2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Replicate and Bundle (RnB) – A Mechanism for Relieving Bottlenecks in Data Centers

Shachar Raindel and Yitzhak Birk
Electrical Engineering Dept.
Technion
Haifa 32000, Israel
raindel@tx.technion.ac.il, birk@ee.technion.ac.il

***Abstract*—This work addresses the scalability and efficiency of RAM-based storage systems wherein multiple objects must be retrieved per user request. Here, much of the CPU work is per server transaction, not per requested item. Adding servers and spreading the data across them also spreads any given set of requested items across more servers, thereby increasing the total number of server transactions per user request. The resulting poor scalability, dubbed the Multi-get Hole, has been reported in Web 2.0 systems using memcached – a popular memory-based key-value storage system. We present Replicate and Bundle (RnB), a somewhat unintuitive approach: rather than add CPUs, we add memory. Object replicas are mapped "randomly" to servers, and requested objects are bundled, selecting replicas so as to minimize the number of servers accessed per user request and thus the total CPU work per request. We studied RnB via simulation in the context of DRAM-based storage, utilizing micro benchmarks and implemented RnB modules for calibration. Our results show that RnB substantially reduces the number of transactions per request, making operation more efficient. Also, unlike most alternatives, RnB permits flexible growth and relatively easy deployment. Finally, in systems wherein data is replicated for other reasons, RnB is nearly free.**

I. Introduction

A. Background

In this work, we consider the scalability and efficiency of RAM-based read-mostly [6] storage caching systems in Web 2.0 data centers (e.g., Facebook, Twitter, Gmail). In such data centers (Fig. 1) , a large number of web servers, located behind a load balancer and nearly stateless, run the web application code. This facilitates scaling of the web server layer. An authoritative copy of the (read-mostly) data for the application is stored in a large, disk based database (DB), such as MySQL, MS-SQL, Oracle, Cassandra, etc. However, DB access is slow, so a special caching layer is employed. Memcached is a RAM based key-value storage/caching service, with a simple network access protocol. Several memcached servers are used to cache recent DB queries and their results, often simply all the data or nearly so. These servers are not stateless, but data loss in them is usually tolerable. Instead, they are optimized for performance. In view of the above, we regard the memcached servers as a RAM based storage with relaxed reliability requirements, not as a cache.

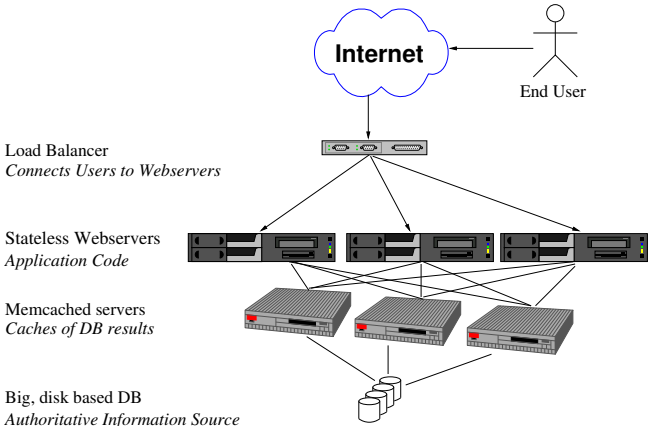

Figure 1. A typical web application stack deployment.

For performance and scalability reasons, the identity of a server storing a copy of a requested item must be determined (usually) without communication. Therefore, memcached employs consistent hashing [1] to map items to servers in a very uniform, pseudo-random manner. As a result, in an N-server system, a client request for M specific items will require sending requests to $N\left(1-\left(1-\frac{1}{N}\right)^{M}\right)$ servers on average. The calculation is detailed in Section II-A. Note that when $N >> M$ ($M >> N$), this number is approximately M (N). When a client request requires fetching a much larger number of items than the total number of servers, every server is likely to be accessed, so adding servers commensurately increases the number of transactions per user request. If a substantial amount of server CPU work is per transaction, not per item, this offers no relief to the server CPUs. This phenomenon has been dubbed the Multi-Get Hole [2].

B. Terminology

An end user sends a request for a set of data items ("items"), the request set, to the web service. The request size for our analysis is the number of items in the request

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.65

set. The user request reaches the web servers, which we refer to as clients. The client translates the request into a number of transactions. Each transaction, containing a list of items, is sent to a different Memcached server ("server"). (Front-end web servers are clients of the Memcached servers.) If an item is not found on the server, there is a miss, and the client will fetch this item from the DB, possibly also writing it back into the relevant server.

Finally, we define several metrics used in this work:

- Transactions Per Request (TPR) – the mean number of transactions needed to satisfy a single user request.
- Transactions Per Request Per Server (TPRPS) – TPR divided by the number of servers.
- Maximum System Throughput ("Throughput") – the maximum request-handling rate of the entire system.
- TPRPS Scaling Factor – the ratio of TPRPS between two systems.
- Throughput Scaling Factor – the throughput ratio between two systems.

For reader convenience, we provide here definitions of terms that are used in a later part of this work:

- Overbooking – providing less physical memory than implied by the declared number of replicas.
- Hitchhikers – piggybacking redundant item-requests onto necessary ones.

C. Our Contribution

We present "Replicate and Bundle" (RnB), a method for reducing the number of transactions required to process an end user request. This method enables increasing the maximum system throughput without adding CPUs. RnB entails 1) data replication and 2) bundling of items requested from the same server into a single transaction. We use a pseudo-random object-to-server mapping for each object's different replicas, placing the replicas on different servers for each object. During data fetch, we choose which replica to access in order to reduce the number of servers that need to be accessed for any given request. Finding a minimal set of servers is the well known minimum set cover problem, which is NP-complete [3]. Therefore, we use heuristic low-complexity approaches. Considerable benefits are obtained even with sub-optimal server selection.

RnB is a stateless, distributed algorithm. It does not require any additional communication, and requires almost exactly the same amount of configuration information as consistent hashing. Therefore, it does not cause an increase in the storage system latency for reads, and is relatively easy to deploy and configure. While our results are in the context of online social network data sets, RnB can be beneficially applied to other similar workloads.

RnB achieves considerable additional gain when the end user request allows for partial results. For example, in some use cases it is acceptable to bring only 90% of the requested records, perhaps also with some probability parameter.

We have also developed two mechanisms that are likely to find use beyond RnB. The first is several approaches for handling two service classes in LRU based caching systems. The second is an extension of consistent hashing, which we call Ranged Consistent Hashing (RCH). This extension allows selecting, for each item stored, a group of servers that will host it. The approach preserves the good attributes of consistent hashing, while achieving a balanced and uniform distribution of the replicas.

In this paper, we present RnB along with an insight-providing simulation study. We also describe elements of a proof-of-concept implementation. In Section II, we analyze the multi-get hole, and in Section III, we present RnB. In Section IV, we highlight some implementation issues, and Section V provides discussion and concluding remarks. Related work is discussed in Sections II and V.

II. THE MULTI-GET HOLE

A. Analytical Quantification for Random Data

Consider a set of N servers and a request for M items,and recall that a single transaction suffices for fetching any number of items from a given server. For a setting with no replication, and with items that have been placed in servers randomly, the TPRPS can be derived as follows. If we regard the servers as urns and the items as balls, the probability that we have a transaction with a given server is the probability that the corresponding urn will not be empty when throwing M balls into N urns. This probability is well known [4]: $W(N,M) = 1-\left(1-\frac{1}{N}\right)^M$. The expected number of servers that need to be accessed (the TPR) is therefore $N \cdot W(N,M)$, so the TPRPS is $\frac{TPR}{N} = W(N,M)$. We are trying to estimate the relative throughput increase achieved by adding servers. Therefore, the relevant metric is the relative change in the TPRPS – the TPRPS scaling factor – and not the absolute value change in the TPRPS. The TPRPS scaling factor when doubling the number of servers is

$$\frac{W(N,M)}{W(2N,M)} = \frac{1-\left(1-\frac{1}{N}\right)^M}{1-\left(1-\frac{1}{2N}\right)^M} \ .$$

Ideal scaling is achieved if there is only one item: $\frac{W(N,1)}{W(2N,1)} = \frac{1}{N}/\frac{1}{2N} = 2$, as expected – doubling the number of servers would double the throughput. However, for multi-item requests, scaling is much worse. Fig. 2 depicts the TPRPS scaling factor achieved when doubling the number of servers versus the number of servers in the initial system, assuming that the system is only sensitive to the number of transactions. Plots are provided for various numbers of items in a request. Note that when the number of servers is significantly smaller than the number of items in a request, doubling the number of servers yields negligible performance benefit. Even when the two numbers are equal, doubling the number of servers only increases throughput by

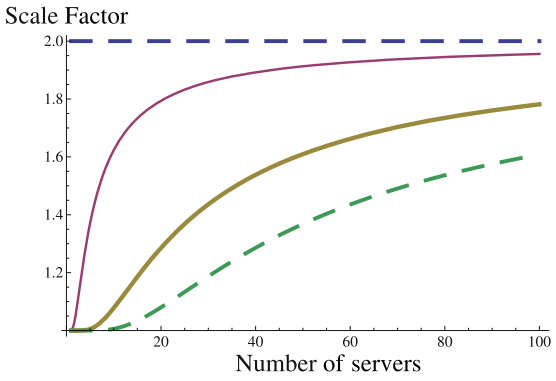

Figure 2. The TPRPS scaling factor when doubling the number of servers vs. the initial number of servers, for requests containing 1 (blue), 10 (purple), 50 (yellow) and 100 (green) items (larger is better).

some 50%. Therefore, whenever the system's bottleneck is the number of transactions and the data items are distributed randomly, the intuitive action of adding servers will achieve extremely poor scaling results.

B. Simulation Study of the Multi-Get Hole

We used a specially built simulator to study the multi-get hole, as it is manifested in memcached systems. For the simulation, we used publicly available social network graph datasets. We ran the simulation with an increasing number of servers and counted the average number of transactions needed to satisfy an end user request. The results are depicted in Fig. 3.(See Section III-B for further simulator and simulation details.)

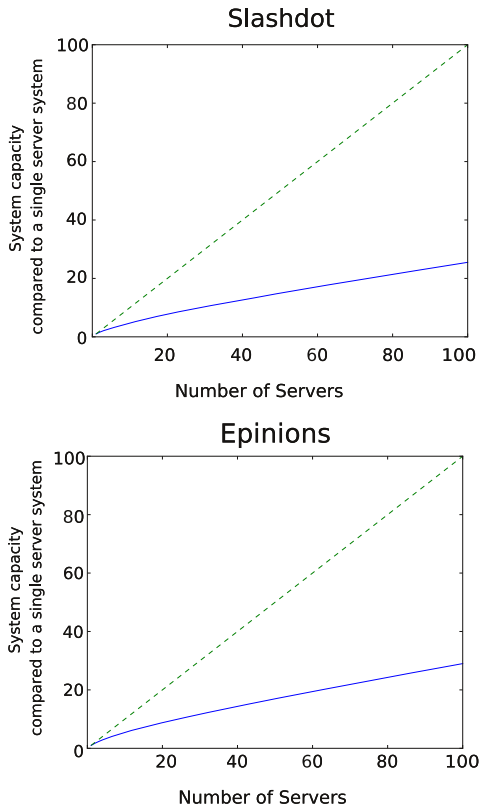

Figure 3. Quantifying the multi-get hole. The solid blue line shows the throughput with a varying number of servers, relative to the throughput of a single server system. The dashed green line is the (theoretical) maximum scaling.

C. Known Approaches to Addressing the Multi-Get Hole

In [5], Pujol et al. attempt to detect groups that demonstrate strong affinity, and store them together on a server. A central directory server is assumed, which is queried whenever there is an attempt to access the user data. This increase in the number of required communication rounds is significant in this case, since the load and delay caused by an additional communication round is larger than the decrease in the load achieved by improving locality. While this solution is useful in a shared-nothing architecture where the data storage nodes are also responsible for the processing, it is not practical for deployments in which there is a strong separation between data nodes and processing nodes.

Various approaches have been considered by the industry in attempt to work around the multi-get hole:

1) In situations wherein each item is already fetched from a different server such that no further fragmentation of transactions is possible, it is common practice to increase the number of servers, utilizing the hash to distribute the load evenly.
2) Many memcached client implementations, such as libmemcached, permits the use of a "master key" for each group of items fetched, to force the fetching all of them from the same server [6].
3) Facebook has reported [2] replication of every memcached in its entirety – both hardware and data, with the clients randomly picking one of the server replicas for each transaction.
4) Facebook engineers also suggest [2] mixing CPU–intensive and network bandwidth–intensive workloads in the memcached server.

The first of these solutions is only reasonably gainful in situations wherein each request requires a very small number of items and in the case of a very large number of servers. Even then, while adding servers will increase processing capacity, the servers are used inefficiently because of the large total amount of work per request. This is wasteful in terms of both capital cost and energy.

While the second solution requires no additional hardware, and theoretically allows perfect scaling, devising such a "master key" system is usually extremely difficult, if not impossible, without replication [5]. Such a system is effective, even with replication, only when there is a clear and known affinity among the data items, ideally forming

cliques or strongly connected components of a reasonable size.

To the best of our knowledge, for the reasons listed above, only the first, the third and the fourth alternative solutions are used by industry for the scenarios analyzed in this work. The first solution is used when each end user request requires a very small number of items, and the effect of the multi-get hole is minor. In the third solution, one gets exactly what one pays for – k replicas of the system yield a k-fold increase in the throughput, but no more. Additionally, the third solution only permits system enlargement in relatively large strides. The fourth solution, while leading to better utilization of the entire storage system, does not actually solve the multi-get hole. It can cause a bigger, harder scaling problem later on, when the mix of available load becomes unbalanced or when the CPU workload significantly limits the achieved network utilization. In view of the above, we will use the third solution – complete system replication – as the baseline for comparison with our approach.

III. Replicate and Bundle (RnB)

A. The Basic RnB Solution

Replication. Each item is written to a preconfigured set of servers, chosen using consistent hashing.

Bundling. The locations of all of the replicas of the items in the request set are calculated, and a group of servers that jointly possess all requested items is computed. The problem of finding the minimum group is NP-complete [7]. However, we show through simulations that a linear time approximation achieves extremely good results in the context of RnB. Clearly, the mean rather than the worst case is the relevant measure.

During most of the evaluation in this section, we assume that the system handles each end user request individually and bundles only items in the same request. In Section III-E, we discuss combining requests.

B. Memcached System Simulator

The simulator was written from scratch and was targeted specifically at the performance of distributed key-value storage systems. We used micro-benchmarks on a single memcached instance to calibrate the simulation. Since our emphasis is on the multi-get hole, we focused on the total amount of server work per request, expressed as the number of transactions per request. Therefore, queuing is not relevant and requests were simulated individually.

Given the interest only in the number of transactions, we assumed that all data items are of the same size. Also, we assume that all objects are found in memory. The latter assumption will be modified as we introduce extensions to RnB (Section III-C).

Since we were unable to obtain real-life traces of accesses to memcached in big deployments, we utilize, for most of our experiments, graphs of social networks to generate

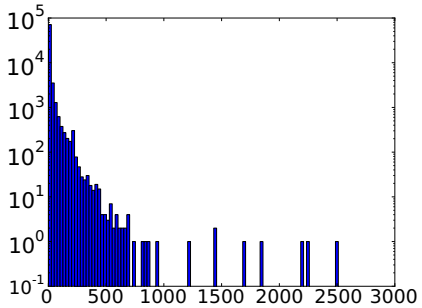

Figure 4. The node degree histogram for the Slashdot network.

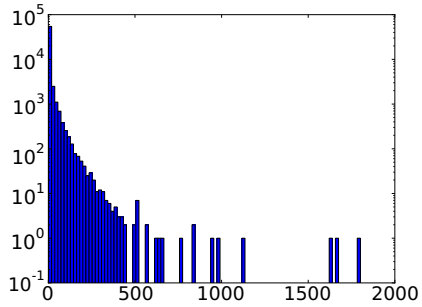

Figure 5. The node degree histogram for the Epinions network.

the access pattern to the memcached. This approach is similar to the approach used in [8] for similar simulations. A thorough discussion of the assumptions and inaccuracies in our simulator is available in [7].

We used two different social networks for generating our request patterns: the Slashdot network (from [9]) and the Epinions network (from [10]). . The Slashdot network is a directed graph containing 82,168 nodes (users) and 948,464 edges, resulting in an average node degree of 11.54. Fig. 4 depicts the histogram of node degrees in the Slashdot network. The Epinions network is a directed graph containing 75879 nodes and 508837 edges, resulting in an average node degree of 6.7. Fig. 5 depicts the histogram of node degrees in the Epinions network.

We assumed that each user in the social network was represented by a single item – the user's "status." The generation of end user requests based on the social network graph is done in two steps. First, we randomly and uniformly picked a user out of all of the users in the graph. Next, we looked at the user's friends. We assumed that to satisfy the end user request, we needed to fetch the items representing all of the user's friends

The goal of the micro-benchmarks is to provide us with real-life values for the performance of memcached-based systems. For the benchmarks we used the freely available memaslap utility, which is included as part of libmemcached.

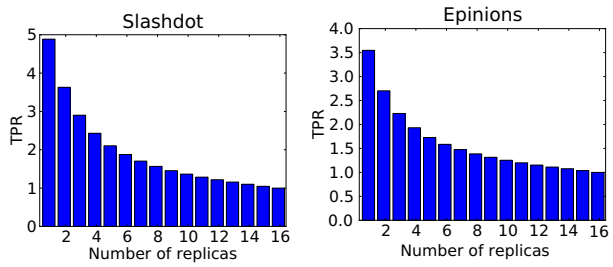

Figure 6. The average TPR when using RnB vs. the number of replicas, for a 16-server system

The utility allows us to test how many transactions per second the memcached server is able to process, using a synthetic workload with adjustable parameters. See Appendix A for further details.

The micro-benchmarks provided us with an estimate of the number of queries per second that the memcached can handle vs. the number of items in the query. We used these values to calibrate the results from our simulator, so the results correlate to actual throughput figures. The simulator produced a histogram of the number of items in each transaction and, based on this histogram, we estimated the maximum throughput of the system.

We next set out to quantify the benefit gained from replication. We modified the simulator to use RnB, replicating the data items using multiple hash functions. For each request, the application layer calculates the location of the replicas of each item, and uses a greedy minimum set cover approximation to decide what replicas to fetch. In order to estimate the gains, we used a test setting of a 16-server system. Fig. 6 depicts the average number of transactions (where each transaction might bundle multiple items) needed to fulfill an end user request as a function of the number of replicas (one replica is the baseline, where each item is stored in one server) used in the simulator.

The results demonstrate a significant reduction in TPR, reducing the number of transactions, in some cases, by more than 50% utilizing a total of 4 copies for each item. Having shown an impressive reduction in TPR, the question of memory cost arises. In some cases, replication already exists for reliability and fault tolerance reasons or as part of a full-system replication. In such cases, using RnB brings a "free" benefit when it comes to memory cost. Deploying our suggested implementation of RnB would involve modifying the location of the replicas, but would not harm the reliability of the system or cause an increase in the amount of memory required. In all cases, one may want to add memory for RnB. We next present enhancements of RnB aimed at improving its utilization of the memory.

C. Enhancements to RnB

We now describe several possible enhancements to the basic RnB approach. Due to difficulty in attributing the improvements to specific enhancements, we show only the results when using all of the enhancements together. As a baseline for comparison, we use the basic RnB scheme with a naïve storage allocation approach, whereby the amount of physical memory is exactly the replication level times the number of items stored.

1) Overbooking with a distinguished copy: This enhancement is aimed at exploiting the fact that different objects are accessed less frequently, and that the degree of different graph nodes is different (i.e. clusters of affinity [11]). The challenge, of course, is to exploit this while retaining simplicity.

Our mechanism combines a feature in the memcached servers and a property of the replica selection algorithm with tuning of the system configuration. Each of the memcached servers keeps a local LRU list of the items stored on the server, and drops unused items when running out of space. The result is that both the number of physical replicas of each object and their locations within the relevant set of servers are determined implicitly, adaptively, and in a fully distributed manner. To ensure that each data item still has at least one copy in memory, we mark one replica of each data item as its distinguished copy. This can be done easily, by selecting, *a priori*, one of the hash functions as the "distinguished" hash function.

The greedy set cover algorithm we use for selecting the servers to satisfy a request has a nice property – if two requests contain similar item sets, the replicas used for most of the items will probably be the same for both requests. This is illustrated in Fig. 7. This property allows us to "automatically" benefit from the spatial locality in the requests, making some of the replicas for each item extremely "cold." The local LRUs on the memcached servers will drop these cold replicas, making more effective use of any given amount of memory. There may be additional phenomena that render Overbooking beneficial, calling for further study.

Overbooking can be tuned by selecting the number of declared ("logical") replicas. Lastly, whenever an item is not bundled, we access its distinguished copy in order not to pollute other server caches with its copies.

Overbooking allows us to achieve a much better trade-off between memory and TPR. In our experimental setup, as detailed in Section IV, a two-fold increase in memory, along with a larger number of logical replicas, achieved nearly a two-fold decrease in TPR.

It is important to mention that when the client is handling a request, it is practically oblivious to the overbooking.

2) Improving Hit Probability via Hitchhiking: This entails adding extra (requested) items to existing transactions. Doing so does not increase the number of transactions. In

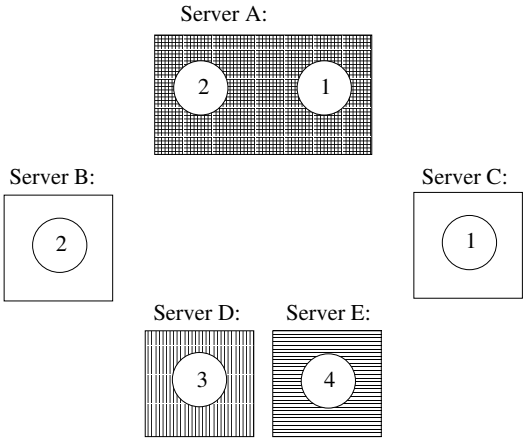

Figure 7. An example of request locality reducing the needed memory. Consider two requests: I) items 1, 2, 3; II) items 1, 2, 4. The figure depicts a possible item placement. Notice that both requests will fetch items 1 and 2 from the same server, A, even though a virtual copy of item 1 exists on server C, and a virtual copy of item 2 exists on server B. Since there is no access to the replicas on servers B and C, the servers will eventually discard the replicas through their LRU mechanism.

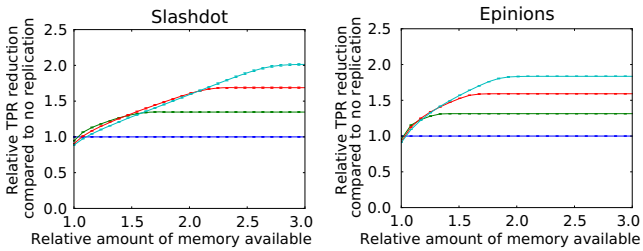

Figure 8. The reduction in TPR with replication relative to that without replication vs. the relative amount of memory available. 1.0 on the horizontal axis is exactly enough memory to store one copy of the data. We show lines for systems with replication levels of 1 (blue), 2 (green), 3 (red) and 4 (light blue). Note the overbooking used in these simulations. The results are for a 16-server system.

conjunction with overbooking, it reduces that probability of a miss and a resulting additional transaction to fetch the distinguished copy, but increases total traffic. It is mostly beneficial when per-transaction processing is the bottleneck. Further details of this policy, such as whether a server's LRU should be updated based on a hitchhiker, are topics for further research. For the results we present in this work, we enabled hitchhiking and updated the LRU only upon a hit in the hitchhiking request. In case of a miss, we write the missing item only to the replica that was the first to be picked by the greedy set cover algorithm and possibly to the distinguished copy as well.

D. Evaluating RnB with Limited Memory Size

We now estimate the benefit from replication in a scenario wherein the memcached servers have a limited amount of memory, with the aforementioned enhancements. Here, we must take into account cache misses. Since the exact cost of a miss is very hard to estimate and changes widely between deployments, we chose to tackle the issue of misses by ensuring that the distinguished copies of the items will never suffer a miss. Doing so allows us to modify the amount of space available for replicas with a well-known and agreeable penalty when accessing a replica that was evacuated – it now translates only into additional transactions. We assumed that we first access the data according to the original fetching plan, and performed a second round of access to fetch the items that were not found, if we did not yet fetch their distinguished copy. Since for single item transactions we always fetch the distinguished copy, we will not perform a second transaction on these items. This allows bundling when fetching distinguished copies after the misses, so the penalty is not exactly a transaction per miss. Note that since we allocate for the distinguished copies the same amount of memory that the original system had, we are guaranteed not to increase the global system miss rate. However, excessive overbooking can increase TPR!

The TPR reduction compared to the TPR of the same request pattern with no data-item replication is presented in Fig. 8. Note that those graphs depict the TPR with all of the aforementioned enhancements. As one can see in the graph, the enhancements allow us to achieve a significant reduction in TPR, with much lower memory requirements than the trivial replication shown in Fig. 6 – instead of requiring 4 times more memory to achieve 50% reduction in the TPR, increasing the available memory by a factor of 2.5 achieves the same reduction in the TPR. If the system already keeps another copy of each item for disaster recovery reasons, we can get 25% reduction in the TPR for free.

E. Merging Multiple Requests

Several ([12], [13]) real world implementations of memcached support merging multiple end user requests, thereby reducing the number of transactions performed with the servers. With RnB, however, these implementations run the risk of affecting request locality.

We modified the full simulator to collect a predefined number of requests and handle them as a single request. In Fig. 9. we show the reduction in TPR when every two consecutive requests are merged, compared to the same request pattern with no replication. Since we normalize to the no data items replication baseline, this graph is directly comparable to Fig. 8 regarding the benefit from adding replicas. While the gain from adding replicas at any given memory level is lower in such a setting, the combination of the techniques is beneficial, as can be seen in Fig. 10. . Since we combine requests, the TPRPS for the no-replication baseline is also much lower than in Section III-D, resulting in a lower TPRPS for all of the replication levels when merging requests.

Whereas there is an apparent intrinsic affinity among

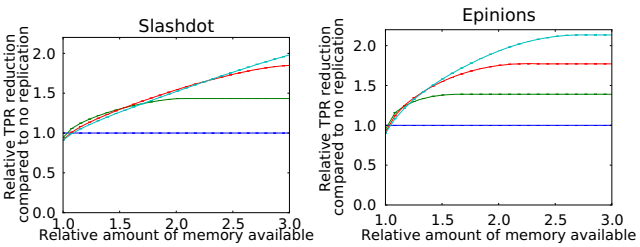

Figure 9. The relative reduction in TPR achieved by using RnB, compared to no replication vs. the amount of memory available, when merging 2 requests, for systems with replication levels of 1 (blue), 2 (green), 3 (red) and 4 (light blue). The results are for a 16-server system.

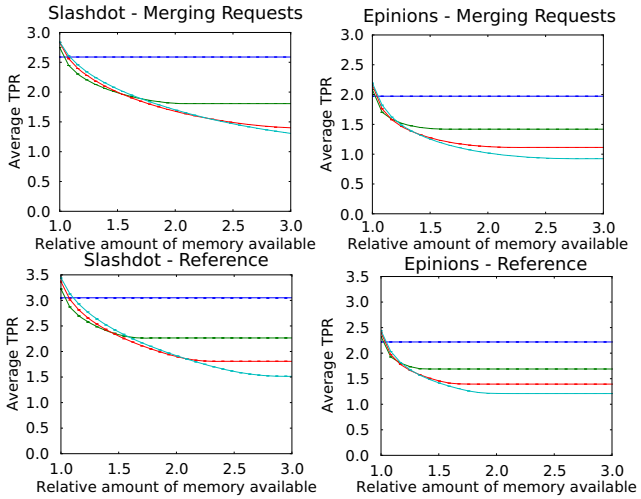

Figure 10. The TPR vs. the amount of memory available, when merging 2 requests (top) and when handling a single request at a time (bottom), for systems with "logical" replication levels of 1 (blue), 2 (green), 3 (red) and 4 (light blue). The results are for a 16-server system.

same-request items, there is none whatsoever among items requested in unrelated requests that happen to take place in close time proximity. Therefore, treating two such requests as a single request may actually hurt the "self organization" property that stems from the affinity: the server to which two different-request items are sent in a transaction of the merged request may be different from the one(s) to which they are sent in separate single-request transactions, thereby increasing the memory footprint.

F. Handling "LIMIT" style Requests with RnB

In social web applications, the end user request is often of the form "fetch me at least X items out of the following list" or "fetch as many items as possible out of the following list within X milliseconds." Such limited queries also exist in traditional SQL databases, as the LIMIT n clause for SELECT statements. Unless the query specifically asks for an ordering, the rows returned in the result can be any set of n different rows out of the table. Similarly, cloud-based services such as Google Base Data API, do not guarantee to

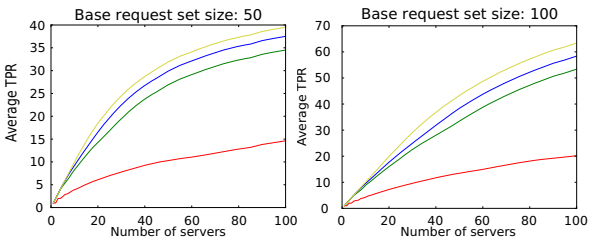

Figure 11. selected so as to minimize the number of transactions; no replication. Fetched fraction: 95% (blue), 90% (green) and 50% (red) and 100% (yellow, full set). Plots are presented for two different request set sizes.

find any, most, or all of the possible results. In this paper, we present results only for requests of the first form – "fetch me at least X items out of the following list." In [7], we also show results for requests of the latter form.

A simple but ineffective implementation would entail picking, at random, X items from the list, and fetching only them. RnB can do much better. The added flexibility of "LIMIT" requests allows RnB to avoid fetching items which would require an additional transaction per item or very few items.

The problem of selecting the minimal group of servers that jointly possess a sufficiently large subset of the request set remains NP-hard. The problem of selecting the servers when handling a request that does not have a "LIMIT" clause can be reduced to this problem by setting the limit to the number of different items in the system. However, it is relatively easy to convert the greedy algorithm to pick partial results by ceasing to pick servers after enough items are covered. While we do not show any bounds on the approximation, our simulation results show that this approximation, for random inputs, gives very considerable performance gains.

We evaluated the benefit of RnB for requests with a LIMIT clause. We used a simplified simulator to estimate the gain from RnB in this scenario. The simplified simulator performed Monte Carlo style simulation. It assumed that the servers have enough memory to completely avoid misses, and that the set of items in each request is random and independent of the previous request. The details of the simplified simulator are available in [7].

Assuming that the implementation selects the items to maximize the bundling, even without replication there is a significant reduction in the number of transactions required to satisfy a request relative to the case of fetching all items. This can be seen in Fig. 11. However, when replication is also used, the gain is, in most cases, much bigger, as can be seen in Fig. 12. With five replicas, which requires at most five times more memory, we can reduce the number of transactions to merely 30% of that required with a single replica, whereby each item is stored exactly once. Even with only two replicas, we can reduce the number of transactions down to around 65% of the TPR without RnB.

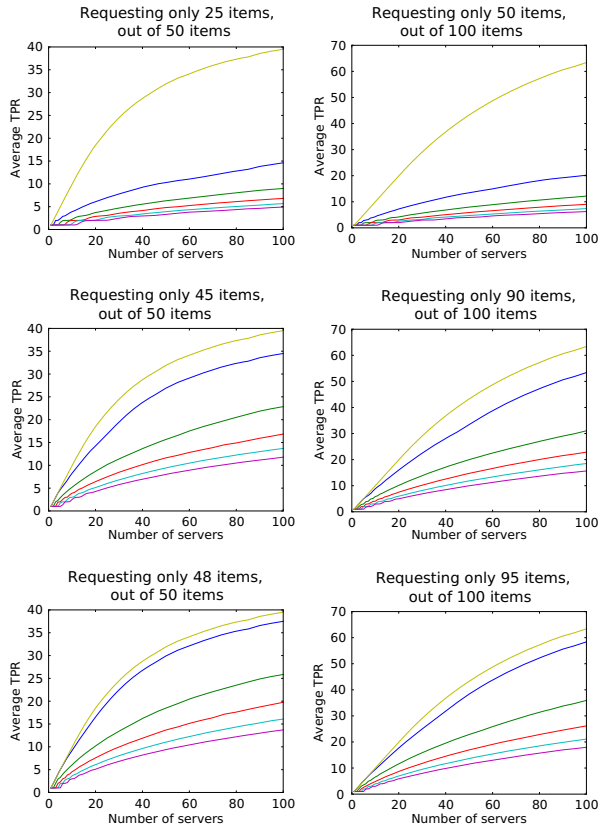

Figure 12. The average TPR for fetching a subset of the request set vs. the number of servers, with replication levels of 2 (green), 3 (red), 4 (light blue) and 5 (purple) replicas, all without overbooking. For reference, we added lines for no replication, with (blue) and without (yellow) the LIMIT clause. Graphs are presented for two different request set sizes, with three different subset sizes: 50%, 90%, and 95%.

We leave the exact estimation of the memory required for replication when handling these kinds of requests to future work.

G. When RnB is not Effective

There are cases wherein RnB will not have any effect, and it could even hurt performance. Examples:

- *The activity is not read mostly*: During write access, RnB requires updating multiple replicas. However, when replication is required for reasons such as reliability, RnB does not further increase the write complexity.
- *Data items are read individually (single-item requests), without any grouping of the requested items*: In such cases, basic RnB would do nothing, but cross-request bundling can still help.
- *Consistency is critical*: RnB, like any memcached deployment, does not provide strong consistency guarantees. It is, nonetheless, not worse.
- *Very large request sets.* Here, per-item work, not per-transaction work, is likely to be the performance bottleneck, so RnB would not help much.

IV. RnB Implementation

We have defined and partially implemented the main elements required for implementing RnB in a memcached setting. We next highlight some of the components that we defined and implemented:

Heuristic for minimum set cover. We developed an implementation based on bit-sets, which finds a cover solution using a relatively small number of CPU cycles.

Ranged consistent hashing. This scheme is an extension to consistent hashing, which improves the runtime efficiency of finding a set of distinct servers for the replicas of a given item while retaining the good properties of consistent hashing. It entails traveling along the consistent hashing continuum, gathering servers until there are enough unique ones.

Consistency and support for atomic operations. We have shown that it is possible to use RnB with consistency guarantees that are no worse than memcached. Additionally, we proposed schemes for atomic operations in an RnB enabled memcached system. For example, remove all but the distinguished copies of an item before modifying it, then let RnB-memcached create the new copies on demand, after the atomic operation completes.

V. Discussion and Conclusions

A. Additional Related Research

FAWN [14] is a distributed key–value storage system with a memcached interface, aimed at power efficiency. In [14] it is compared with disk based systems. It makes no use of redundancy.

CRAQ [15] uses redundant copies of the data to allow better read performance, but only for single-item requests.

Concepts of replication and bundling, similar to the one RnB is based upon, have been previously studied in storage systems with the goal of improving system performance ([8]). However, the focus in [8] is on data arrangement within a disk to reduce seek work.

In the context of RAM based storage, Ongaro et al. [16] consider replication, but the focus of their work is on fault recovery. As such, it assumes that only one replica is memory resident, with the secondary replicas written to mechanical disks.

In [17], Mike Mitzenmacher proposed the use of a choice between two options for load balancing. While the utilization of choice is common between this work and [17], Mitzenmacher's work focused on achieving a better load balancing, while this work focuses on achieving a better bundling, which reduces the total amount of work that the system performs.

B. Conclusions

RnB combines object replication and requested-item bundling in order to reduce the amount of work (transactions) required by the back-end memory servers for handling a user request.

While each of the techniques has been employed in other contexts before, it is their combination that permits flexibility in the bundling, which is the key to the contribution.

In addition to the basic RnB scheme, various enhancements such as declaring a larger number of replicas than can actually be stored in memory, have been proposed and evaluated. Both analytical techniques (for random data) and simulation (for "typical" data) suggest a very substantial reduction in the number of required server transactions per multi-item user request.

We have also implemented the core of such a system, and in so doing developed efficient techniques such as ranged consistent hashing.

RnB does create some extra work for the front-end servers. However, these do not hold data and can be scaled very easily.

Finally, object replication is often done anyhow. In such settings, the main cost element of RnB comes almost for free. (One may still want to add some memory and declare an even larger number of replicas.) RnB also supports smooth scalability and is relatively easy to incorporate in existing systems.

Our simulation study was carried out for a relatively small number of servers. Given the promising findings, one topic for further study is the scalability of RnB, both in terms of the quality and overhead of the bundling algorithms and in terms of the degree of improvement. Studies simulating or implementing RnB on tens of thousands of servers are called for.

Additional topics for further research include improved tuning and evaluation for real, large-scale systems. A specifically interesting case to consider and evaluate in future work is when the dataset is so big such that a large number of servers is required for a single replica, and adding memory requires adding additional servers as well. Additional future work includes measuring the impact of RnB on the latency and throughput metrics of real and simulated systems.

RnB might also assist in mitigating the TCP incast problem[18].

REFERENCES

[1] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *ACM Symposium on Theory of Computing (STOC)*, pp. 654–663, 1997.

[2] J. Rothschild, "High Performance at Massive Scale – Lessons learned at Facebook." http://cns.ucsd.edu/lecturearchive09.shtml#Roth, October 2009.

[3] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[4] N. Johnson, *Urn models and their application : an approach to modern discrete probability theory*. New York: Wiley, 1977.

[5] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," in *ACM SIGCOMM*, SIGCOMM '10, (New York, NY, USA), pp. 375–386, ACM, 2010.

[6] "Memcached Overview Page." http://code.google.com/p/memcached/wiki/NewOverview, Feb. 2013.

[7] S. Raindel, "Replicate and Bundle (RnB) - A Mechanism for Relieving Bottlenecks in Data Centers," M.Sc. thesis.

[8] I. Hoque and I. Gupta, "Social Network-Aware Disk Management," tech. rep., University of Illinois at Urbana-Champaign, Dec. 2010.

[9] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed Networks in Social Media," in *Conference on Human Factors in Computing Systems (CHI)*, 2010.

[10] M. Richardson, R. Agrawal, and P. Domingos, "Trust Management for the Semantic Web," in *International Semantic Web Conference (ISWC)*, pp. 351–368, 2003.

[11] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the Facebook social graph," Nov. 2011.

[12] "Feature List of Moxi." http://code.google.com/p/moxi/, Feb. 2013.

[13] "Spymemcached Optimizations Description." http://code.google.com/p/spymemcached/wiki/Optimizations, Feb. 2013.

[14] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *ACM Symposium on Operating Sys. Principles (SOSP)*, (Big Sky, MT), Oct. 2009.

[15] J. Terrace and M. J. Freedman, "Object storage on CRAQ: high-throughput chain replication for read-mostly workloads," in *USENIX*, pp. 11–11, USENIX, 2009.

[16] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *ACM Symposium on Operating Systems Principles (SOSP)* (T. Wobber and P. Druschel, eds.), pp. 29–41, ACM, 2011.

[17] M. Mitzenmacher, "The power of two choices in randomized load balancing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, pp. 1094 –1104, Oct. 2001.

[18] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Workshop on Research in Enterprise Networks (WREN)*, pp. 73–82, 2009.

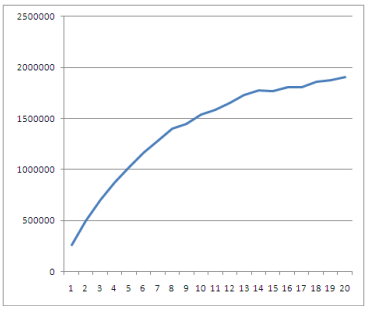

Figure 13. The average number of items fetched from the Memcached server per second vs. the number of items in a transaction.

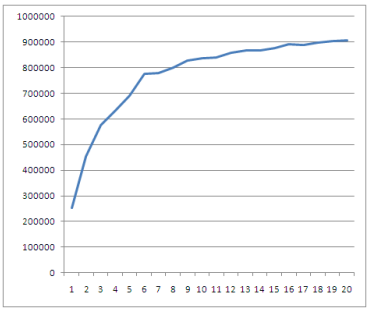

Figure 14. The average number of items fetched from the Memcached server per second vs. the number of items in a transaction, using two client computers.

APPENDIX

Micro-benchmarks of Memcached

The simulation provided us with the number of items in each server transaction the simulated system. However, for these numbers to represent the actual capacity of a real system, we had to calibrate them. For that, we performed micro-benchmarks. These benchmarks provided us with the actual performance values of a single memcached server for varying transaction sizes. We used these values to calibrate the simulation results, so as to provide an estimate of the actual throughput values.

We performed the micro-benchmarks using two PCs, with a Core i7-930 CPU, clocked at 2.8GHz and a 1Gb/s Ethernet network interface by Realtek (RTL8168B). To eliminate any external network effect, the computers were connected through a direct, relatively short LAN cable. The computers were running Ubuntu Linux with kernel versions above 2.6.32 and were set to use jumbo packets of 8KB. We used the most recent release of *memaslap* ("1.0", from bzr revno 951) , which is distributed with *libmemcached*. We set it to use extremely small items, 10 bytes each, and varied the number of items in a transaction. In addition to the get transactions, we set the benchmark program to perform set transactions, with one set transaction of a single item for every 1000 items fetched by get transactions. We focused on the maximum throughput that the memcached server can achieve, and consequently, we configured the system to perform as many transactions per second as possible. We opted to use TCP and not UDP. We made this choice since the benchmark program suffered, as expected, from considerable packet loss issues when attempting to communicate with the server as fast as possible over a protocol without flow control.

The results from this benchmark are shown in Fig. 13. Here we see that until there is a relatively large number of items in a transaction, the number of items fetched per second is linear in the number of items in a transaction, which means that the throughput is indeed bounded by the number of transactions it processes per second and not by the number of items fetched.

The situation is different when the size of the objects is bigger. In such cases, the network bandwidth becomes the bottleneck, and increasing the number of items in a transaction does not improve the performance. However, since we assume that the requests are for a large number of small items, this is not the situation when analyzing the multi-get hole.

Finally, to make sure that the bottleneck is not in our benchmark mechanism, we connected the benchmarked memcached servers through a dedicated Gb/s Ethernet switch to two "client" computers. We simultaneously ran *memaslap* on both of these clients, using command batching over SSH to make sure that they launch within microseconds of each other. We summed up the number of transactions that each of the benchmarking clients counted, to obtain the total number of items that were fetched from the server on average in each second. The results of this benchmark are shown in Fig. 14. As can be seen in the graph, the two-client configuration actually achieves a significantly lower performance. We suspect that the cause for this is the benchmark software, which might not be handling network congestion gracefully enough. However, it can be clearly seen that even when two clients are used, the server is able to supply many more data elements when a larger number of items is fetched in each transaction.