A Fault-Tolerant Cache Service for Web Search Engines: RADIC Evaluation

Carlos Gómez-Pantoja[1,2], Dolores Rexachs[5], Mauricio Marin[3,4], and Emilio Luque[5]

[1] Universidad Andres Bello, Santiago, Chile
[2] DCC, University of Chile, Santiago, Chile
[3] University of Santiago of Chile
[4] Yahoo! Research Latin America, Santiago, Chile
[5] University Autonoma of Barcelona, Barcelona, Spain

Abstract. Large Web search engines are constructed as a collection of services that are deployed on dedicated clusters of distributed-memory processors. In particular, efficient query throughput heavily relies on using result cache services devoted to maintaining the answers to most frequent queries. Load balancing and fault tolerance are critical to this service. A previous paper [7] described the design of a result cache service based on consistent hashing and a strategy for enabling fault tolerance. This paper goes further into implementation details and experiments related to the basic scheme to support fault-tolerance which is critical for overall performance. To this end, we evaluate the performance of the RADIC scheme [14] for fault-tolerance under demanding scenarios imposed in the caching service.

1 Introduction

Data centers for large Web search engines (WSEs) contain thousands of processors arranged in high-communicating groups called services. Usually each service is devoted to a single specialized operation related to the processing of user queries. Typically a WSE is composed by three relevant services: Front-End/Broker Service (FS), Caching Service (CS) and Index Service (IS). The FS receives queries and handles query routing; the CS keeps results for frequent queries; and the IS uses an inverted index to calculate top K results when the query results are not in the CS. The CS plays a key role in enabling high query throughput [1] as the cost of searching a query in the CS and returning the answer stored in the respective cache entry, is by far less costly in running time than computing the query answer from the IS.

The traffic generated by WSE users is not uniform neither constant, it is variable, unpredictable and follows Zipfian distributions [3]. It means that users always generate new queries and a few very popular queries can have a huge impact in performance degradation since they can cause imbalance. In addition, failing nodes can affect performance as it is necessary to distribute the load assigned to them on the remaining nodes. The service that is most exposed to imbalance situations is the CS. This is because it splits queries into disjoint sets using hash functions so that each query is allocated to only one partition. Therefore, bursty queries can overload partitions.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 298–310, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The literature related to caching is extensive, but it lacks of efficient solutions for the problem relevant to this paper. A noticeable exception is the Amazon Dynamo [4] system. Almost all of previous work propose eviction algorithms, admission policies and query invalidation strategies to improve the performance of *individual* cache nodes [1,11,6]. We aim at a *distributed* system suitable for clusters of processors. Peer-to-Peer (P2P) systems using consistent hashing [8] -like Chord [15]- assume uniform key distribution, which tends to favour static assignments of keys to nodes. It is important to bear in mind that our application domain is radically different from load balancing P2P systems [16], which means that not all solutions from this domain are applicable to our setting. In our case, queries must be solved in a few tens of milliseconds and thereby there is no room for approaches based on data movement across processors [12], extra messages to locate processors [15], or the like. A fairly similar idea, though fully distributed, is proposed for P2P systems in [2]. This work does not balance the load caused by stored items or their popularity. They indirectly try to do this by making similar the range of keys that each node is responsible for. We do the opposite by using ranges of variable length as a function of node popularity. Our proposal is intended to form part of dedicated search services where homogeneous processors are not shared by other applications and no virtualization technology is used for load balancing because of its overheads.

We propose a dynamic load balancing algorithm upon consistent hashing in order to cope with imbalance in CS nodes. The balancing process is reactive in the sense that it is triggered when imbalance is detected. We also propose to mitigate the effects of node failures by using a protection system for frequent queries. For this purpose we use the RADIC approach [14]. Here, a cache p_i selects a set of pairs <query, answer> according to a criteria, and then these queries are sent to a secondary cache p_j. In case of failure of cache p_i, all of its requests are redirected to p_j (the *protector* of p_i). The protection of selected cache entries is a proactive action.

The remainder of this paper is organized as follows. Section 2 describes the system architecture. Section 3 presents our proposal. Section 4 presents experimental results using simulation, and Section 5 presents results of our RADIC implementation. Finally, Section 6 presents conclusions.

2 System Architecture

The Front-End Service (FS) comprises several replicated nodes. Each FS node receives and routes user queries, and sends back the top K results to users. After a query arrives to a FS node b_i, it asks the Caching Service (CS) to determine whether the query result has been previously stored there. A baseline CS cluster architecture is formed by an array of $P_c \times D_c$ processors (or CS nodes). A scheduling method in FS carries out the distribution of queries onto the P_c partitions. When a partition p_i has been selected, one of its D_c replicas is chosen at random to search for the query. If the query is cached, the CS node sends the query answer to b_i. Afterwards b_i sends the results to the user. If the query is not found in cache, the CS node sends a hit-miss message to b_i. At this point, b_i sends the query to the Index Service (IS).

For the IS, the standard cluster architecture is an array of $P_i \times D_i$ processors or index search nodes, where P_i indicates the level of text collection partitioning and D_i

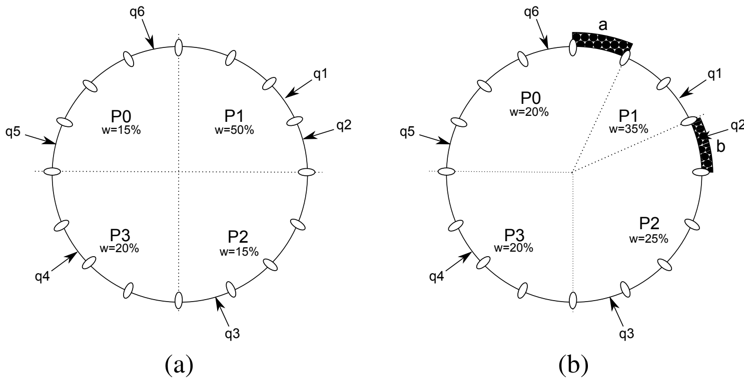

Fig. 1. (a) Consistent hashing assignment; (b) Our proposal for load balancing

the level of text collection replication. The use of this 2D array is as follows: each query is sent to all of the P_i partitions and, in parallel, a random replica in each partition determines the local top K query results. These results are then collected together by the FS to determine the global top K results for the query. Each index search node contains an inverted file which is a data structure used to efficiently map from query terms to relevant documents.

The FS and IS do not experience significant imbalance. Newly arriving queries are evenly distributed on the FS nodes. When a query is solved in the IS, all partitions work in parallel to produce the local top K results in each partition, so a query generates almost the same load in all IS partitions. Neither of these two services face serious risks of imbalance. Given the access pattern to CS partitions and the bias of user queries, the probability of significant imbalance is high in this service as we show below.

The only service in which data (inverted index and text) must be distributed before the operation is the IS. The CS populates its distributed memory with query results on-the-fly, and the FS only handles small data related to current query routing.

3 Caching Service

All memory space of CS nodes is divided into blocks (typically 4KB), and each block stores query terms and their associated top K results (HTML page). Each CS node follows an eviction policy when it is full.

The distribution of items is implemented with consistent hashing [9], which partitions the query space into P independent subsets (each subset can be seen as a non-overlapping arc in a ring (see Figure 1(a)). The idea is that each partition handles all queries (each query is assigned to one point in the ring by means of hashing) that intersect its arc. Figure 1(a) illustrates an assignment following an equidistant distribution of points (please ignore for now the small white ovals), and the assignment indicates that partition $P1$ serves queries $q1$ and $q2$.

The decision of how many partitions depends on the hit ratio we want to reach and the global set of queries (namely, the number of distincts queries in skewed distributions).

The more diverse the query space, the more partitions are needed because each node holding a partition has limited capacity. But dimensioning the number of partitions is not the only decision, also it is necessary to consider the volume of queries per unit of time that one node can accept. To cope with this later issue, replication is introduced as a form to spread the load of a particular partition; also it helps to provide fault-tolerance. This translates into D replicas for each partition. The underlying idea is to select one of the P partitions via consistent hashing, and then select one of its D replicas at random.

To reach a consistent state (all replicas of one partition), we can use an optimistic protocol [13] in each partition where replicas are guaranteed to converge after a period of time. We choose this kind of protocol instead of a protocol that provides strong consistency, because this later solution can saturate the network.

The first baseline approach we study, called *Baseline DAC* (Amazon Dynamo and Chord), can be seen as a matrix of $P \times D$ nodes. The location process works as follows: consistent hashing to select a partition, then we select one random replica. Each partition runs an optimistic consistency protocol.

The previous strategy has a configuration that can be prone to variations in the user query traffic (for example, bursty queries) and nodes failures. To control the load imbalance produced by Baseline DAC, we can use a Greedy algorithm to move replicas between partitions. The key idea is to measure the utilization at partition level each Δ units of time, and then we decide whether one partition needs more replicas taking into consideration the output of the Greedy algorithm. We call this strategy *Baseline DR* (Dynamic Reallocation). The location process is the same as Baseline DAC. Also each partition runs an optimistic protocol. We use these two strategies as a basis for comparison. More details can be found in [7].

Having fixed points in the CHR[1] is not a good policy. Our solution consists in using the same $P \times D$ matrix and a CHR at partition level (i.e., P partitions/arcs), but allowing arc lengths to dynamically change in size to split traffic between neighboring partitions.

Firstly, the ring is divided into small equally-sized buckets to discretize the range covered by each arc. Each partition covers an arc of size $1/P$ of the ring and is responsible of a disjoint set of adjacent buckets as shown in Figure 1(a). Like the strategy Baseline DR, the service utilization is reported every Δ units of time to the FS. If the efficiency[2] is less than a predefined threshold T, a load balancing algorithm that considers each partition utilization is triggered in the FS. The output of the algorithm is a set of bucket movements between neighboring partitions. This prevents all cache entries from being invalidated because only a small number of queries change of partition. Namely, those inside the buckets moved.

The threshold T used in our proposal and the DR strategy defines the maximum degree of imbalance to be tolerated. When the efficiency is below the threshold T, the balancing process is triggered.

Our proposal starts with an initial uniform distribution of the buckets as shown in Figure 1(a). Each partition handles four contiguous buckets. When measuring utilization, we detect that partition $P1$ is processing half of the system load. Figure 1(b) shows the effect of moving one bucket from $P1$ to $P0$ and one from $P1$ to $P2$ (neighbors of $P1$).

[1] Consistent hashing ring.
[2] The efficiency is defined as the average load divided by the maximum load.

The arc a belongs to $P0$ and arc b belongs to $P2$, meaning that $P0$ and $P2$ are now in charge of more queries which decreases the load of $P1$. An advantage of this strategy is that only little portions of cache entries are invalidated when buckets are moved. Namely, those with consistent hashing values falling in the range of the buckets moved to neighboring partitions.

The method to re-distribute the buckets is a diffusion algorithm and it is based on the Sender Initiated Diffusion (SID) algorithm presented in [17]. In this algorithm each overloaded partition distributes excess of load to neighbors with less load. The authors show that in a system with P partitions and load L unevenly distributed, the algorithm will eventually converge to load L/P in each partition and also it is stable.

Replicas associated with each partition are handled as follows. When a FS node selects partition A for query q, we apply a second hash function over the query terms to select one replica from A. This strategy increases the total number of entries available for caching different items across the replicas. This increases overall hit ratio but also node failures are expected to enhance its effects on hit ratio reduction.

To provide fault-tolerance, we use the RADIC framework [14] to efficiently replicate selected queries. It is based on two components: *Protectors* and *Observers* which we propose to use as follows.

Each partition runs a separate RADIC process. Every δ units of time all Observers send their checkpoint to the corresponding Protector. If node m_i belonging to partition A fails, all requests to m_i are re-directed to its Protector m_j allocated in the same partition. In this case, m_j processes its own queries and those originally directed to m_i. The load increment in partition A is not a severe problem due to the balancing algorithm we apply on the partitions. The imbalance generated by a faulty node m_i will be corrected decreasing the range of partition A. Performance degradation is also controlled as the most frequent entries of node m_i are likely to be already checkpointed in its Protector m_j when the failure takes place, so this node will cover the most important queries.

As said, only the most frequent queries in all nodes m_i are protected. Copying all cache entries to the Protector and doubling the number of entries in each node, is not feasible. For the purpose of comparison, to be fair to other strategies, we decrease the available cache entries in each node to make space to hold the checkpoints. From empirical evidence, we conclude that the best distribution in each CS node is 70% of memory space to hold cache entries and the remaining 30% of space to hold checkpoints.

Note that queries tend to be the same between checkpoints in any particular node, since the queries selected to be part of the checkpoint are the most accessed of that node. Hence, instead of forcing the sending of all cache entries to its Protector, only the modifications are sent to it. To this end, each node can log only modifications: priority change, entry eviction and item insertion. This decreases communication.

4 Evaluation through Simulation

To simulate the strategies described above, we have modeled and implemented discrete event simulators that are able to precisely predict a set of metrics. The methodology to build the simulator is based on the facts that (i) the major operations in our context are coarse-grained, and (ii) given our architectural design, a request in any of the nodes

of a specific service takes almost the same amount of resources. The first step is the identification of the most important operations evolved in query processing (resource utilization). Then, we profile these operations and insert their costs to a discrete event simulator. We have previously used and validated this claim in [10]. In conclusion, we simulate trace-driven events, where the traces are benchmarked from real executions on a search engine using query logs of commercial WSEs.

We perform the experiments using a one-day real query log from a commercial search engine (as on April 1st, 2011). The query log comprises 68,019,311 queries. We configured the WSE as follows: (i) the FS comprises 10 replicas; (ii) the CS has $P = 20$ partitions and $D = 4$ replicas; and (iii) the IS possesses 50 partitions and 20 replicas (in order to simulate a complete inverted file loaded into RAM). Each CS node has 100,000 entries for cache. The time interval Δ to measure the partition utilization is 5 minutes and the efficiency threshold T is set to 95%. In our proposal, the service checkpoint is passed every $\delta = 10$ minutes.

Two of the most important metrics are *Average Query Response Time* and *Hit Ratio* of the CS. The first metric shows how the strategies behave under the occurrence of failures, while the second indicates the percentage of answers found in cache. We do not only show the cases of failure, but as well the results without failures for comparison purposes. We start the evaluation with 1, 2 and 3 random failures of CS nodes. Furthermore, we examine a special case, where 10% of CS nodes fail.

In all experiments, failures occur between $x = 640$ and $x = 800$. Note that the nodes where failures were injected are randomly chosen and are not re-incorporated by the service. The measurement starts with the first injected failure.

We labeled the curves in the following figures as “Baseline DAC”, which is the Baseline Amazon Dynamo and Chord, and “Baseline DR”, which stands for Baseline Dynamic Reallocation.

Figure 2(a) shows the average query response while no failures were triggered. Three different trends can be clearly identified in steady state (from $x = 480$): (i) the Baseline DR strategy shows an average of 98 [ms], (ii) the Baseline DAC shows an average of 95 [ms] and (iii) our approach outperforms both with an average of 83 [ms]. This presents 15.3% and 12.6% better query response time than the Baseline DAC and Baseline DR, respectively.

Figure 2(c) shows the results in the case that three failures occur. Here, as well as in Figure 2(b) and (d), it can be observed that the Baseline DR has the worst performance. The reason of this behavior is that the movements of machines between partitions is a disproportionate action, which implies that all entries of the moving nodes are lost (they are not useful for the new partition). The figures reflect the impact. On the other hand, the Baseline DAC as well as our approach present small variations in performance.

A special case is shown in Figure 2(d), in which 8 nodes are randomly chosen to stop. This is an extreme case, since 10% of CS nodes are lost. In this case, the behavior of Baseline DAC remains almost constant. While it is true that our approach experiences an increase of 9% (from 85 to 93 [ms]), it still outperforms the Baseline DAC. Moreover, our approach of dynamic load balancing helps to reduce the average query time. For this reason, a decrease of query time can be observed towards the end of Figure 2(d). Considering this metric, we have demonstrated that the proper combination

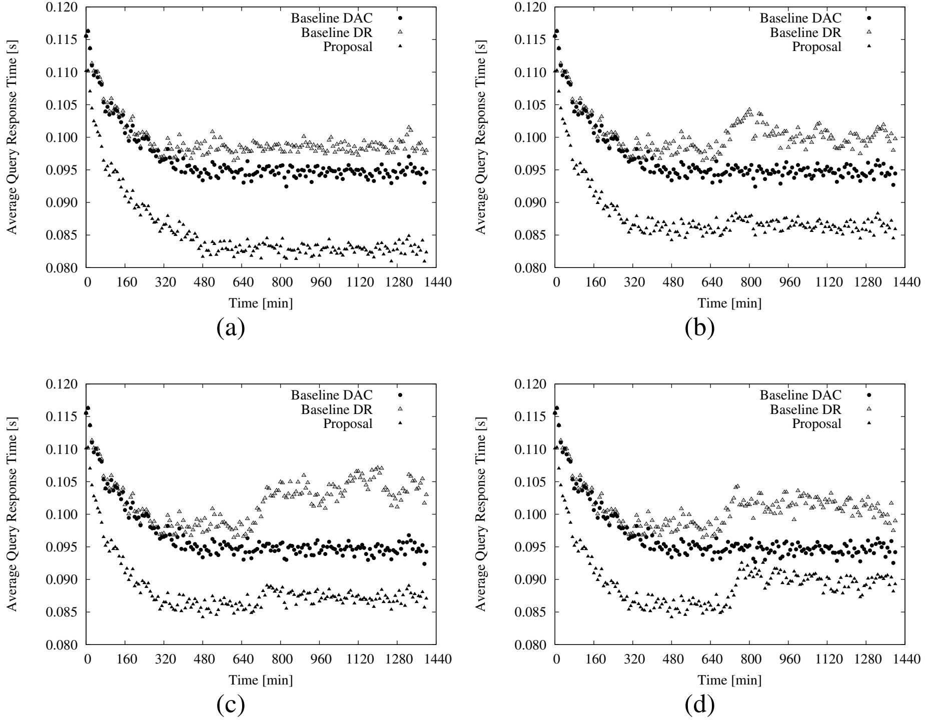

Fig. 2. Evaluation of Average Query Response Time: (a) zero, (b) one, (c) three and (d) eight failures (10%). In cases of failure, they are triggered between $x = 640$ and $x = 800$ minutes.

of dynamic load balancing and a methodology to protect valuable information (RADIC) is important to consider during the design and deployment of caching services. Table 1 summarizes the improvements that we achieved through our approach in all aforementioned cases.

The Figure 3(a) illustrates the hit ratio considering the different options. The performance of Baseline DAC and Baseline DR is similar, having a hit rate between 25% and 30% once the steady state is reached. The optimized utilization of cache entries by our strategy is another important fact. Using our strategy almost all entries show higher hit ratio, while information is only replicated for protection purpose. The proactive replication of queries helps to keep a similar hit rate in case of faults, even in situations where more than one failure occurs (Figure 3(b), (c) and (d)). Despite the failures, our proposal outperforms the other strategies in all cases and is in addition just slightly affected. Figure 3(c) shows the same behavior as before (hit ratio improved by 25% on average compared to Baseline DAC).

Figure 3(d) shows results with greater variation in all cases. This is due to imbalance issues that emerge when a large number of machines fail. The objective of reaching (and keeping) a better hit ratio compared to other strategies is accomplished, despite the high number of failures. See Table 1 for more results.

Table 1. Percentage of improvements of our Proposal against Baseline DAC and Baseline DR considering Figure 2 and 3. Values are obtained while the services were in a steady state (after failures).

	Average Query Time		Hit Ratio	
	Baseline DAC	Baseline DR	Baseline DAC	Baseline DR
Zero Failures	12.6%	15.3%	31%	46%
One Failure	8.4%	13.0%	27%	60%
Three Failures	8.4%	16.3%	32%	94%
Eigth Failures	7.2%	12.7%	37%	105%

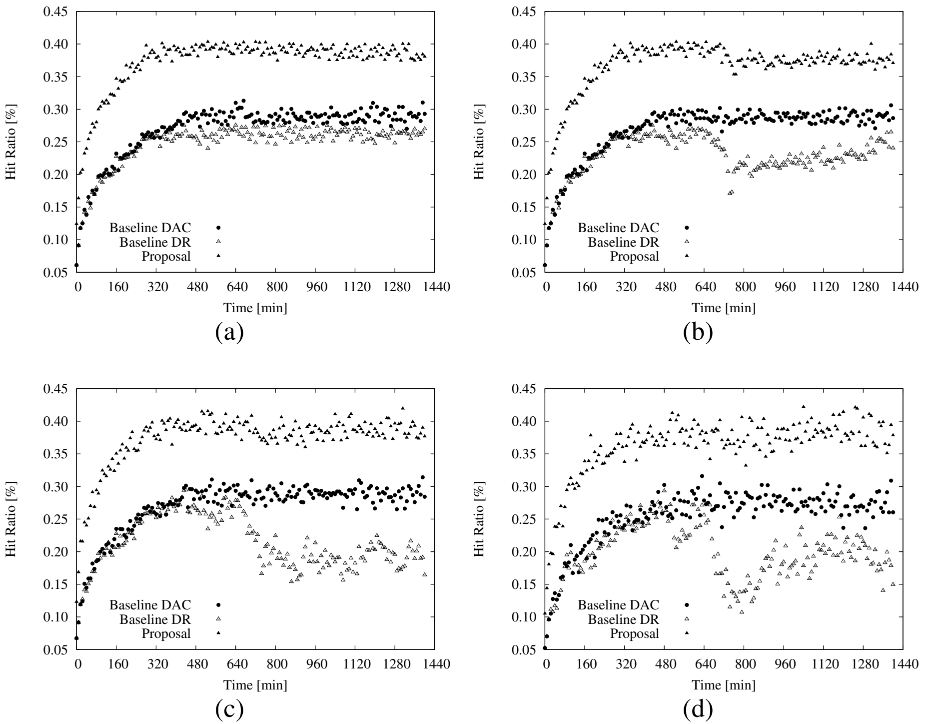

Fig. 3. Evaluation of Average Hit Ratio: (a) zero, (b) one, (c) three, and (d) eight failures (10%). In case of failures, they are triggered between $x = 640$ and $x = 800$ minutes.

4.1 Analysis

We have shown that a better organization of resources, by taking proactive actions (protect important queries) and dynamically balancing load, are important aspects to reach lower response time and higher hit ratios. We argue that the most important factor to achieve a high throughput is a suitable load balancing strategy. Nevertheless, performance grows even more when cache entries are better administered and the consistency protocol is avoided. This technique in conjunction with the protection of queries, improves the performance in all aspects as examined above through the average query

time and hit ratio. Finally, this two techniques help to decrease the impact of failures in case that an organization is used, which exploits all available memory.

Also, the previous results demonstrated the benefits and limits of our proposal. At first, our strategy diminishes its performance in case of failures, but the dynamic load balancing helps to overcome this situation quickly. Moreover, it remains the best strategy. Secondly, the baseline DAC is almost not affected by failures because of the replication, but at the same time it does not utilize the resources properly, and hence this strategy does not attain the best results. Finally, there is a trade-off between replication and performance, and our proposal certainly points in the following direction: only replicate the most frequent queries and use them in case of failures following the RADIC approach.

5 RADIC Implementation

This section describes our RADIC [14] implementation and how it works in a real setting. As we mentioned above, this strategy allows us to integrate the protection of important queries to be used in case of failures. To test RADIC performance, we have implemented a C++/MPI prototype of a CS with RADIC. First of all, each processing node or processor can be seen as a container of entries with a limited capacity that follows an eviction policy when it is full. Well-known algorithms for this purpose are LRU, LFU, SDC and PDC [11]. Regardless of the strategy, in all cases important cache entries can be identified. To decide which is the next entry to be replaced, all algorithms use a priority queue. *Memcached* [5] follows the LRU policy by default (other strategies can be used).

In our service, we designed a priority queue in conjunction with a hash table to implement a LFU strategy. The choice of the LFU strategy is made to simplify the selection of the most frequent queries, which are the ones to be protected by RADIC.

Following the WSE architecture, only one node of the Front-End Service is responsible for triggering the checkpoint process (the decision is centralized at the FS side). Namely, each δ units of time the FS node sends a message to all CS nodes indicating that they must start the checkpoint process. This process consists of three stages in each node CS_i: (i) CS_i gets the most important queries from its memory, which translates into N pairs <query,answers>; (ii) CS_i sends them to its protector node using MPI; (iii) CS_i waits for the checkpoint (N pairs) from the node that it is protecting; and (iv) CS_i stores the received checkpoint in its memory (a separate area of memory). All these phases do not affect the query processing in the node (we control the concurrent access to the structures). As we mentioned above, a node only protects a node of its own partition, in this way the protection on each particion (and their replicas) is independent of other partitions.

To study the behaviour in case of failures, we did not need a fault-tolerant version of MPI, because the routing and handling of queries belongs to the FS and only this service needs to know when a node falls. Failing nodes are selected randomly and, for the sake of simplicity, we "simulate" a failure sending a MPI message to the failing nodes (to stop the processing), and then updating the state of active nodes in the FS (routing tables). Once a failure is detected in the FS, all requests to failing nodes are

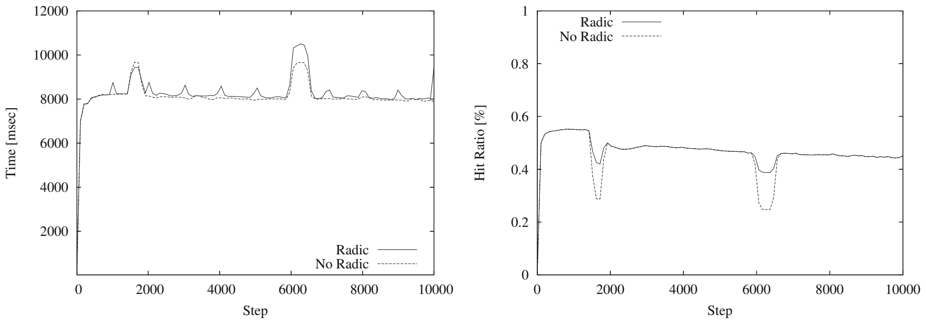

Fig. 4. Real deployments to evaluate overhead with failures: (a) running time of the complete caching service, and (b) hit ratio of a single partition of the caching service

routed to the protector of them following the RADIC algorithms/tables. As the protector maintains information regarding the failing nodes (checkpoint), it starts to process the redirected requests taking into account this information. This helps to preserve the hit ratio in case of faults.

5.1 RADIC Overheads

An important performance metric in a fault-tolerant system is the overhead imposed by the protection scheme relative to the same system without a fault tolerance strategy. Hereinafter, we evaluate the RADIC overheads through an actual implementation running in our CS service. The implementation is therefor deployed to a cluster of processors (cluster composed of 20 nodes connected by an InfiniBand switch). To effectively measure overheads and to therefor achieve a situation in which there are no queries that cause imbalance, we implemented the baseline approach (namely, the $P \times D$ matrix described in Subsection 3) with the RADIC system running in background.

We ran the complete log described in Section 4 in a caching service with P=5 partitions and D=4 replicas. We measured the time required to finish a set of queries and the resulting hit ratio (each measurement is a point in the X-axis, and only the first 10,000 points are plotted). Furthermore, we injected two failures in the system (x=1,500 and x=6,000). After the failures, nodes are re-inserted in the service after 100 y 500 steps, respectively. Note, that the implementation without RADIC has 100,000 cache entries per node, while the implementation with RADIC has 95,000 cache entries and 5,000 for checkpoints.

Figure 4(a) shows the time required to finish the queries. The overhead imposed by RADIC is 1.8% on average, what does not represent a big impact on the service. The checkpoint takes place approximately every 1,000 steps and only the top 5% of the most important queries are checkpointed. We optimized the checkpointing process by processing it through a pipeline: multiple steps are used to send the checkpoint to the Protector. The overhead and the checkpoints become important when failures occur and when queries are protected. Figure 4(b) displays the results in terms of average hit ratio inside a partition when failures occur in the same partition (D=4). It is clear that the hit ratio is less affected by failures since the implementation of RADIC allows to continue

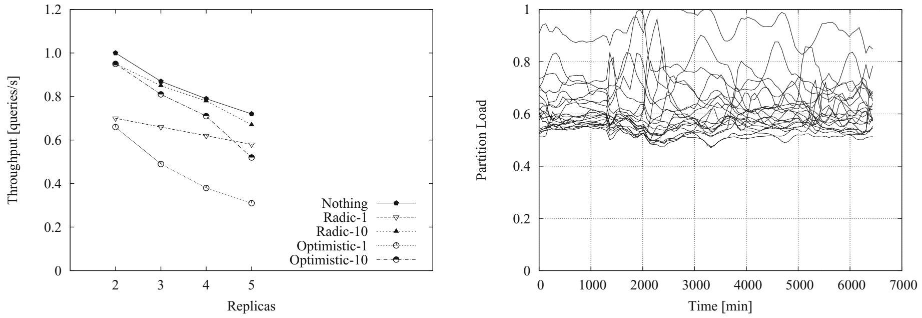

Fig. 5. (a) Evaluation of a real implementation of the Optimistic Protocol and the RADIC Proposal (Section 3) to test the scalability of one partition. (b) Load assignment to 20 partitions when we run 300 millions of queries issued during May 1-5, 2011.

the operation using the checkpointed query results in the Protector of the failing node. The average hit ratio of the partition using RADIC is 47.2% and 46% without RADIC (an increase of 2.6% in the presence of failures). We would like to remark that the results of this section were obtained with the same baseline strategy, which was extended with RADIC to expose its inherent overheads relative to the same strategy operating without RADIC.

Figure 5(a) shows results obtained with actual implementations of the consistency protocols. We ran these experiments in a cluster of processors connected by a commodity switch, taking care that each replica is allocated to a different processing node so that communication among processors takes place through the communication network. The purpose of these experiments is to evaluate the effects of extra communication required to keep consistency across the replicas of each partition at running time. The curve labeled "nothing" is the best that a protocol can do as in this case no communication bandwidth is used to replicate cache entries. They just arrive to a target replica and a search in the cache is executed. Here it is not relevant whether there is a cache hit for any query or not. Note that each replica is assumed to receive the same number of queries. This implies that as the number of replicas grows, more queries are processed in total. The remaining curves are showing the results of RADIC and the optimistic protocol for cases in which the consistency protocol is executed each 1 and 10 minutes. In both cases, the RADIC protocol outperforms the optimistic one. For 10 minute intervals the RADIC protocol achieves a performance quite similar to the optimal case. The overhead in terms of memory consumed by checkpoints is negligible because hit rate is not affected significantly keeping constant the total number of cache entries.

6 Conclusions

We conclude by referring to Figure 5(b) which shows that queries tend to produce significant imbalance when no strategy is used to load balance the amount of queries that receive each CS node.

We have shown that load balancing and protection of queries can help to improve the performance of WSEs. On the one hand, we use a load balancing algorithm to

cope with user query variations and faulty nodes, this is a reactive action. Then, we move forward proposing the application of RADIC methodology to protect valuable information, which is a proactive action.

The objective of this paper was twofold: (i) analyze the resilience of our proposal, and (ii) analyze the performance and usefulness of RADIC framework. In section 4, the experimental results show that our proposal outperforms the commonly used baseline alternatives by a wide margin as shown in Figure 2 and 3 ((a) no failures; (b), (c), (d) with failures). This is because our proposal is able to significantly reduce the imbalance shown in Figure 5(b). Notice that our proposal can be easily extended to clusters with heterogeneous nodes since load balance is made considering only the utilization of processors, which can be determined by performing benchmarks on individual nodes and stablishing a relationship between incomming query traffic and utilization.

Finally, in section 5 we evidence that our RADIC implementation imposes a very low overhead to the query processing tasks, which shows its competitiveness and usefulness in the context of caching services. To the best of our knowledge, there are no previous works that addresses the protection of queries.

Acknowledgements. This research has been supported by the MICINN Spain under contract TIN2007-64974 and the MINECO (MICINN) Spain under contract TIN2011-24384, and partially funded by FONDEF project D09I1185. The first author (CG) has been supported by a Chilean PhD scholarship from CONICYT.

References

1. Baeza-Yates, R., Gionis, A., Junqueira, F., Murdock, V., Plachouras, V., Silvestri, F.: The impact of caching on search engines. In: SIGIR 2007, pp. 183–190 (2007)
2. Bienkowski, M., Korzeniowski, M., auf der Heide, F.M.: Dynamic Load Balancing in Distributed Hash Tables. In: Castro, M., van Renesse, R. (eds.) IPTPS 2005. LNCS, vol. 3640, pp. 217–225. Springer, Heidelberg (2005)
3. Breslau, L., Cue, P., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: INFOCOM, pp. 126–134 (1999)
4. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. SIGOPS 41, 205–220 (2007)
5. Fitzpatrick, B.: Distributed caching with memcached. Linux J. (2004)
6. Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: WWW 2009, pp. 431–440 (2009)
7. Gómez-Pantoja, C., Gil-Costa, V., Rexachs, D., Marin, M., Luque, E.: A fault-tolerant cache service for web search engines. In: ISPA 2012 (to appear, 2012)
8. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: ACM STOC 1997, pp. 654–663 (1997)
9. Karger, D., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y.: Web caching with consistent hashing. In: WWW 1999, pp. 1203–1213 (1999)
10. Marin, M., Gil-Costa, V., Gomez-Pantoja, C.: New caching techniques for web search engines. In: HPDC 2010, pp. 215–226 (2010)
11. Perego, T.F.R., Silvestri, F., Orlando, S.: Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. In: ACM TOIS 2006, pp. 51–78 (2006)

12. Raiciu, C., Huici, F., Rosenblum, D.S., Handley, M.: ROAR: Increasing the flexibility and performance of distributed search. In: SIGCOMM 2009, pp. 291–302 (2009)
13. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37, 42–81 (2005)
14. Santos, G., Duarte, A., Rexachs, D., Luque, E.: Providing Non-stop Service for Message-Passing Based Parallel Applications with RADIC. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 58–67. Springer, Heidelberg (2008)
15. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM 31, 149–160 (2001)
16. Surana, S., Godfrey, B., Lakshminarayanan, K., Karp, R., Stoica, I.: Load balancing in dynamic structured peer-to-peer systems. Perform. Eval. 63, 217–240 (2006)
17. Willebeek-LeMair, M., Reeves, A.: Strategies for dynamic load balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems 4(9), 979–993 (1993)