Distributed Collaborative Caching

Wardwell, Steven  
Contact Author  
Department of Computer Science,  
Rochester Institute of Technology,  
70-3005 102 Lomb Memorial Drive,  
Rochester New York, 14623 United States  
sjw1579@rit.edu

Bischof, Hans-Peter  
Department of Computer Science,  
Center for Computational Relativity and Gravitation,  
Rochester Institute of Technology,  
Rochester New York, 14623 United States  
hpb@cs.rit.edu

Abstract—In this paper we explore distributed collaborative caching. Caching in distributive networks can improve data access performance and reduce dense communication between the client and the server. Collaborative caching can improve caching by allowing clients to share and organize (coordinate) their cached data. This paper will describe our profile algorithm and its implementation and algorithms that presently exist in the literature and their implementation. The object is to establish a baseline of current research existing at the present time to allow for implementation of our proposed algorithm. We will evaluate the algorithms for efficiency, accuracy, disk or server access, hit and miss rates etc. Our presentation of our algorithm of profiling client cache data will consist of constructing a real time profile of cache data to predict a likely client(s) match for data requests. Simulation and graphing analysis will show the effectiveness of our scheme.

Keywords: Profile, Distributed, Collaborative, Adaptive, Cache, Score

1. Introduction

In various networks, file sharing and data sharing are widely used. One such technique to improve the performance of wired, wireless or distributed network sharing is caching. The aim of this type of caching is to reduce traffic, congestion, data accessibility, and bandwidth management of resources, such as battery power [5]. Caching schemes typically do not facilitate data access based on knowledge of distributed data schemes [5]. Our work attempts to bridge this gap between data access and knowledge of distributed data. In simple caching, it is the requesting node that performs the caching. This cached copy is then utilized by the node to service subsequent requests, as they are needed. In most cases, the requester must retrieve data from the data center or in our case, the "Server/Disk" when missed cache requests occur. The caching of frequently requested data in distributed environments can potentially improve data access, performance and availability. Cooperative or Collaborative caching in distributed networks allows for the sharing and organization of cached data between and/or among various clients or devices and groups[2]. Things such as mobility and resource limitations, as well as limited device battery power, make some collaborative caching management schemes more and more attractive [2]. The ultimate goal is limited server and/or disk access.

In this paper, we propose a distributed collaborative caching scheme based on previous algorithm work, simulation and evaluation. We further suggest our own "Cache Profile" scheme which builds off and utilizes previous ideas and incorporates them into our framework and simulation. Within this paper, we describe the design of our basic framework or simulation architecture and its clients, Content Manager, thread communication layer, logger, various scripts and server. We will describe the basic idea of our cache profile mechanism and what constitutes a "good" profile among client caches. We will describe how each client will use a similar mechanism to the "Summary Cache" [3] to update the Content Manager with its profile at various times or percentages of cache data. We will also describe how the Content Manager will act as a pseudo intelligent proxy, much like how the "Adaptive Cache" [2] mechanisms work in our earlier researched algorithms that evaluate each clients broadcast profile and group/cluster like profiles. It is based upon various criteria or threshold and will mark each related client in the cluster from a strongest to weakest match. This clustering idea was based on previous work found in adaptive collaborative cache mechanisms [2]. We will also describe the mechanism by which the client will retrieve its matching cluster information. It will make a direct request on a fellow node or client for fast retrieval to make use of limited resources and battery power in real world applications.

We will also show, through graphical analysis from simulation studies, predicated actual results of our work. Our contribution will be a unique way of viewing profile data from client caches, making a semi-intelligent Content Manager while cutting down on the number of hops to clients and server requests. The data retrieval among clients will form a collaboration of their data which will be beneficial for limited resources in distributed networks and further the
2. Related Work

There has been a vast quantity of work performed in the area of collaborative cache and distributed collaborative cache to improve performance, such as "Summary Cache" [3] and Adaptive Caching algorithms [2]. Most of the adaptive work has been carried out on MANETS and mobile networks. We have applied that work to our static distributed simulation work. In the "Summary Cache" [3] algorithm, different proxies (every other) store a summary of client’s cached data in something called a directory of data. The local proxy shall be required to check the stored summaries to determine if the requested document might be present in other proxies. If the document is not found, the request is then forwarded on to the web server [3]. The summaries are not required to be up-to-date or accurate. The updates may occur at regular specified intervals or at the time a certain percentage of the cached documents are not reflected in the summary (unchanged data) [3]. Our representation of this summary proxy is reflected in the content manager that performs the same work.

Other related work is known as "Adaptive Caching with Heterogeneous Devices" [2]. This work was performed in a mobile peer-to-peer network. Our work is a static distributed network, but we have adapted it to basic ideas from this paper. Many applications today, such as file sharing or multimedia streaming such as Netflix [6], are widely used in wireless networks. Different users may carry heterogeneous mobile devices with different transmission ranges, latency, and cache sizes [2]. Service is usually provided by Mobile Support Stations [2], much like the proxy mentioned previously or similar to our Content Manager. Mobile peer-to-peer (P2) networks can also provide services to each other. This P2P would require a cache mechanism that could handle heterogeneous devices and sharing among themselves [2]. This paper [2] proposes a cache scheme that is adaptive to the actual device condition and its neighbors (clients). In this scheme, a distinction is made between what they call "Strong" peers that retain popular data to do self service work. This protects what they refer to as "Weak" peers with their limited cache space. We have adapted their work to fit our simulator, using what we call "Client-to-Client" services.

The "Adaptive Caching with Heterogeneous Devices" [2] paper also builds from another called "A caching and Streaming Framework for Multimedia" [4]. It investigates the middle ground on caching and streaming technologies for multimedia [4]. The algorithm described in this paper is meant for broadband architectures. The foundation of that work is based on the internet standard "Real Time Streaming Protocol" or RTSP [7]. They also use a proxy manager or what they call an "Intelligent Agent" or "Broker", much like our Content Manager that has some "intelligence". Their scheme is much like an enhanced RTSP that maintains state information, similar to the "Summary Cache" [3]. The broker contains their cache algorithm’s and runs them, our algorithms are run in the individual clients cache and/or the "Content Manager". The main goal of their work attempts to create the right model for RTSP to perform broker-based streaming/cache architecture.

Our work for "Distributed Collaborative Cache" builds upon the previous work mentioned by simulating the algorithms, environment and incorporating it into our "Cache Profile System". It utilizes the proxy/summary basic idea and intelligent idea while applying them into our individual clients and Content Manager.

3. Algorithm

The algorithm is our "Profile Algorithm". It is often called "SPro" for lack of a better term. This algorithm was built using some of the ideas from the Summary algorithm [3] mentioned above. The basic idea was derived from reading countless collaborative cache and adaptive cache papers. No one paper is responsible for the general idea, except for ideas from the Summary algorithm. Part of the idea was born from implementing and running cache algorithms while observing and studying their results. Also, it was helpful to study and analyze actual trace data while observing how the method runs and the results. The last contribution that led to the idea came about from studying the various replacement policies and then implementing some on a small scale. We found LRU was the most interesting to begin our algorithms investigation. We discovered how one could "stack" the trace data to eliminate the need to access disk or content manager request or trip. We found this useful in our understanding and design of our algorithm. From watching the updates to the summary algorithm (since we implemented this algorithm in our simulator and others to evaluate) and while reading trace data, there always seemed, on average, a certain "pattern" to the cache and its current trace data. This led to the idea that each client must have some type of data "signature" to its cache at any given time and over time. We started to investigate this idea, as well as reading mathematical publications [18] [19] [20] [21] [22] [23] to determine how to profile or recognize similar groups of numbers.

From there, we began with the idea that we could build on the summary algorithms timer update method to update information back to the content manager to perform a type of mathematical calculation. This would allow the client to build an up-to-date profile of its cache data at that given time. We also developed a scheme to build the profile as quickly as the data streams into the cache. We sought to build off the idea that we could use an idea similar to a heap to calculate the score in real time using the efficiency of a heap, comparable to a "score heap" (our term for it).

This idea was gleaned from [29] and from papers such as the FSR [2]. We noticed most distributed cache or networking systems deal with power, bandwidth, frequency
and hop issues. These examples tend to lead to distributed systems not hopping or going to each client in the network for data or trace requests. We built off this idea, as it was our desire to limit the number of hops and bandwidth while utilizing the profile algorithm in order to group or segment similar client cache profiles within an adjustable tolerance or threshold.

Also, borrowing from the summary algorithm idea of compact summaries idea or proxies [3], we used this to create a small compact table that uses a group id and uses the client id. Both are integers. In future work, it could be possible to make this more compact or even smaller using some sort of binary number or other ways. We left this design very flexible for that reason. We made it simple to enable us to prove our algorithms theory first.

We also expanded on the idea that over time, the data or traces seem to repeat. The intervals seem to go in cycles based on a cache life cycle. Using this data, we discovered the "accumulator" parameter which allows clients to belong to many groups or to limit the clients group membership. The theory being that if the clients were in a group at one time, the cycle will repeat for that group and become a match again, increasing the hit ratio or probability. The draw back to this idea, theoretically, in the worst case scenario, all accumulate in the same group, defeating the hop and bandwidth and power limitations. By using the "accumulator" limiter, we can force group re-evaluation and limit group membership to pairs or a small number. If the client is in a group, it forces a re-evaluation and asks, "does this client still belong in this group" and removes it if it does not meet the criteria.

Finally, for the general description of the profiler algorithm, we use all the mathematical calculations to arrive at a final score. We devised the simplest way to combine all the data in such a way to utilize it all in the final evaluation, but quickly. The score is merely a combination of all data; we basically add the total and use the absolute value. Based on the score, we compare clients against each others’ score and use the threshold or tolerance to determine ultimate group membership.

Each client stores its own group data, but the content manager does all of the work by calculations and redirection based on which client is in the group. The request is made and then returns it to the requesting clients.

To build a profile, it is calculated by using six different measurements. These measurements can be improved. We used these as a starting point to prove our theory and study its results. The calculations are Euclidean distance raw and normalized. The obvious mean, median and mode. Finally, the standard deviation figure.

The Euclidean distance is used on the cache trace data within the same cache per client. We do toss out the "extra" data piece if the snap shot of the cache is odd, going on the assumption one piece will not give us inaccurate results. This warrants further study in the future. We also considered calculating the distance between two or more clients caches but felt this may overwhelm the system. It can be explored in the future. What would constitute a good sample to perform this cross distance, if not all? We treat every trace data as a point on an imaginary line to calculate our measurements. We use the same concept on the trace data for a normalized Euclidean distance. This idea emerged from [21] reading and studying this paper. We believed we could achieve a type of second opinion on the result by using this idea.

Finally, in our profile calculations, we use the standard deviation across the cache trace numbers. We desired to use this measurement to produce an overall measure of the degree of desperation of a probability distribution. We compared how far the data or trace points are from the average of the cache traces. This calculation, we believed, was the winner or top contender that gave us an accurate profile of the data [28]. This idea, as well, could be performed across the different client’s cashes. For the same reasons stated above, we did not explore this at this time.

Once the content manager calculates all of the above measurements, it then calculates the score for the clients’ cache snap shot.

The content manager will then set or pass this score back to the client requiring the update. At this point, our algorithm will assign the group or groups from within the content manager. It will perform this across all clients. All of the clients are retained in a table with their respective scores that is updated as the profile is built. This data may be kept in any kind of allocated memory within the content manager, for our needs it is a passed in a thread table to the content manager. The score of the requesting client is qualified by using the threshold from the user and plus/minus as a test to determine if the score is within the range of the client at that point in the table. The content manager will also issue the next group number in increasing order. If there is a match, both clients are assigned to the same group; if not, the group number issued is removed or put back and saved for the next client request. If the accumulation is set, there is no re-evaluation if this client or clients already exist in a group assignment. If it is not set, and if the client has a group membership, it will get re-evaluated for that group and expelled if it is no longer eligible to be a member. This all occurs at the point the update timer fires per the timer setting.

Once the groups are set or the clients are clustered (grouped), the cache is able to make a request for data. It is able to do this without group membership, as there is a default. There is no penalty other than a wasted request, as there will be no match and the content manager will pass operations off to the server.

If there are legitimate group assignments, the content manager will sort through the table and find the matching client in the requesting clients’ group and immediately stop.
It will then move to the client who possessed the match and request the data. If the data is not found, it is passed on to the server.

4. Implementation

Our implementation of our algorithm was built using java. The architecture was developed using Java SE-1.7, so on Windows 7 and other parts on Mac OS X Yosemite version 10.10.5.

The basic architecture in figure 1 called "CCDistSimm" or Collaborative Cache Distribution System, consists of five parts, a client object; a communication layer (java threads) which is the client object; logger/tracer; content manager (brains) and disk/data server. Each client (thread) owns its own cache object. The logger/tracer content manager and disk/data server are singleton pattern objects. The logger/tracer is a singleton (built from using the java.util.logging [11]) to allow us to trace all data through one file while also doing statistical analysis on one file for convenience.

The basic architecture was set to accommodate using a shell script to run the simulator overnight without supervision to gather data. Once data or log files were collected, we used secondary scripts to parse the data. We used tools such as, grep [8] to collect statistics and run calculations to feed into another file to graph. We studied the results using Excel [9] and other programs, such as OpenOffice [10].

The overall run architecture derives from the SimmDriver object which contains the main entry point. When the main entry point commences, an array list is used to collect the command line arguments to send to our SimmIniReader object. This is responsible for handling and distributing the input parameters. Once all of the input or command line parameters are collected, the drive sets up the content manager. It is required for the clients once they start to instantiate. After the content manager is up, the server object is instantiated and its memory size is set. Its memory is filled from a pre-made file. This pre-made file may be any size. At times, this disk memory file took some time or what we call "warm-up" up time, as we had up to one million trace data, in some cases.

Once the server was set up, all clients would be instantiated (but not started, as this is important). This operation was performed using an array of clients and a basic for loop. The reason for utilizing an array was that some operations in the algorithm required knowledge or data from other clients in real time. The best way to relay all client data over to the Content Manager or "brains" of the operation was to collect all client references in an array which was then passed on to the Content Manager. We refer to this as the client table.

This array was then passed or sent to our previously set-up Content Manager in a LUT or Look Up Table method. Once all the various parts necessary for run time operations were set up, all clients were then started. This order is very crucial to prevent the clients from performing operations until all data and necessary basic architecture objects are in place.

The initial trace data was found in this area [12] [13] [14] [15] [16]. Some, if not all, was used. In some cases, this data was combined to diversify the data.

Our algorithm used the general synchronized java key- word to synchronize the thread operations. Great care was used to avoid dead lock.

The "tick" count shown in figure 2 mechanism is our way of tracing service area and a high level way of monitoring the performance of our algorithm and its implementation. This eliminates the need to have the evaluation tied to any processor speed, making future reads of our paper applicable to any processor. Tick counts are tracked in two ways in our CCDistSimm system. The first way utilizes a simple print string in the log called "TICK". This method can be problematic in that it slows down the run process/simulation as it prints the string to log file every tick. The other method utilized was an actual counter in the SimmTickCounts object. This count is incremented every tick count instead of printing a string to the log. Each client object owns its own SimmTickCounts object and presents its final result once the LRU execution is completed. This count is processed by a script to give final tick count numbers. For our convenience, there are times where we use a combination of both methods.

Our algorithm implementation or cache Profile algorithm was set up and executed within our simulator along with the Summary algorithm [3] and FSR [2]. We use an LRU, Least Recently Used replacement policy out of convenience and familiarity with the mechanism and its impressiveness. We found this mechanism able to feed itself with internal hits after the cache size (determined by user) gets larger and proportional to the size of what the data feed (from file) traces in the clients are filled or fed with. As an example, if the file feeding the client has 1000 traces, this number is usually 500 for cache size.

This algorithms main objects are SimmProfileEngine, SimmProfileGroupInfo. The different portions of code are executed based on SimmAlgorithm settings. The first one being SimmProfileEngine which contains the engine behind all the math utilized to build the calculations for the profile generation score. We set this up as a separate object for future modification, as we were ultimately unsure which mathematical operations would best build up our profile. In this way, we add or subtract methods for calculations.
The methods in this object were determined by some research and some obvious measurements using this small sample of site [18] [19] [20] [21] [22] [23]. The methods used are rawEuclidean(), normalizedEuclidean(), cacheAverage(), findDeviation(), calculateMedian() and calculateMode(). The three main methods of significance being rawEuclidean(), normalizedEuclidean() and findDeviation(). We believe the remainder are obvious to the reader why they were picked, but we will briefly touch on it in subsequent sections.

These calculations were chosen as a starting point and can be improved. We drew upon our research and previous algorithm knowledge for distance between numbers.

The first was Euclidean distance [24] shown. We calculate this to be over every two points in our cache, the local cache. We attempted to calculate this over one or more caches between clients, but ran out of time. For odd numbered cache trace data, we discard the non-matched trace. We desired to baseline, therefore, we did not worry about the accuracy hit we would take by discarding one.

Next, we attempted to calculate the normalized Euclidean distance. Some of our references above mentioned the way it can be more accurate than the raw Euclidean distance. It helps scale the variance and quantifies the distance. The raw measure has no bound value for maximum distance. Raw works best under relative ordering for a fixed set of attributes. This also discards an odd valued cache. The same reasoning applies as before. We merely desired to benchmark and prove our theory before we were more precise with our handling of even and odd data.

The third calculation worth mentioning was to discover the deviation. This was used to measure the spread across the cache "snap shot" that was given to the content manager by the timer object. We think of it as taking it across the cache trace population to attempt to summarize the continuous cache data. We were not sure how this would react with skewed data or a number of outliers in the cache. Some preliminary data analysis showed this not to be the case, on average.

The SimmProfile object contains the results of the SimmProfileEngine() calculations. It also holds the groupId. Every Client object has a profile object for use by the Content Manager. The total calculations in this object are raw Euclidean, normalized Euclidean, average, deviation, median, myScore, mode and groupId. This object has defaults set for beginning threads which proceed first (setters and getters). It serves as a type of a holding place to hand off to other methods that require the numbers for calculation in the content manager, as well as for final group matching and group data look up.

The majority of the work is completed in the content manager; again, what we named the "brains" of this operation or simulator. The trigger point is the timer, where, as we mentioned before, used the SimmSummaryUpdateTime() object. It runs the buildProfile() method in the Content Manager when the timer goes off, taking a "snap shot" of the cache, that point, it calculates all profile fields while also calculating the score and group matching. There also exists a parameter we call accumulation that allows clients to belong to one or more groups. If this is false, the group membership is evaluated each time the group membership is calculated.

The timer can be unpredictable, as it is a thread. It runs when there is neither enough clients nor any data traces in the client’s cache. We therefore have built in some mechanisms to reject the profile build request by the timer object to "tell" it to comeback when the cache reaches a minimum size. This size is usually two. Not enough clients being up is not as important as the buffer size, as we find this in what we refer to as "warm up" in the data analysis.

The client object instantiates the buildProfile() object and the SimmProfileEngine() object every entry. Once all the calculations are made, the score is tabulated by simply adding the total, score = getRawEuclidean() + getNormalizedEuclidean() + getAverage() + getDeviation() + getMedian() + getMode(). We left this number in its raw double form.

The next method to execute was the assignGroup(), which keeps track of all the groups at the place it left off and removes it if re-evaluated, based on the accumulation parameter. The tolerances are checked at this point to determine group matching on the scores. The range is based upon the user tolerance input, and it is a plus/minus range. Attention must be directed to the raw initialized score (then move on) assigning it its own id when determining the group membership. If both match, a group id is not required. The next group id in line is assigned if one is needed. If one is assigned to a group, the other is added to its group. The group number is rescinded when there is no match, as it gets

![Fig. 2: The basic Tick counter method[2]](image-url)
pre-incremented. If no match is found and this client is in a group, it will be reevaluated if there is no accumulation (set to false).

It should be noted, whenever any client requires a request from the content manager and it has to sweep the allclients[] array, it must always exclude itself to avoid mis-reading.

Once the cache is full in the client object, we simulate a request and in the profile algorithm the getDataFromClientsInMyGroup() method is called from the content manager. It confirms the requesting client is assigned to a group, and if so, looks to find a match among the clients. It also confirms the group does not include itself and is not the default group. It will proceed through all groups or the other clients in the group depending on settings and search for the requested data and return it, if found. Otherwise, it proceeds to the server for a memory search or disk.

The group assignment table is shown in figure 3 below with exaggerated id. In reality, they are integers, for space reasons. The table is virtual, in other words, it does not truly exist as a table in the code. It is merely a concept to illustrate the way the group to client relationship architecture is constructed. Within the code design, the table is just a for loop that performs group id comparisons to find matches.

The single group assignment table figure 4 is shown below, again the id, is an exaggeration.

The overall group algorithm request is shown in the figure 5 below.

The last section of our implementation is the server or
SimmServer object. This is a singleton that simulates a real
world server. Its file assignment is kept in the SimmCon-
stants object. When the simulator starts up, its file is loaded
into its memory store using its fillMemm() method called in
the constructor. This has two searchMemm() methods; one
is over loaded to handle a client object parameter in order
to run the second tick count method. It searches its memory
for a memory hit, or else merely logs a disk hit.

5. Performance/Results

Our approach to the performance and evaluation of our
Profile algorithm developed in our distributed collaborative
cache simulation was to predict the results that were pre-
determined during our research phase. It is summed up by
using predetermined expected graphs.

Predictions were based upon our initial research of what
we would expect to find under the conditions graphed.
We researched approximately 20 papers in the literature
on caching, adaptive caching and distributed collaborative
caching to arrive at our predictions. The predictions were
not necessarily correct, merely what we would expect to
find in our distributed system.

To start, we gathered data with small hand-made data
sets. Our reasoning was to obtain a "feel" or to discover
any patterns on a smaller level, rather than large. We were
able to run any number of clients within system resource
confinements. We also used small hand made files to verify
our theory for group assignments and group profiles. We
used files with all the same traces (integers). We did every
other trace as the same, or completely different, to ensure
we would obtain the results we expected. If the results were
something other than what we expected, would would then
have to explain it. This enabled us to establish a baseline to
determine if the larger data should run.

We also tested different range traces within the same
session, as an example, 12345 and 1234567. As you will
note, the range of these numbers is large. This large range
made a difference when it came to the profile data. It would
skew the group scores and at times, if the data skew was
extreme, it did not find any matches. This caused the toler-
ance or threshold to be adjusted, accordingly. It also verified
that caching works best when all devices are processing
something similar, such as different devices viewing the
same movie.

We also studied the results of utilizing homogeneous trace
data verses data that was the same or similar.

The graph shown in figure 7, was one of our first eval-
uations involving the affect of the accumulation parameter
on the algorithms performance. With the accumulation pa-
rameter turned off (disabled), group membership reevaluates
each profile execution. As can be observed in the graph in
figure 6 the line is "jagged" and increasing. The increase,
which is the increase in hits, is caused as the number of
clients is increased, as are the number of hits. This behavior
is expected. Also, around 20 clients the affect of "warm
up" can be observed with the sharp "dip" in the line. The
"jagged" observation is a result of the group reevaluation.
This "jagged" look is expected, but we must admit, we
were a bit surprised until we examined the data in order
to discover the reason for this behavior. The reason the
"jaggedness" occurs is because as group membership comes
and goes, so does its hit ratio. In other words, as the client
enters a group, it has a certain hit count. As it is expelled
and/or joins another group (in and out of groups), its hit
ratio changes again, as seen in the jagged line. This "jagged"
phenomena can be observed in other evaluations, as well,
whenever the accumulation parameter is turned off.

The graph shown in figure 7 displays the tick counts
for the Profiler algorithm. The settings for this run are 1k
clients in the system, 1k trace data per client, with group
accumulation enabled. The results of the graph show, as
the cache size increases, the tick count decreases. Note,
that since the group accumulation is enabled, there is no
"jaggedness" to the graph trace. This result is as expected.

The graph in figure 8, evidences the miss rate for the
profile algorithm. The settings for this run are group ac-
cumulation enabled, 1k clients in the system and 1k trace
data. This run had a high threshold, as well. This high
threshold would allow "easy" or a wide range for group
membership. The results in this graph show, as the cache size increases, there was one miss! We attribute this one miss to the "warm up". This was not expected behavior. We were quite surprised that the Profiler algorithm worked so well. We were concerned that this may have hit the worst case scenario where every client is a member of the same group, nullifying the efficiency of hop count, battery power etc. We surmise this not to be the case, as the testing finished in under 15 minutes. If every client was in the same group it would have taken hours to complete from previous observations. Without further study, we cannot say this is a fact. Note the small "dip" in the graph, it looks to be a negative number, but it is not. This is the same phenomena observed with excel as before [9].

The graph shown in figure 8 shows the percentage of hits for the Profile algorithm with increasing cache size. The run settings are accumulation disabled, 1k clients in the system and a wide threshold as above. This graph shows the "jagged" phenomena. This occurs due to the clients entering and leaving groups, changing the hit ratio as the group membership changes. This behavior is as expected.

The graphs shown in figures 10 and 11 show the Profiler algorithm with accumulation disabled, 200 clients in the system and increasing cache size for miss and hit rates. Both graphs show as cache size increases the miss and hit rate decrease. This behavior is as expected due to self service from the increased cache size. In graph 10 the "jaggedness" can be slightly observed. This run only had 200 clients. As can be observed, it is less accurate than the run with 1k clients. This is a result of possessing less clients with which to collaborate.

The next two graphs shown in figures 12 and 13, show the Profile algorithm for Hit and Miss rates. The run settings are 1k clients in the system, group accumulation enabled, average threshold and increasing cache size. Observe that the graph in figure 12, has two misses! This was not expected. The performance of the Profiler algorithm exceeded our
expectations. Note that the performance improves greatly with 1k clients, as compared to 200 or smaller.

The graph in figure 12 is shown for Miss rates with increasing clients. This graph illustrates the "jaggedness" phenomena that is very exaggerated. This behavior is as expected.

It should be noted that once our performance study for the Profile algorithm reached the point of 1k clients, its performance was so outstanding that we could not accurately display the percentage numbers. We observed miss rates of just one with group accumulation and without group accumulation. We saw numbers for misses fall from 249 to 28 to 11 to 1 ultimately to 0 within the same run. This data was also averaged over a handful of runs (20). This justifies further study and could be possible publication in the future.

Figure 15 shows the tick counts for the Profile algorithm with accumulation disabled. It can be observed that as the cache size increases the tick count decreases. Once the cache is large enough, it can service itself. This behavior is as expected.

6. CONCLUSIONS

Our conclusions are based upon our graph and observation data. We have proposed our Profile algorithm. The results of our CCDistSimm, while it needs some improvements, has been quite productive. With our profiler algorithm, the more groups that exist, the more accurate the system is on hit. There is also less server access.

The memory layout as set forth by a file determines disk access. If we randomly sample client cache files to create memory, we obtain real world results. We found we can skew it, or make it so that we never get any memory hits, just disk. We did observe there is a science to the data that should be placed in server memory, as opposed to what is retained on disk. This is something that is worth future research.

With our profile algorithm, we discovered theories that some optimization could be performed by not using a timer but by calculating how many fetches are made. In other words, there could be a "fetch" profile created to determine the update of the cache profile. This could be utilized in conjunction with a timer, and/or our idea mentioned in previous sections of a "score" heap.

In future work, the number of groups should be studied. To observe that if by having the accumulation feature on, would the worst case scenario ever happen where every client
belongs in each others group. We could research to find a better mathematical way to build the profile (if needed).

The LRU or Least Recently Used replacement policy was observed to possess a threshold for cache size and internal hits that is proportional to the size of the incoming data. One average was around 1/2 the data file size. We do understand that in reality, this incoming data is not a file and is to be considered infinite. We feel this observation is worth noting.

Our Profile algorithm shows a lot of promise. The results far exceeded our expectations for speed and accuracy. One such expectation was that of minimizing the cost to server memory and disk hits. Our Profile algorithms main ideas should be studied and improved in future work. These results show the Profile algorithm warrants a test bed and/or real world scenario testing and analysis.

References

[27] http://www.codeproject.com/A rticles/29224/Six-Sigma-Interview-Questions
[29] Algorithm Design, John Kleinberg and Eva Tardos