in the server memory before accessing the data from source. The easiest way for data to be accessed is from memory. This project seeks to identify improvements in a Distributed Cache by having multiple masters or nodes that hold an extended cache. It seeks to design, implement and measure the performance parameters in a Distributed Cache employing multiple masters and present the findings with their respective conclusions.

This report also goes over the work done till now in the realm of Distributed Cache to serve as a primer to the design and seeks to understand the details that future work may entail. It also goes over Bloom Filters and parallel Bloom Filters so that it may provide insights into the challenges that might be faced in promoting the simulation to real world design.

PRIOR WORK

By Li Fan, Pei Cao, Jussara Almeida and Andrei Z Broder

Introduction
This paper proposes to put forward the advantages of cache sharing, measure the overhead of existing protocols and propose a new protocol - summary cache. They rely on two things - efficient representation of the cache dictionary and lazy update of cache summary. Trace simulation and prototype implementation of the new protocol were used to measure the performance against existing protocols, mainly Internet Cache Protocol (ICP). Another advantage they claim is that the new protocol is scalable to multiple proxies.

ICP causes $N*(N-1)*(1-H)*R$ messages to occur among $N$ web proxies receiving an average of $R$ messages with a cache hit ratio of $H$. This number increases quadratically when the number of proxies and misses increase. To counter this, other mechanisms, like URL partitioning amongst proxies was suggested but it is inefficient over a wide area network. The advantage of summary cache is that each proxy stores a summary of cache information (dictionary) of all proxies. This helps to narrow down communication by querying the dictionary only when there is a cache miss. Cache dictionaries are stored as bloom filters to provide for efficient data storage representation.

Traces and simulations
5 sets of traces of HTTP requests were collected: DEC (Digital Equipment Corporation Web Proxy Server traces), UCB (University of California, Berkeley Dial-Up Service), UPisa (Traces of HTTP requests at
An Analysis of an Improved Distributed Cache

Computer Science Dept., University of Pisa, Italy), Questnet (logs of HTTP GET requests seen by parent proxies at Questnet, a regional network in Australia) and NLANR (1 day log of HTTP requests to the 4 major parent proxies of National Lab of Applied Network Research). To simulate cache sharing, the clients in Upisa, UCB and DEC were grouped. This is similar to proxies being shared across the groups. A client belongs to a group if the client ID mod group size equals group ID. Since the GET requests in Questnet have child proxies, simulations can be run with the assumption that all requests go through the child proxies. NLANR data can be simulated assuming that the cache is shared among all 4 proxies. The simulations all use LRU as the cache replacement algorithm. Docs greater than 250KB are not cached in the simulations. Cache consistency is maintained by treating access requests to documents whose modified time has changed as cache miss.

Benefits

In infinite cache capacity, studies show web cache hit ratio grows logarithmically as compared to the size of the population being served. This paper has focused on finite cache size by simulating runs using traces obtained above. The simulation scenarios are: No cache sharing, Simple Cache Sharing, Single-Copy cache sharing and Global cache. This is to study whether cache hit ratio is affected by the type of scheme used to control the cache and how they perform under different cache sizes. The paper claims that all types of cache are better than no cache at all. Also, hit ratio in single copy and simple cache sharing fare better than global cache. This is because of LRU algorithm being used. There are also very minor differences between single copy and simple cache. However, the authors note that global cache is good when load imbalances occur.

Proposed cache and results

The proposed cache - summary cache stores a dictionary of all the proxies’ cache. If the query is not found in a node’s cache, it queries the dictionary to find if the other nodes have it and proceeds to query them. Two kinds of errors are tolerated – false misses and false hits. False miss happens when the summary does not predict the actual presence of a document in some other node. False hits happen when the summary falsely reflects the presence of a document. The constraints for the summary cache are presented in terms of memory required and inter proxy network traffic. Network overhead is measured through the frequency of summary updates and false hits and remote hits. The authors also argue that the impact of update delays are minimal and can be tolerated by the implementation.

Bloom filters were used in the implementation to efficiently store data and preserve memory. A bloom filter uses m bits and k independent hash functions. For every query Q, we calculate the hash function \( h_1(Q)...h_k(q) \) and update the corresponding bit in the Bloom Filter to 1. In case of a query, if the bit enquiry comes as zero, we know that that particular query is not present in the bloom filter representation. The probability that a bit is 0 is \( (1 - 1/m)^k \). The advantage of using bloom filters is that it requires very little storage and offers scalability. When an update is done to the summary cache, the node can send the entire array or info about which bit has been flipped.

The authors claim that bloom filter based caching is 55% to 64% better than the ICP protocol. The authors suggest a configuration having an update threshold between 1 and 10% and using at least 4 hash functions.
Their implementation uses UDP for implementation as ICP has been implemented on UDP and thus offers better parameters of comparison. The study shows that the hit ratio is almost constant for various data representations as shown in the figure below:

![Fig 1: Summary Cache for various representations of data elements and their effect on the Hit Ratio](image)

As far as the cache size goes, the increase in cache threshold sees a significant increase in the Hit Ratio for the Summary Cache. It shows that having greater memory for data queries has a direct impact on the cache performance more than any other parameter.

Paper: Optimizing Data Popularity Conscious Bloom Filters
By Ming Zhong, Pin Lu, Kai Shen and Joel Seiferas

Introduction
This paper studies minimizing of false positive errors in Bloom Filters and ways to improve it. They suggest an algorithm to better improve accuracy of the data structure and claim to provide up to 27 times error reduction compared to standard Bloom Filter. They model the problem as a non-linear integer programming problem. Integer programming is a linear programming model where all the intermediate results are integers. From the previous paper, we know that hit ratio is 1 - ((1 - 1/m) ^ km). The popularity conscious Bloom Filters proposed by the authors have the disadvantage of incurring additional offline computation in calculating hashes specific to each object and associated storage requirements.

The authors used both integral numbered hash functions and arbitrary real numbered hash functions in their Bloom Filters to conduct their basic studies. They assigned each member in the Bloom Filter an importance score based on polynomial and (2+\(\epsilon\)) approximations.
Traces and Simulations:

The authors used a Zipf-like function for their query probability distribution and used a uniform distribution for the simulation members. The queries were sent to the master under different skewness parameters in the Zipf function to gather performance characteristics of the Caches using Bloom Filters employing arbitrary real number of hashes and integer numbered hashes. They found that traces using a high skew in the Zipf function have a lower false hit ratio, tipping to below 1 per cent for integral number hash functions. The effect of this skewness on real numbered hash function Bloom Filters was found not to change.

The authors used the traces generated by National Lab of Applied Network Research (NLANR) Web Caching project in their design of a Distributed Cache. The traces have 42.7 million web requests with 15.9 distinct URLs. The proxies send their cache requests to the simulator which adds each query to a queue. The simulator processes each request based on the time they entered. The simulator broadcasts the summary cache to all caches when there is a change of 1% in it. The simulator checks the other caches for the presence of a data block if there a cache miss. If another cache has it, the client request is redirected to it. The caches themselves were studied using two different replacement policies – Least Frequently Used (LFU) and Least Recently Used (LRU) replacement policies.

An LFU replacement policy has a reference count and replaces the least frequently used entry for a new entry into the cache. An LRU cache replacement policy replaces the least recently used block in cache with a new block. More replacement policies were not studied as they had little impact on the performance of Bloom Filters on their own. The performance of integral and real numbered hash functions were compared with the standard Bloom Filters. Also, a popularity driven optimal solution was also designed and compared for performance analysis. For larger datasets, the performance of Bloom Filters are dominated by set operations. The authors have researched Reynold and Vahdat’s algorithm for set intersection operation. However, for their implementation of popularity based Bloom Filter, they have used the following algorithm. For a query consisting of k keywords, with ascending order of inverted list sizes, the simulator visits the hosts in the same order and calculates $S_1 \cap S_2, (S_1 \cap S_2) \cap S_3 \ldots \cap S_k$ where $S_1, S_2, \ldots S_k$ are inverted lists representing each of the k words and the inversions are calculated using Reynold’s and Vahdat’s algorithm. They claim that this results in lower communication overhead for the same Bloom Filter size.

Results

The authors’ proposed 2-approximation algorithm for a polynomial time integer approach to designing Bloom Filters gives a run time complexity of $O(N^c)$ where $c \leq 6$. They claim that their popularity based Bloom Filters can achieve better performance characteristics than the standard Bloom Filters. They also claim that their solution is also applicable to real world solutions as their simulations used real data sets with stable skews in queries to measure their performance.

They also rely on the prior work on word inversions using set operations to reduce the number of requests being sent to each of the master nodes hosting the cache, thereby reducing the overall communication overhead between the different nodes in the cache.
An Analysis of an Improved Distributed Cache

Paper: Theory and Practice of Bloom Filters for Distributed Systems
By Sasu Tarkoma, Christian Esteve Rothenberg and Eemil Lagerspetz

Introduction
The main aim of this paper is to reduce the computational costs and network overhead in Distributed Cache systems with focus on Bloom Filters. The authors present various Bloom Filters and the circumstances which warrant their use. They also go over a basic representation of Bloom Filters and how to implement them and explore the various parameters that determine the performance of Bloom Filters.

Content
The basic Bloom Filter only supports insert and update operations of elements and its accuracy is determined by size of the bloom filter, number of hash functions and number of elements added to the set. The probability of reports of false positives increase as the number of elements inserted increases. Fig 2 [1] shows an example of using Bloom Filters in a real world application.

Fig 1. Overview of a Bloom filter

Add:
\[
\begin{align*}
    h_1(a) &\mid h_2(a) &\mid h_3(a) = 0000000100001010 \\
    h_1(b) &\mid h_2(b) &\mid h_3(b) = 1000001100000000
\end{align*}
\]

Bloom Filter: 1000001100001010

Positions 8 and 3 have collisions,

Query:
\[
\begin{align*}
    h_1(q) &\mid h_2(q) &\mid h_3(q) = 0010010001000000 \\
    h_1(z) &\mid h_2(z) &\mid h_3(z) = 0000010010000000
\end{align*}
\]

Fig 2: Example of Bloom Filter

The number of hash functions used can have a significant impact on the performance of the Bloom Filter. Data element x is inserted and queried using the algorithms [1] below
An ideal Bloom Filter should have the hash functions mapping to a number distributed uniformly over the range. A Bloom Filter constructed using the pseudo-code above will use up $O(n)$ space and has a computational complexity of $O(1)$ in answering membership requests. The advantage of Bloom Filters is that they allow for efficient storing and represent of data. However, they have a disadvantage of sending false positives when being queried for membership. The probability of false positive is given as:

$$\left(1 - e^{-kn/m}\right)^k$$
Where \( k \) is the number of hash functions, \( n \) is the number of elements entered and \( m \) is the size of the Bloom Filter. The authors suggest that the number of bits \( m \) for a given number of elements \( n \), and given false positive rate \( p \) is given by:

\[
m = -\frac{\ln p}{(\ln 2)^2}
\]

The authors also touch upon hashing schemes and explain double hashing in the paper. Double hashing is a method of generating a pseudo-hash \( h_i \) from two hash functions, \( h_1 \) and \( h_2 \) and function hash index \( f(i) \).

\[
h_i(x) = h_1(x) + h_2(x)*f(i)
\]

Also available is partitioned hashing, where the hash functions in Bloom Filters are assigned only part of the entire bit space of the Bloom Filter. The authors also have touched upon variants of Bloom Filters. One of the variants is the Counter Bloom Filter that adds a counter to each of the \( m \) bits in a Bloom Filter to keep track of insertions and deletions. Another variant is the d-left counting Bloom Filter that is based on d-left hashing and fingerprints. It divides the hash table into \( d \) subtables of equal size. Each bucket has \( c \) cells, which store fingerprints \( (f_x = H(x)) \) along with a counter. The fingerprint calculates the hash for each element entered in a cell. Another variant is the compressed Bloom Filter that is associated with systems where sending Bloom Filters across nodes as part of a communication is of importance. In a compressed Bloom Filter, the number of hash functions is chosen so that the bit string representing the elements is reduces to half that of a regular Bloom Filter. A deletable Bloom Filter is another variant of Bloom Filter introduced with purpose to reduce false positives. It ends up removing an element if there a collision detected in the bit string representing the Bloom Filter. Hierarchical Bloom Filters work by splitting the element into different blocks and inserting them into a standard Bloom Filter after concatenating them with an offset. The main effect of this is to lower false positives. Other Bloom Filters discussed in the paper include Spectral Bloom Filters, Bloomier Filters, Decaying Bloom Filters, Stable Bloom Filters, Space Code Bloom Filters, Adaptive Bloom Filters, Variable Length Bloom Filters, Filter Banks, Scalable Bloom Filters, Dynamic Bloom Filters, Generalized Bloom Filters, Distance-Sensitive Bloom Filters, Data Popularity Conscious Bloom Filters, Memory Optimized Bloom Filters, Weighted Bloom Filters and Secure Bloom Filters. While the paper does not go into the details of implementation of each of the Bloom Filters, they provide a good reference as to when to use these Bloom Filters based on the requirements.

Paper: A Multi-attribute Data Structure with Parallel Bloom Filters for Network Services

By Yu Hua and Bin Xiao

Introduction

This paper proposes a new data structure to support multi-attribute data representation using Bloom Filters. The structure is made up of parallel Bloom Filters and a hash table for accurate data representation and query. It uses a two-step verification process to reduce the probability of false positives from the Bloom Filters. The paper also leverages on the fact that network APIs require multiple parameters to be passed as part of queries. The aim of the authors is to preserve the dependency between multiple
attributes in designing the data structure. The hope is to allow the dependency information to better represent the data element so as to reduce the false positive probability. The Parallel Bloom Filter (PBF) has a matrix having counters and is comprised of submatrices. Each submatrix is designed to store an attribute of the data element. A hash table stores a verification value for an item. It is part of a two-step verification process that queries the presence of a particular item. This is present because the PBF alone is not sufficient to query for a data element’s presence. The data structure also has counters to facilitate adding, querying and removing items with constant time complexity. The authors present prior work before proposing the data structure.

Proposal
The proposed data structure is an offshoot of the counting Bloom Filter. It has a PBF and hash table as part of it. PBF uses the counting Bloom Filter to facilitate deletions. A figure representing the proposed Parallel Bloom Filter is shown below:

![Parallel Bloom Filter Diagram](image.png)

**Fig. 1. The proposed structure based on counting Bloom filters**

The authors view it as a matrix containing submatrices. The m submatrices represent p attributes of the data elements. A submatrix consists of q parallel arrays to represent one attribute. An array consists of m counters and is associated with one hash function.

If $a_i$ is the $i^{th}$ attribute $H_{ij}(a_i)$ represents the value of the $j^{th}$ hash function where $1 \leq i \leq p$ and $1 \leq j \leq q$. Therefore, there are $m \times q$ counters and uses $m \times q \times p$ counters in all.
The hash table is used as a verification mechanism. If $v_i = F(H_{ij}(a_i))$ is the verification value of the $i^{th}$ attribute of data element $a$, verification value of $a$ is given by $V_a = \sum v_i$ for $1 \leq i \leq p$. As mentioned earlier, the major role of the hash table is to find the dependency between the attributes of the data element in the Bloom Filter. For a test for the presence of an element, both the hash table and the parallel bloom filter should return a value of true. This theoretically reduces the false positives. The hash table can use the verification values to represent the data elements. The way this is done is by summing up all the hash values and storing them in the hash table. However, it is possible that more than one element might end up having the same summation value. In order to prevent this sequential information of hash functions is used to distinguish between verification values of different data elements. This is done by allocating different weights to the hash functions to reflect the difference in hash functions. The value of the $i^{th}$ element is represented as $H_{ij}(a_i) / 2^j$ for the $j^{th}$ hash function. This value is then inserted into the hash function. According to the authors, the addition operation into a PBF takes $O(1)$ complexity. It is given by the figure below:

---

**Fig. 2.** The algorithm of adding an item with multiple attributes

---

The array counter values are initialized to 0. The insertion operation updates the Bloom Filter counter by one at $PBF[H_{ij}(a_i)]$ to represent the addition of data elements into the Parallel Bloom Filter. A query that reaches the Bloom Filter for existence of a data element will return false if the counter value in the array is 0. When an element is removed, the array counter value is reduced by 1. After the membership query is checked for value 0, it goes on to check whether the hash table has the data element value if the counter array value is not set to 0. The pseudo-code for the query is shown below:
Fig 3. The algorithm for querying an item with multiple attributes

<table>
<thead>
<tr>
<th>Membership_Query_Item (Input: Item a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize $V_a = 0$</td>
</tr>
<tr>
<td>for ($i = 1; i \leq p; i++$) do</td>
</tr>
<tr>
<td>\hspace{1em} for ($j = 1; j \leq q; j++$) do</td>
</tr>
<tr>
<td>\hspace{2.5em} Compute $H_{[i][j]}(a_i)$</td>
</tr>
<tr>
<td>\hspace{2.5em} if $PBF[H_{[i][j]}(a_i)]==0$ then</td>
</tr>
<tr>
<td>\hspace{4em} Return False</td>
</tr>
<tr>
<td>\hspace{2.5em} end if</td>
</tr>
<tr>
<td>\hspace{1em} $V_a = V_a + \frac{H_{[i][j]}(a_i)}{2^j}$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>if $V_a$ is in the hash table then</td>
</tr>
<tr>
<td>\hspace{1em} Return True</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>Return False</td>
</tr>
</tbody>
</table>

Fig 7: Pseudo-code representing membership queries in a PBF

Results

The authors found that the Parallel Bloom Filter (PBF) shows improvements over a basic Bloom Filter. The false positive probability for a PBF ends up being around 0.001 per cent as compared to 1 per cent in a traditional Bloom Filter for a value of 320 for the number of counters. Also noteworthy is the fact that the traditional Bloom Filter ends up reaching a plateau. However, with the PBF, it is seen that the increase in count numbers ends up in a slightly steeper ascent for false probability numbers. The figure below shows the results obtained by the authors:
PROBLEM STATEMENT

From the paper we have read on data popularity conscious Bloom Filters, we know that having extended caches improves the performance of a Distributed Cache. This project will look at how the increase in the number of masters will end up affecting the performance characteristics of a Distributed Cache.

Expectation

Before proceeding with the running of the simulator, it is important to know what the expected results might be. From the prior work done, it is reasonable to expect that for a given cache size and algorithm, as the number of masters increase the hit ratio for the cache will increase. Also, it is to be expected that the miss ratio would be a direct complement of the hit ratio if Bloom Filters are not used. Using Bloom Filters might skewer the ratio graphs more because of the presence of false positives.

The following simulations would try to understand the effect of masters on the performance characteristics of a Distributed Cache.
In order to design a Distributed Cache, it is important to take into consideration the kind of data being requested, the manner of communication and the manner in which the response is sent back to the client. In the paper on summary cache, the authors have implemented an IPC protocol based design so as to compare their system with ICP. In most other work related to Distributed Cache, having simulators mimic the performance of caches is accepted as a reasonable design for measuring cache performance. We employ a design based on a simulator where we assume that the network parameters are comparable to I/O metrics.

The simulator is used to simulate the Block ID requests from clients to the master. The master will process this data by querying the cache and send the request back to the client. If not, it will verify for the presence of data in other clients and if the simulator detects other clients having data, it will redirect the client to the client having data. If the data is not found in the other client’s cache, a request to gather the Block ID from source is sent to the master.

Also of importance is the manner in which the simulator is run. One of the ways to run the simulator is to hard code each value for parameters and start the simulator. However, this is cumbersome and requires a lot of manual intervention. On the other hand, having the maximum limits for configuration parameters like number of masters and clients, cache size etc. would set an upper bound on the parameters. Once we decide on a lower bound for parameters and interval step, we can run the simulator for all combinations.

Data Representation
We’ve seen prior work related to Distributed Cache focusing on data representation in terms of Bloom Filters. The reason for this is that one of the most frequent use of cache is in accessing data over the internet using URLs. The cache stores data from these URLs to distribute it to the client requesting them. The high volume of data and the string representation of URLs require that the data be represented efficiently by occupying less space. Bloom Filters provide a good opportunity to do this because the arbitrary length data can be reduced to fixed bit size representation using hash functions.

In this simulation, we use the simulator to process data. Since the data is represented as Block IDs in as integers, we may not need the use of complicated Bloom Filter implementation. A regular collection provided by standard programming languages will be sufficient for such representations. Using these representations for data query, access and retrieval should suffice for the simulator to run effectively. Our project has been implemented using C++11 with threads and Standard Template Library (STL). Hence, we will not involve hash functions and Bloom Filters in our implementation.

Fig 9 below shows the functioning of the distributed cache with 3 masters and 4 clients. In the figure, the cache summary is shared across masters. However, since each of the masters is responsible for sharing the workload, it is sufficient for the masters to have their own copy of the summary cache. The cache stored in masters are read-only cache and do not support updates as of now.
Fig 9: Representation of a multi-master Distributed Cache

Protocol
One of the important design considerations to take into account is the communication between masters and clients. If we are to measure the performance characteristics against a Distributed Cache like ICP would be to design a simulator using IPC protocol. However, developing a simulator that uses socket programming is beyond the scope of this project. Hence, we use threads to communicate across various class objects using reasonable design patterns.

Traces
In order for the simulator to use up the queries, the clients have to be able to send a sequence of Block IDs that mimic real life scenario effectively. One way of doing this is for each of the $k$ clients to send $m/k$ queries, where $m$ is the total number of elements in the entire dataset. However, this is not an optimal representation of a real life scenario. In actuality, multiple clients may end up querying caches using random Block IDs or URL. Another option would be for all the clients to just repeat the entire dataset. This too would not be an effective reconstruction as most clients end up querying the cache for some familiar and often used URLs while most others will be completely random data sets.

The trace vectors created have to use a combination of overlapping and random Block IDs to query the masters. The vector in this particular simulator was a combination of sequential and overlapping queries that all the clients query simultaneously.

Warm Up
It is reasonable to expect that a cache without sufficient data in the cache would end up driving the miss count up. This is because the cache has no data in it when the queries are first being sent. Therefore, it will have to end up querying the data at source or disk to get it in its cache before sending it to the client.

In order to prevent this, the cache is first warmed up with random data from the dataset. This way, there is a sufficient representation of simulation data within the cache in the master before the first Block ID is queried for by any of the clients.
Measures

When the simulator runs and fetches the results for the clients to use, it is important to have the right parameters to judge the performance of the cache. One of the way of doing this is to measure the time taken for the request to be returned from the masters. Another way to do it would be to keep track of the time taken for a Block ID to be queried by clients, processing in the master and returning it to the clients. However, these methods have an inherent disadvantage. It is that the system clock time might not be the best representation of the activity in the masters. It could be that the system time would also include the CPU using up time taken by other applications and processes running in the system. Another reason why time is not an accurate representation is because in extending this work to a simulation across nodes in a system, it is possible for 2 systems to have their system clocks showing differing times or belonging to different time zones. It is therefore important to come up with another reasonable measure of performance of the Distributed Cache.

Another measure would be to keep track of the number of times a Block ID is being fetched from the source as it is usually the determining factor for number of misses. The disadvantage of this measure is that it would only end up measuring the disk reads and not the cache hits. Measuring cache hits is an important determining factor representing the performance of a cache.

A third way to deal with this issue would be to trace how many times a Block ID is in the system by using a counter. This tick will be incremented for every change – once for sending the request to the master, once for client redirection and once again when the data has to be accessed from disk. So a cache hit will increment the tick count by 1, client redirection will increment the tick count by 2 and disk reads will end up increasing the tick count by 3. This is a more effective way of measuring how often the system is busy with data Block ID requests.

Replacement Algorithm

The replacement algorithm generally plays an important role in a Distributed Cache. One of the ways this affects the cache performance is by affecting the hit and miss ratios in a cache. The reason being that the sequence of Block ID queries representing the entire data set is almost always greater than the cache size itself. It is therefore imperative that the manner in which these data blocks in the cache are replaced is an important factor in determining how the next request for the same data block is responded to. From the prior papers discussed in this report, we know of the LRU and LFU algorithms and how they affect the cache performance. There are many more replacement policies that can be employed, even within the same replacement paradigm. However, in this project, we are concerned more with the effect of masters and hence employ only a basic LRU model. We also study behaviors based on Random replacement algorithm.

Masters

Since the project’s scope is to study the effect of the number of masters present in a Distributed Cache, it is important to design the masters effectively. A master will contain a cache and keep track of Block IDs being used by all the clients it services. It should also keep track of all the tick counts. This is done by keeping an individual counter for each of the masters and use a method to dump the final result in a static
counter. The masters will have functions to let clients know whether the data is in cache and whether it is in some other client. Apart from these, the master will also have a function to pull data from source.

Clients
The clients have to interact with the trace vectors and masters to send the Block ID request to the right master. The client is responsible for initiating a request and updating the tick count when they send the request to the master. Each client runs through the entire trace vector. It should also be responsible for verifying whether the redirected client information is correct, i.e. the redirected client has the same Block ID as the current client. Once the simulation is complete, the clients should also provide information about incorrect redirection to the caller. Such a design makes sure that both masters and clients can work in tandem under different implementations of the simulator without any major changes in their respective codes.

IMPLEMENTATION

Configuration File
The configuration file is the master file that sets the limits for all the parameters being used. The list of parameters being parsed for are total number of data elements, maximum size of the cache, maximum number of masters, maximum number of clients and the debug level being used. The debug value can take values of 1 and 2. The debug value of 2 is a verbose representation of all events happening during a simulation run. Runs take longer while using the debug level 2.

The simulation assumes that there is at least 1 master and 10 clients with 20 as the cache size. These represent the minimum values of masters, clients and cache size respectively. The main file parses the configuration file for these parameters and sets its respective value in the DataEngine class. Once the limits are set, the simulation launches an instance on the DataEngine class and calls a startSimulation function.

Data Engine
The data engine class is used as a driver to run the actual simulation. The simulator starts with the minimum values of the parameters – 1, 10 and 20 for masters, clients and cache size respectively. The masters are incremented by 1, clients by 10 and cache size by 20 as part of every simulation run till the maximum limits are reached. For every call to the run, the same process repeats – creation of trace files, creation of masters and clients, warming up the caches and send queries from client to masters.

The trace vectors are created by referring to the maximum data entries parameter. It is divided into sequential and overlapping Block ID queries that is sent by the clients to the masters. Once the trace vector creation is complete, the next step of creating masters is executed. It ends up creating a vector of pointers to the new masters created so the clients can query them for their query inquiries. Once the masters are created, the same process is carried out to create clients. Heap memory is used to allocate memory for new client creation and the pointers are stored in a vector. Once the clients are created, the function proceeds to warm up the cache in each master by populating them with random numbers within the data limit parameter.
so that the effect of an empty cache affecting the hit ratio negatively is avoided. The warmup is an uncomplicated function that creates a local copy of the trace vector in every master. This avoids the same warmup vector to be run each and every time a master is created. It also sets up the clients to run the sendQueries function. This is done by creating as many threads as there are clients simultaneously. Each client is passed a reference to the vector containing all the masters along with the trace vector that contains the Block ID to be requested from the master.

Once all the simulations are run, the threads are joined into the main thread and deleted. The clients and masters are then queried to dump their results into a data file. The data files are named uniquely based on the parameters that are set up in the simulation run. The result set in the data file contains tick counts for each block, number of cache hits, number of disk reads, number of correct client redirections and number of incorrect client redirections.

**Master and Client**

The client class contains functions to send queries to masters requesting Block ID. They also contain variables that keep track of the number of correct and incorrect redirections from the master. Also present are mechanisms to dump these results into the calling function which accumulates these results to be dumped into the data files.

The client receives the list of masters in the form of a reference to vectors. It also gets the list of Block ID to request the master. The client identifies the master that could contain the Block ID using a simple distribution. If there are m masters and k data entries, each master will handle k/m requests. This guarantee uniform distribution of requests across masters. Once the correct master is identified, the client checks whether the master chosen has the data in it. If it is present, it ends up requesting the master to provide data from the cache. If the master has data in the cache, it returns it and increments the block tick count. If not, the client is redirected to one of the other clients if the master presumes that there is another client that is accessing the same data. Since the masters are responsible for only part of the data set, it is not required to query other masters to find out the right candidate holding the Block ID. If the master is unable to provide the required Block ID, the client ends up requesting the master for information from the source or disk. This is counted as a disk read by the master, which updates its instance variable which will end up dumping this count into the data engine’s data file.

Once all the trace entries have been queried, the control reaches the data engine’s calling function that proceeds to collate and collect data from the clients and masters and dump them into the data files. Once the data files are gathered, the function ends up releasing the resources, thereby destroying the masters and clients, preparing the program for the next set of simulations.

**Post-simulation**

The data files themselves are of little value directly as they do not serve the purpose of studying the effect of varying masters. This is because each file is a representation of a fixed number of masters. In order for meaningful information to be extracted, we require a method of collating the information. Since this
project was developed on a Linux Operating System, we have used a shell script which uses the join command to collect data based on column entries in the files, which are sent as arguments. Since each Block ID and its corresponding tick count is sent to the data file, it is possible to use the join on the Block ID. The only manual intervention required in the shell script is make sure the parameters in the shell script match the parameters set in the configuration file.

The script filters out the data files that have varying masters and runs the join command on the first column of all these files, taken two at a time. The results are placed into a results folder. All the result files are compressed using a tar command. The only requirement is to see that the shell script variables match the configuration file parameters used for the simulation.

Logging
The application developed for the project consists of a logging and debugging feature that logs each event to a log file. It is implemented using a singleton pattern which assumes that there will only be one instance of the class in the entire application.

The logger uses two levels 1 and 2. Debug level 1 is a basic logging feature while debug level 2 is used for more advanced logging purposes. The disadvantage of using debug level 2 is that the computation and clash for resources would end up making the simulations slower. Therefore, it is preferred to have the logging level at 1.

Utility
The simulation source also uses a utility header file that consists of a list of commonly used functions that are used in parsing, debugging and logging operations.

RESULTS
The result files were collated to run and collect graphs showing the effect of various parameters on the performance of the cache. Once the graphs are generated, we are able to see the performance characteristics more succinctly and draw conclusions from them. The intermediate driver result files have data representations whose sample graph can be given as below for 60 clients and 120 cache size:
These intermediate graphs are indicative of what is happening with the simulation. Also important to note is that the hit ratio sees a steep increase after a while. It is important to note that the only difference is seen where the rise in hits occur and more importantly, how quickly. In the intermediate graphs for cache size 120 in Fig 10, we find that the slope change happens at around 8 masters. However, for the graph in Fig 11 with cache size 200, the increase is at around 5 masters. This is seen as being consistent across different masters and client numbers.
These form the intermediate results which are used to construct the larger consolidated results. The larger consolidated results are used to draw conclusions on the effect of larger number of masters in the parameters determining the performance characteristics of a Distributed Cache.

Effect of varying cache

The simulator was run for both varying master numbers and varying clients. In this case, we gathered data representing varying masters running a Least Recently Used algorithm. The results had collected data samples for varying cache sizes from 100 to 200. It is reasonable to expect that the number of clients should neither be too small nor too large. Therefore, a client count of 60 was chosen to be an adequate representation of the data being studied.

For a list of 60 clients having LRU algorithm, the following results were obtained. The hit ratio measured in terms of direct reads from cache is mapped against the master count increased incrementally, as shown in Fig 12. The hit ratio sees a steep climb after a while. The graph is significant because it effectively says that the spike in hit ratio is almost the same in all cache sizes but more importantly, the spike occurs earlier as the cache size increases. This means that the hit ratio becomes higher faster as the cache size increases.

Graphs for hit ratio and miss ratio complement each other. As in the case of hit ratio vs number of clients in the case of miss ratio drops significantly faster when the size of the cache size increases. This is consistent with the expectation of miss ratio after having seen the hit ratio graph. This also shows that there are no other sinks in the measures as the simulation runs.

It is also noteworthy that the different cache parameters having the same overall cache size have similar hit ratios. From the graph above, we see that 10 clients having cache size 100 has a hit ratio that is close to 5
clients having cache size 200. Another thing to note is that these numbers are closer when the cache is performing to capacity. That is, when the cache is still in the warmup phase, the hit ratio numbers for overall cache size is not as close.

Fig 13: No. of masters and Miss Ratio

Another important measure that determines the performance of cache is the number of wrong blocks returned. This is done by calculating the times the requests for disk reads happen. It is noteworthy that as the cache size increases, the wrong blocks returned drops by a significant amount faster. This means that as the number of masters increases, the wrong blocks returned by the cache drops after staying up high. This drop gets faster as the cache size increases.
Fig 14: No. of masters and wrong blocks returned

Fig 14 shows the effect on correct client redirections for varying cache as the number of masters increase. The simulations have been run using LRU. It shows that the slope of correct client redirections drops significantly. By the first look at the graph it seems like this is a bad thing. However, a closer inspection along with the information we already have, it is evident that the drop in client redirection number has to be taken into account in conjunction with hit ratio. That is, as the number of masters and cache size increases, the hit ratio increases and most of the Block IDs sought are found in the master’s cache. This means that the client is not told by the master to look for the information in other clients before requesting the master to access the data from disk. This explains why the redirections plummet to a stop.
The figure below shows the effect of varying masters on the total tick count for all the Blocks. It is noted that the numbers drop down to a flat line after a while. It shows that the number of redirections overall comes down drastically as the cache size increases. This, in conjunction with the hit ratio graph is evident of the fact that the cache size improves cache efficiency in terms of direct hits in the cache. When the cache size increases, more blocks are stored in the cache. Therefore, more queries end up querying the master’s cache than disk reads, thereby reducing the tick count.

Fig 15: No. of Masters Vs No. of Correct Client Redirections

Fig 16: No. of Masters Vs Tick Count
Effect of varying clients

Similar measures have been gathered to study the effect of varying clients on the Distributed Cache. Following observations have been made in a similar fashion to find out if the effect of clients on the parameters mentioned.

It was found that the hit ratio for increasing masters using LRU spikes up after a while. The spike happens at almost the same place, when there are about 5 masters. One other interesting fact is that this is consistent across various client numbers. Also noteworthy is that the hit ratio reaches a plateau with hit ratio close to 1 as represented by Fig 17.

![No. of Masters Vs Hit Ratio for Varying Clients](image)

Fig 17: No. of masters and Hit ratio
As was found in the case of varying cache sizes, the graph for miss ratio complements the hit ratio graph in the sense that there is a fall in the miss ratio as the number of masters increases. The fall is seen at around 6 masters and is consistent across number of clients. Fig 18 also shows that there are no sinks when it comes to measurement of this data metric. It is also noteworthy that larger the cache size, lower the number at which the miss ratio bottoms-out.

As was found in the case of varying cache sizes, the graph for miss ratio complements the hit ratio graph in the sense that there is a fall in the miss ratio as the number of masters increases. The fall is seen at around 6 masters and is consistent across number of clients. Fig 18 also shows that there are no sinks when it comes to measurement of this data metric. It is also noteworthy that larger the cache size, lower the number at which the miss ratio bottoms-out.
Fig 19 represents the number of wrong blocks returned for varying client numbers over a range of masters. The number of wrong blocks returned comes down sharply at around 6 masters. It is noteworthy that the number of clients ends up being a determining factor in the case of lower masters but when the number of masters increases, it slumps down to almost zero. Also of importance is that the client numbers are relevant before the slump. Higher the client count, higher the number of wrong blocks returned. This means that for a given cache size and replacement algorithm, the effect of multiple masters is to increase the available cache size so that distribution of load on the system is distributed in such a way that inaccurate blocks go down at higher master numbers.

Replacement Policy

As mentioned earlier in the report, the effect of replacement algorithms is evident in the hit and miss ratio. This has been found to be true using the simulation for 60 Clients and cache size of 160. This was measured for both random replacement policy and Least Recently Used (LRU).

As can be seen from Fig 20, both the LRU and Random algorithms cause an increase in Hit Ratio with increasing number of masters. However, it is interesting to note that the hit ratio rises more gradually by using a random replacement as compared to LRU, where the spike is stark. Also noteworthy is that the hit ratio ends up plateauing at a higher number for LRU, reaching close to 1. With a random replacement algorithm, it is seen that the hit ratio ends up at 0.8.

![No. of masters vs Hit Ratio for Different Replacement Algorithms](image)

**Fig 20:** No. of masters and Hit Ratio

Fig 21 represents the response to miss ratio by varying the number of masters and differing replacement algorithms. We find that the miss ratio drops down gradually in case of random replacement algorithm while in the case of LRU, it is seen that the miss ratio plummets down over a number of master increments. It is also noteworthy that both the graphs end up complementing their respective hit ratios.
The effect of replacement algorithms is stark when it comes to the accuracy of blocks being returned as shown in Fig 22. In case of using LRU, we see that there is a slump in number of wrong blocks returned as the number of masters increases. It ends up flattening out at around 7 masters close to zero. However, when it comes to the point of random replacement algorithm it is seen that the number of wrong blocks remains almost constant, hovering at between 5000 and 7500. It shows that the choice of replacement algorithm affects the performance in which Distributed Cache performs.
CONCLUSION

From the results seen above, it is seen that the number of masters increases the hit ratio. The hit ratio hits a plateau after seeing a spike. This shows that the effect of cache warmup is minimal on the hit ratio when the number of representative data is high for querying. Also significant is that increasing cache sizes makes these jumps occur faster. This shows that the cache size is more of a determining factor when it comes to hit ratio getting high. The effect of varying clients on hit ratio is minimal as the behavior is consistent across client numbers. The effect of miss ratio on varying cache size and varying client numbers are complementary with their corresponding hit ratios. The miss ratios slump down and end up remaining constantly high for quite a while.

The number of wrong blocks returned after client redirections is a noteworthy measure. In case of varying cache sizes, the number of wrong blocks plummets faster as the cache size increases. This means that the effect of varying masters in conjunction with cache sizes is a determinant of the rate of decrease in number of wrong blocks returned. Also important to note is that this slump is similar in case of varying clients too. However, the difference is not as evident and as far between as the graph representations for varying cache sizes. The curves plummet at almost the same time. This means that the presence of number of clients are not relevant when measuring the wrong blocks returned.

Also important is to note the effect of cache size in case of having multiple varying masters. For example, when we analyze the hit ratio of 10 masters having cache size 100 with that of 5 masters with cache size 200, we note that the values are close to each other. Also of importance is the fact that these numbers of hit ratio for same overall cache size is more far apart when the cache is still in the warmup phase. It suggests that the impact of cache size is far more important than having multiple masters.

While it is evident that the performance measures show improvement in having multiple masters, it is important to note that the project’s scope did not cover the implementation of a read-write cache. That is, the simulator has been developed for reading operations alone, its performance while significant updates and writes happen has not been studied in depth.

From the project and the results gathered using the means mentioned, it is seen that increasing the number of masters has a positive effect in influencing the hit and miss ratios. However, when it comes to the accuracies of the data blocks returned, the number of masters matters very little. It would entail a study of the impact of varying masters and in conjunction with the data structures and replacement policies.

FUTURE WORK

From the simulation results, we see that having multiple masters does have a positive effect on distributed cache performance, in that the hit ratio increases and using it in conjunction with LRU makes it reduce the number of wrong blocks returned. Although we have seen that the overall cache size is a significant determining factor in cache performance, it is impractical to expect a single cache to have
unlimited cache size. Therefore, it is important to look at how we can increase the overall cache size by increasing the number of masters.

The next step would be to study the effect of other replacement algorithms on the simulation with multiple masters to incorporate additional parameters for data representation. Multi attribute data entries and their effect on cache performance is still not studied to the extent expected. Considering that most Bloom Filters end up making very little differentiation between all the various attributes, it will be a natural progression for the simulator.

Also of importance to look at the simulation performance when the communication occurs through sockets, especially sockets with IPC protocol. This might give us a better indication of how the performance changes when the simulation gets promoted to the real world environment. This also gives a valid performance measure against the ICP cache, which is the standard against which the Summary Cache was measured.

Another area to study would be to look at the influence of data structures in the performance of the Distributed Cache. While there are many variants of Bloom Filters, the application of this simulation in real world scenarios would entail the use of larger and varying data. This would mean that not only would integer based Block IDs become irrelevant, it would also mean that data may end up being multi-attribute string values representing URLs.

Finally, making the simulator run updates as it is servicing the clients would be the most important step. This is because it is expected that making the cache read-write would end up bringing the performance characteristics like hit ratio down. When the updates to data structures represented in the cache take place, there would be a block that might end up slowing the cache down. It would be an interesting idea to study the effect of a read-write cache on overall performance.

REFERENCES


ACKNOWLEDGEMENT

I would like to thank Prof. Dr. Hans-Peter Bischof for his inputs and guidance in bringing this project to fruition.