2009 11th IEEE International Conference on High Performance Computing and Communications

Cache-aware load balancing vs. cooperative caching for distributed search engines

David Dominguez-Sal
Computer Architecture Dept.
DAMA-UPC
Barcelona, Spain
ddomings@ac.upc.edu

Marta Perez-Casany
Applied Mathematics II Dept.
DAMA-UPC
Barcelona, Spain
marta.perez@upc.edu

Josep Lluis Larriba-Pey
Computer Architecture Dept.
DAMA-UPC
Barcelona, Spain
larri@ac.upc.edu

Abstract

In this paper we study the performance of a distributed search engine from a data caching point of view. We compare and combine two different approaches to achieve better hit rates: (a) send the queries to the node which currently has the related data in its local memory (cache-aware load balancing), and (b) send the cached contents to the node where a query is being currently processed (cooperative caching). Furthermore, we study the best scheduling points in the query computation in which they can be reassigned to another node, and how this reassignation should be performed. Our analysis is guided by statistical tools on a real question answering system for several query distributions, which are typically found in query logs.

1. Introduction

The construction of distributed search engines is a complex task where many components with high computational cost interact. New search engines combine additional modules to refine an answer and achieve a better precision. However, these more advanced features come with large computational costs that must be addressed to make systems scalable. We take Question Answering (QA) as an example of these next generation search engines. QA systems return short, precise answers, e.g., person and location names in response to natural language questions [1]. For example, a QA system that receives as input the question "In which city is the Eiffel Tower?" will answer "Paris".

Caching and distributed systems are two fundamental pillars required to improve the final performance of these systems. In this paper, we study how these two factors interact and how they impact the performance of a fully fledged search engine.

A search engine receives many queries with overlapping computation: queries may share terms, they may access similar document sets or completely different queries may be looking for the same answer [2]. In these common scenarios, caches are crucial because they store these partial results in the main memory, thus saving execution time for subsequent queries. In this paper, we implement a cooperative cache that enables all the computers in the system to introduce and retrieve data from the system transparently, similar to a regular local cache. The cache is managed in accordance to the recent accesses to data in each node of the network. The system records the dataset that is most frequently accessed locally and disseminates a summary of this dataset to the rest of nodes. This information is updated dynamically and is used by all the nodes in the network to decide which is the best node to place a document, and to control the number of replicas of a document in the distributed system.

Moreover, we implement a load balancing algorithm that is cache-aware. The objective of a load balancing algorithm is to assign the workload following a policy that optimizes the overall system performance. The load balancing considers not only the current load in each node but also the expected real execution time of the query, given the state of the global cache. Thus, a query may not be assigned to the most idle node but to a node which has the query partial results cached, which in the end yields a faster query execution.

Load balancing and cooperative caching are two useful techniques to improve the throughput of a system. However, to our knowledge there is no previous work that studies the interaction of cache aware load balancing algorithms with cooperative cache algorithms. In this paper, we combine these two techniques and analyze the interaction between them. Both techniques implement different facets of a global management scheme that improves the data locality for the executed queries. On the one hand, cooperative caching sends the information where the queries are being currently processed. On the other hand, the load balancer applies an alternative policy: it sends the queries to the nodes that currently have the data stored. We study with statistical tools wether any of the two approaches alone is sufficient or if they can be successfully combined. Furthermore, we use a similar approach to understand the system configuration

The authors want to thank Generalitat de Catalunya *for its support through grant number GRE-00352 and Ministerio de Educación y Ciencia of Spain for its support through grant TIN2006-15536-C02-02, and the European Union for its support through the Semedia project (FP6-045032).*

978-0-7695-3738-2/09 $25.00 © 2009 IEEE
DOI 10.1109/HPCC.2009.31

such as the importance of scheduling points in the system, or if a new reassignation of queries once received in a node is beneficial. We perform all the analysis for several query distributions and on a fully-fledged QA system.

The paper is structured as follows. In Section 2, we describe the distributed QA system, the cooperative cache algorithm and the cache-aware load balancing algorithms tested. Section 3 reports the experimental results and its corresponding discussion. In Section 4, we review the related work. Finally, Section 5 concludes the paper.

2. QA architecture

In this paper, we use a fully-fledged factoid QA system, whose implementation details are presented in [3]. We depict the system modules in Figure 1. The implementation of the QA system follows a traditional architecture of a pipeline with several sequential *computing blocks*: (i) Question Processing (QP), which analyzes the query, understands the question focus, and transforms the natural language question into a traditional Information Retrieval (IR) query; (ii) Passage Retrieval (PR), which is an IR system that obtains from disk the set of the most relevant documents for a query; and (iii) Answer Extraction (AE), which applies natural language processing tools to process the documents extracted in PR and identify the most relevant answers for the query. The system is modular and we can vary its configuration to test its performance in different environments.

From a data processing perspective, our QA system implements a two-layered architecture: first, we extract the relevant content from documents that are lexically close to the input question, and second, we semantically analyze this content to extract and rank short textual answers to this question, e.g., named entities such as person, organization, or location names. Because both these blocks are resource intensive, the former in disk accesses and the latter in CPU usage, we implement a cache in main memory for each stage. The first layer caches the documents read from disk in PR, and the second caches the document analysis coming from AE. This local cache configuration is analyzed in [3]. This system obtained state-of-the-art performance in an international evaluation [4].

Our computing architecture is a cluster of SMP nodes, connected by a local area network. In order to build a distributed system, we replicate the local system in each node of the network. QA systems with text collections that are too large to be replicated can partition the collection and assign each partition to a group of nodes [5], in which each group behaves similarly to our architecture.

Each query runs in its own thread, so several queries can be simultaneously executed in a node, even if they are executing the same computing block. The computation is performed in several steps decided by the scheduling points described below. In our system, the set of CPUs on a node share a waiting queue for pending tasks. We allow one more active query threads than CPUs in order to avoid having multiple threads competing for the same resources. If a query is going to start the execution of a computing block and there are no resources available, the query is queued until another query finishes its computing block. In addition to the query computing threads, the system implements several management threads: (a) a planning thread that monitors the incoming queries, starts the execution of incoming and waiting queries and decides if a query must be relocated to a different node; (b) an Evolutive Summary Counter thread that summarizes the local information and receives the updates from other nodes (see the section below); (c) a cooperative cache thread that serializes the cache victims and decides its forwarding target. The communications are implemented over TCP, except the ESC-summary diffusion, which is over UDP. We choose a multithreaded architecture because it decouples the search engine internal management computation with query computation, it is modular and takes advantage of the current multicore processors.

Summary of system caches: The distributed QA system implements an algorithm to monitor the state of the caches in each node of the network efficiently. Each node maintains a data structure, called Evolutive Summary Counters (ESC, described in [6]), that keeps a record of the recent documents accessed in a node (during both PR and AE). ESC monitors what documents are accessed in each node, and it can be used to monitor the current state of the distributed cache. The data structure is shared by all the cache aware algorithms. An ESC is similar to the summary caches proposed by Fan [7]: both report recent information about the nodes in the network. Both structures use Count Bloom Filters (CBF), that is a variant of Bloom Filters [8] to count the approximate number of elements in a set. Like Bloom Filters, a CBF is very compact because it keeps an approximate count that can differ from the real value, with a fraction of error that can be tuned as desired. In both proposals, summary counters and ESC, each count filter is active for a certain period of time in a round robin fashion and, at certain intervals of time, each computing node generates a summary of its local CBFs and sends the summary to the rest of nodes. However, the summary caches report only the current contents in the cache, while the ESC summaries cover all the documents read in the recent history. This difference is important because an ESC contains information about the usage of non frequent documents, which may not be cached, but it is needed to improve the load balancing of the system. All in all, in our QA system, each node receives an ESC summary from each node in the network: the ESC summary received from a node i contains the number of times that document d has been read recently in that node, $ESC_i(d)$. The target architecture for ESC is inspired by the design of distributed search engines: a huge data collection

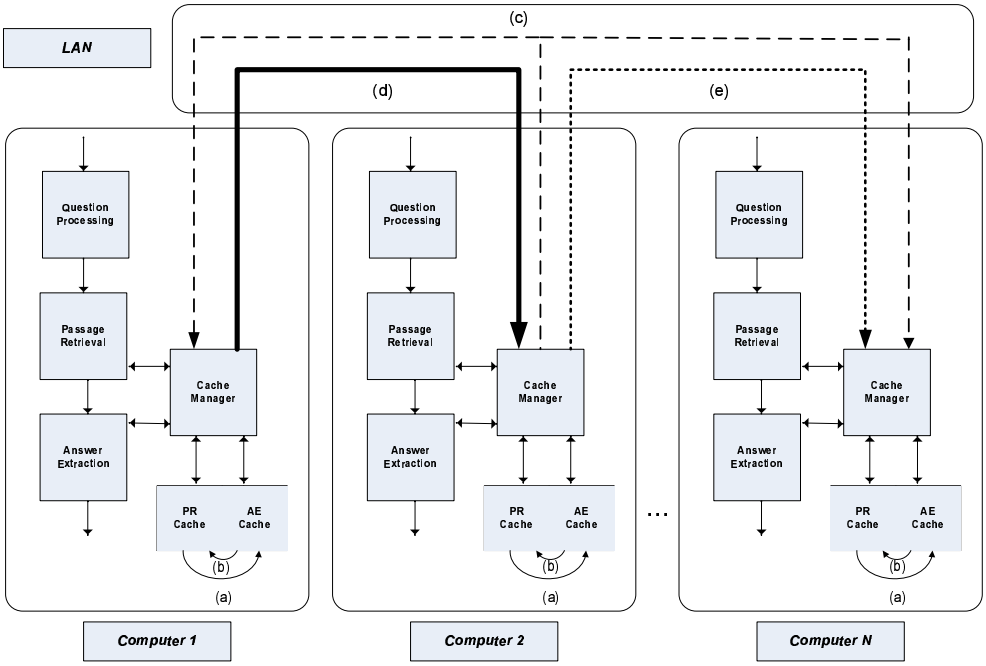


Figure 1. Diagram of the three computing blocks of our QA system: QP, PR and AE. The figure depicts the operations of the cooperative cache manager: (a) store a document processed in AE; (b) store a document read from disk in PR; (c) request a document in the cooperative cache; (d) send the requested data through the cooperative cache; (e) transfer the least recently used entry to a different node.

is divided in several partitions [9], each of these managed by a separate group of nodes that execute queries related to its partition (with the aid of the ESC) and share a local area network. For other configurations with large pools of nodes, it would be necessary to partition the nodes in teams to avoid excessive network traffic from the ESC-summary diffusion. Note that the ESC only use the network during periodic updates. However, a node can check at any moment the number of times a document has been accessed in a certain neighbor node without any new communication.

2.1. Cooperative cache

We deploy a cooperative cache algorithm that relies on the information disseminated by the ESC. The cooperative caching algorithm is in charge of the placement and location of the cached contents [6]. The placement algorithm is triggered for each victim of the local cache policy, which in our system is the least recently used entry. The node checks the summaries received from the rest of nodes, and forwards the entry to the node that has the largest number of accesses to that document. Thus, the procedure sends the victim to the node which is more probable to cache the entry. We store a counter with each cached entry, which stores the number of forwards since its last access. This counter is incremented after each forward and is reset when an entry is accessed. Entries whose counter is above a certain threshold are not forwarded and are discarded from the cache. This policy avoids long chains of forwards and removes the unused entries similarly to a global LRU policy. Our search algorithm in the cache is similar to ICP [10]: a node queries all the rest of nodes in the network to retrieve the data associated to a document identifier, and if any node has the contents available in its cache, it sends the requested data to the querying node (operations (c) and (d) in Figure 1). Once the data is received, it is added to the cache of the requester node. The cooperative cache is able to retrieve cached entries for both PR and for AE blocks: the full raw text of the document as well as its natural language analysis. Following this procedure, any node can see the cache contents of the rest of nodes that belong to the distributed QA system.

2.2. Load balancing

We implement a cache-aware load balancing algorithm with several scheduling points, situated before each computing block, as described in [11]. When a query reaches the scheduling point, the node triggers the load balancing algorithm to decide in which node the query is going to continue its execution. If the load balancing algorithm decides that the query should continue running locally, then the query continues its execution immediately, or it

is queued if there are no resources available for that task. If the load balancing algorithm selects a remote node, the query is packed and transferred to the selected node. Each time a query finishes a computing block in the system, all the queued queries are rescheduled by the load balancing algorithm again with the information received from the rest of nodes since the last update. A query that is waiting in the queue to be executed locally can thus be rescheduled and assigned to a new node because, for example, the remote node has new cached contents or is less loaded.

Each node i measures its current load in two dimensions: one for the I/O ($Load_{(i)}^{IO}$) and another one for the CPU ($Load_{(i)}^{CPU}$). Each node sends its load measure to the rest of the nodes in the network periodically or if their current value differs in more than fraction since its last update (25% difference in our experiments). Summarizing, all the nodes compute their local load, and receive recent load information from all the computing nodes. Additionally, we use this periodic communication to detect on the fly when a node is not available, and when a new computing node has joined the network, thus, tolerating hardware faults.

The cost to process a query in a node is estimated according to the next computing block. The algorithm stores a history record of the CPU and I/O time from previous queries and applies this record to estimate the fraction of CPU and I/O of the next computing block of the current query. The cost to compute the query q in node i is a weighted sum of the node load ($Load_{(i)}^{IO}$ and $Load_{(i)}^{CPU}$), averaged by the fraction of time that the query q is going to spend in each of the dimensions ($W_{(q)}^{CPU}$ and $W_{(q)}^{CPU}$) [12]. We call this combined cost $Load_{(i,q)} = W_{(q)}^{CPU} \cdot Load_{(i)}^{CPU} + W_{(q)}^{I/O} \cdot Load_{(i)}^{I/O}$.

In order to make the previously described algorithm cache-aware, we implement two improvements [11]. The first is a more accurate cost prediction, which reduces the cost to process documents in the nodes that hold the information already cached. The cache contents in a node are inferred from the information contained in the ESC. The system computes the probability to find document d according to the recent accesses of d in a node, which are stored in the ESC. The intuition behind the search algorithm is that the more frequently a document is accessed, the higher the probability it is to be cached in that node. So, each node estimates a different hit probability for each different document access frequency. This estimation is corrected dynamically in such a way that the probability of hit is increased for future queries when a document is found, and the probability is reduced otherwise. The cache algorithm uses this probability to give a more accurate computing cost of a query according to the global cache state. The cost to process a document is weighted by the expected hit probability: a document with an expected hit probability 90% accounts for a lower computational cost than a document not expected to be found in the cooperative cache. The algorithm distinguishes two types of hit: cooperative cache hits (remote hits) and local hits. Although the former is more expensive than the later because of the network,. cooperative caching is much faster than the recomputation of a query. The algorithm records the computational cost of the latest local and cooperative cache hits to dynamically estimate the cost of new queries. A more detailed description of the search algorithm can be found in [6]. We name this new estimation of the cost $Load\_cache_{(i,q)}$.

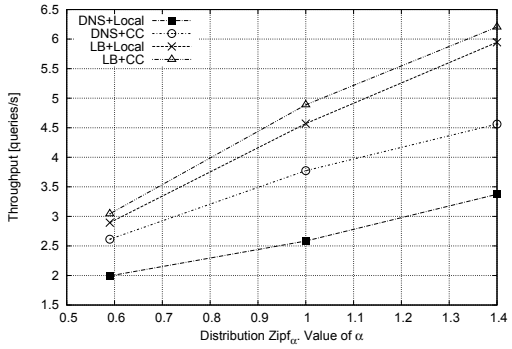
The second improvement is the addition of a new term to select preferably servers whose cache contents are similar to the data requested by the query [11]. We compute the similarity between a query and the cache contents in a node with a formula frequently used in information retrieval for this task: the $tf \cdot idf$ [13] (term frequency - inverse document frequency). The formula scores high for the nodes that already contain the data locally, and low otherwise.

Eventually, the node selected to continue processing a query corresponds to the node which scores the lowest value for the combination of the computing cost of query q, and the similarity between q and the cache contents: $Load\_cache_{(i,q)} \cdot \frac{1}{tf \cdot idf}$ [11]. The previous formula favors servers which have the resources to process the next query available, and their cache contents have a good affinity with the current query.

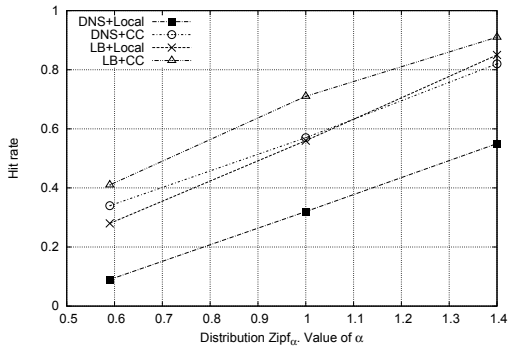
3. Experimental Results

We performed several experiments to analyze the combination of cooperative caching and load balancing. The experiments are performed incrementally, i.e., every new experiment uses the best configration from the previous experiment. First, we compare the throughput of Question Answering systems with and without cache-aware load balancing and cooperative caching. Then, we set different scheduling points in the system and measure the effect of each of these points. And finally, we study if it is necessary to perform multiple query forwards to improve the system performance.

Setup: For our tests we use the fully-fledged QA system described above, running on a cluster of 16 nodes connected with a gigabit Ethernet network. Each node in the system is equipped with an Intel dual core CPU at 2.4GHz and with 2GB of RAM. We use as textual repository the TREC document collection [14], which has approximately 4GB of text in 1 million documents. The database in our experiments is replicated, and in case a document is not available in cache, each node can load it from its local disk. An additional computer is used as a client that issues each new query to a different computer in a round robin fashion. The question set uses 700 different queries to create a workload with a sequence of 5000 queries. The 700 queries in the final workload follow Zipf$_{\alpha=0.59}$, Zipf$_{\alpha=1.0}$ and Zipf$_{\alpha=1.4}$



(a)



(b)

Figure 2. (a) Throughput of a system with different configurations of cache and load balancing. (b) Hit rate of a system with different configurations of cache and load balancing

distributions. We choose these distributions as a result of several analyzes of query logs from different web engines: the former due to a study from Saraiva et al. [15] where they analyzed a query log which fitted a Zipf$_{\alpha=0.59}$; and the rest as a sample of more skewed distributions that can be found in other studies such as [16], [17]. The questions from the query sets were selected from questions that were part of former TREC-QA evaluations (700 different questions). The client issues the queries to keep the system under a high load, with an average of eight simultaneous queries per node.

The design of all the experiments follows a similar pattern. We performed factorial analysis[1] to analyze the system throughput: we picked the set of factors (variables) to study, and for each possible configuration of the factors and distribution we obtained three observations. The testing script executed the configurations in random order and cleared the computing nodes and its caches after each execution. Once we obtained the performance observations, we analyzed them with the statistical package SPSS [20]. All our statistical conclusions are obtained with a significance level of 0.05. The plots included in the article correspond to the average of all the observations for that configuration.

1. The analysis of variance (ANOVA) is an statistical technique to define models that quantify the changes of a system's outcome to a set of *factors*. We call factor to categorical variable with a small number of levels, under investigation in an experiment as a possible source of variation [18]. In this paper, we apply several ANOVA techniques [19], which are commonly drawn from *factorial experiments*, where the scientist defines a set of relevant factors that may affect the system and tests the outcome for all combinations of factors.

3.1. Load balancing and Cooperative Caching

In this experiment, we enable and disable the cooperative caching and the cache-aware load balancer to quantify if the improvement of each technique is statistically significant and to see if they exhibit interaction. The cooperative caching (CC) corresponds to a system where data is transferred to the node processing the query, and the cache-aware load balancer (LB) corresponds to a system where queries are sent to the node with the cached content. When the cooperative caching is disabled we keep local caching in each node activated (Local), and when we disable the load balancer we assign the queries following a round robin policy (DNS). Thus, we test the resulting twelve configurations of a complete factorial design compound by two binary variables and a variable with three levels: the cache policy (CC or Local), the load balancing algorithm (LB or DNS) and the distribution (α = 0.59, 1.0, 1.4). For each configuration, we repeated the experiment three times, which adds up to 36 observations in total.

In Figure 2(a), we plot the throughput for each of the configurations and query distributions. We observe that the activation of either cooperative or load balancing improves the system importantly, over the system without any of these techniques (DNS + Local). The improvement from the addition of a cooperative cache is up to 46%, and the cache aware load balancing increases the system throughput up to 77%. However, the best system is when both techniques are combined with an increase in the throughput of 90%.

We analyzed our results with a General Linear Model [19] that is a statistical procedure to quantify the variance introduced by each factor in an experiment. We tested different models in order to fulfill the parsimony principle, and picked as our final model the one that takes into account all the the three variables (the distribution, the cooperative cache, and the load balancing) plus the interaction between the cooperative cache and the load balancing. Hence, the resulting model is the following:

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + (\beta\gamma)_{jk} + \epsilon_{ijkl}, \qquad (1)$$

where μ is the overall mean of the observations; α_i is the effect of the i-th level of the query distribution; β_j is the effect of the j-th level of the cooperative cache; γ_k is the effect of the k-th level of the load balancer; $(\beta\gamma)_{jk}$ is the effect of the interaction between the j-th level of the cooperative cache and the k-th level of the load balancer; ϵ_{ijkl} corresponds to the experimental error (or residue); and,

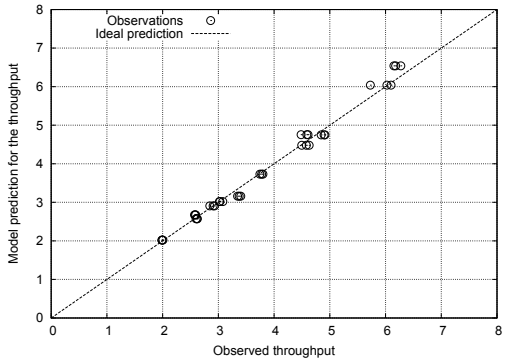


Figure 3. Factorial analysis for three factors: cache-aware load balancing, cooperative caching and query distribution. Predicted throughput by the model vs. observed throughput (Equation (1)).

Factor	Level	Description
α_i	i = 1	Distribution Zipf$_{\alpha=0.59}$
	i = 2	Distribution Zipf$_{\alpha=1.0}$
	i = 3	Distribution Zipf$_{\alpha=1.4}$
β_j	j = 1	Local cache
	j = 2	Cooperative Cache
γ_k	k = 1	DNS
	k = 2	Cache-aware Load Balancing

Table 1. Description of the factors in Equation (1).

y_{ijkl} is the throughput of the l-th observation for the system configured with the levels i-th, j-th and k-th. We detail the correspondence between the levels of a factor and the system configuration in Table 1. The GLM procedure estimates the value of each term by the minimum least squares method, which minimizes the sum of the squares of the residual terms of the model.

The statistical tests for the model indicated that all the included terms were statistically significant, and the response variable (i.e. the overall system performance) strongly depends on the independent variables (i.e. the cache, the load balancer and the distribution.). The estimated model is very precise: $\mathcal{R}^2 > 0.98$, which means that only less than 2% of the variability is not explained by the model. Thus, we observe in Figure 3 that the correspondence between predictions and observations lies very close to the identity function, which is a perfect fit.

The model indicates that the best configuration activates both the cooperative caching and the load balancer and is statistically better than the other configurations. Nevertheless, the model quantification shows that the benefit from moving queries is four times larger than the one coming from moving the cached contents. The reason for this result is the data size: a query is much smaller than a document. In other words, it is faster to transfer the queries through the network than the data requested by a query. However, cooperative caching is still valuable because it introduces a global management of the cache contents that turns into better hit rates. Figure 2(b) shows that both techniques improve the hit rate significantly and in a similar amount. But, cooperative caching and cache-aware load balancing increase the system hit rate from two different perspectives. Thus, when we combine them, the hit rate is better than any of them individually as can be observed in Figure 2(b). The model proves this with the quantification of the interaction: although it exists and the benefits do not accumulate linearly, the contribution of the interaction is smaller than that from the main effects. This proves that both techniques are complementary.

The model shows no interaction between the distribution and the rest of factors. The model estimates the benefit of caching and load balancing as a constant improvement independently of the query distribution. This means that the two techniques are effective for all the query distributions tested, which are the commonly found in the logs of real search engines. Indeed, the system execution time is smaller for skewed distributions because the data set of frequent items is smaller, and the baseline cache is more efficient under this circumstances.

3.2. Scheduling points

According to the previous experiment, the load balancer plays an important role to improve the system performance. We test here which are the most advisable locations to include scheduling points in a question answering system. We introduced up to four scheduling points in the system before each of the computational blocks in the system: (a) when query is received, before QP; (b) before accessing the indexes of the collection in PR; (c) before reading the documents from the collection in PR; (d) before processing the received documents in AE. We tested five configurations with an increasing number of scheduling points. The first configuration only enables the scheduling point for the most expensive computing blocks, and we sequentially add more scheduling points according to the following most expensive computing block. We tested the following combinations: only (c) or (d) enabled, (c+d) enabled, (b+c+d) enabled, and finally (a+b+c+d) enabled.

The execution time for each configuration is plotted in Figure 4. We analyzed our results with a General Linear Model [19], which includes only the main factors, without any interaction between the distribution and the scheduling points. The model was statistically valid for all levels of the tested variables with a high $\mathcal{R}^2 = 0.99$, that means the model predictions correlate significantly the performance and the scheduling points.

In Figure 4(a), we observe that the number of scheduling points influence the system throughput: the more scheduling

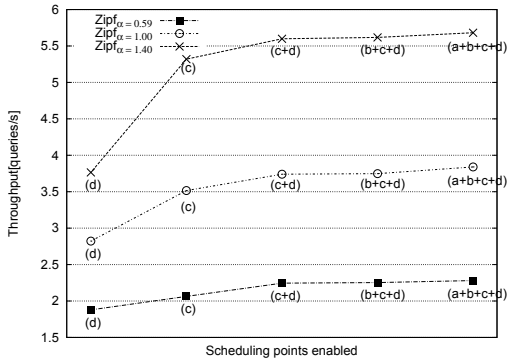


Figure 4. (a) Throughput of the system with different scheduling points enabled. (b) Number of forwards per query with different scheduling points enabled.

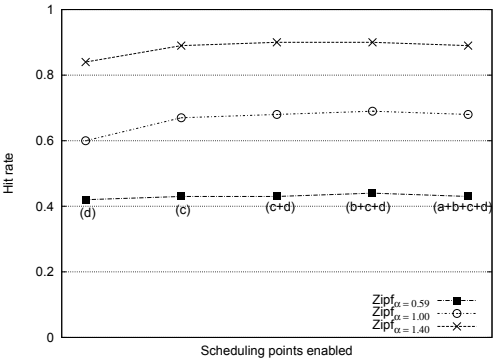


Figure 5. Hit rate for different combinations of scheduling points.

points the better performance. Nevertheless, the addition of some scheduling points (a and b) does not affect significantly the system throughput. We confirmed this intuition by computing a set of contrasts among the different configurations of scheduling points [19]. The contrasts showed that configurations (a+b+c+d), (b+c+d) and (c+d) showed no statistical difference in the system throughput, and all of them were better than the single scheduling point configurations. We also recorded the number of forwards for each configuration, which are reported in Figure 4(b). The plot shows that (c+d) requires the smallest number of forwards among all the multiple scheduling point configurations, and consequently takes less network traffic.

The hit rate does not depend on all scheduling points equally (Figure 5). The PR scheduling point is the most relevant from a hit rate perspective. Once we enable PR as scheduling point (c), the hit rate is the same as with all the scheduling points enabled (a+b+c+d). But, considering the final performance of the system, (c+d) is the preferable option. If we compare system (c) with system (c+d), we observe that most queries are forwarded when they reach PR because they are transferred to a node with cache contents affine to the query. Almost all queries change their execution node after (c). The probability to reassign a query is smaller when they reach the AE scheduling point. Only a few queries (about a third of the total) change their executing node because the query is already in a good node from the cache perspective. We note that the source of forwards in AE come from a different source than from PR: queries are forwarded due to load unbalances in the cluster. Thus, it seems plausible that cache-aware algorithms may be improved if they become flexible: first node assignments should be more cache oriented, and then the query can be transferred to an underloaded node if severe unbalance is detected.

Although the distribution modifies the throughput of the system, the model shows that the best set of scheduling points does not depend on the distribution because there is no significant interaction between the distribution and the scheduling points. According to our experiments and the statistical model generated, a load balancer for a Question Answering system should include two scheduling points in PR and AE (c+d), because among the best possible configurations it is the simplest, generates a small number of forwards, and achieves the best or close to the best performance.

3.3. Number of forwards

The distributed load balancer implemented in each node has an incomplete view of the load information in the rest of nodes because load updates are not instantaneous, and the update messages may be lost because a non-critical broadcast message is typically implemented over UDP. This can create a hazard: in a short period of time, nodes may forward queries to the same subset of nodes and overload them. Our solution is to allow the receiving node to decide if there is a better choice once the query is received and forward again the query. This procedure can be extended recursively until a good candidate is found or a maximum number of forwards is reached.

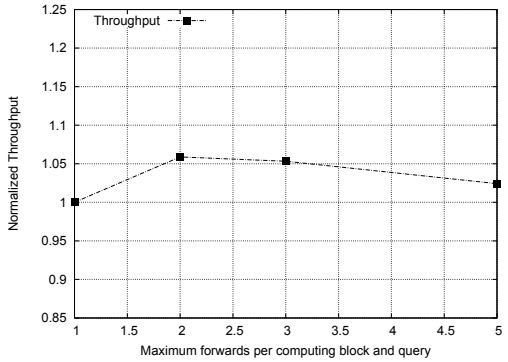


Figure 6. Analysis of the number query forwards for question answering. System throughput vs. maximum number of forwards per query.

We took the best system configuration from previous experiments and varied the number of query forwards. In Figure 6, we plot the system throughput against the maximum number of forwards per query. For example, a value of 2 means that for each computing block we allowed up to two forwards. In order to show a clearer plot, we averaged the results shown in Figure 6 for the different configurations because they show a very similar behavior.

Our results indicate that the optimal number of forwards is two, which is roughly 5% faster than the original system. A larger number of forwards does not improve the load balancing, and also produce more network traffic. We computed a one way ANOVA (analysis of variance) to compare the different configurations. The test indicates that the difference was significant between one and two forwards (but not for two and three). Even though we found that the stated overloading hazard exists in a search engine, the performance penalty is not huge. The second forward improves performance at the expense of more network traffic, which in large systems may not be desirable.

4. Related Work

Search engines have become ubiquitous for many daily tasks. These systems are formed by clusters of computers [21] that are executing many queries simultaneously. In order to support such large scale search engines, the data locality and the cooperation among the computers is fundamental. Two different approaches can be used to improve data locality: move the queries to a node with the cached contents or use cooperative caching.

Cooperative caching appeared in the mid-90s to create large distributed caches, which it looks like a traditional local cache from any of the computing node [22], [23]. Since then, many high performance applications have included cooperative caching to improve its performance: distributed file systems [24], [25], distributed web servers [26], [27], applications on ad hoc networks [28], database engines [29] or question answering [6].

Distributed systems include different strategies for load balancing depending on the workload of the system [30]. In this work, we focus on workloads where caching plays an important role in the system performance. These systems evolved from the use of heuristics such in [31], where the load balancer tries to send a certain query to the same set of nodes to improve locality, to more complex algorithms which consider also previous logs [32], the current disk load [33] or the incoming query distribution [34].

However, these two families of techniques have been applied individually. To our knowledge there is no analysis of how they interact when both are implemented in a system simultaneously. Bunt et al. [35] performed a similar study where they performed simulations to study the improvement of a load balancer that takes into account cache contents in each node. However, that paper did not consider the effect of cooperative caching. Andrade et al. also studied a distributed system where a caching service compound by multiple servers is available to a pool of application servers [36]. In the database community, we find the MOCHA system, which implements a client-server query analyzer that detects if a query contains operations that reduce or add size to the tuples of a result set. And hence, decides if the query is going to execute faster in the database server, or if it is preferable to transfer the data to the client [37]. In this paper, we have taken state-of-the-art algorithms for cooperative caching and load balancing for question answering [6], [11], implemented them in a fully-fledged question answering system, and measured their combined performance with empirical experiments backed by statistical models.

5. Conclusions

The performance of a search engine is very related to its in-memory data management. This paper presents an statistical analysis of the performance of a fully fledged distributed question answering system for different query distributions typically found in web query logs. We compare two different approaches to improve the system performance: (a) cooperative caching, i.e, send the cached contents from a node to the node that is currently computing a query; (b) cache-aware load balancing, i.e. send a query to the node whose cached contents are related and its computing load is small. To our knowledge, this work is the first study that analyzes and quantifies which data management policy is preferable.

On the one hand, cooperative caching creates the illusion that computers share a large virtual cache pool created by the merging of the available memory in each computing node. In our tests, the speed up of this approach was up to 1.46 and always over 1.32, which is significantly better than the original system. The source of the improvement

is a better hit rate, which surpasses the local policy by more than 25 percentile points. On the other hand, cache-aware load balancing sends queries to nodes with similar cache contents. Our results show that both policies achieve a similar improvement of hit rates, but the throughput of cache-aware load balancing is 1.77 times higher due to reduced network traffic. Queries are smaller than data contents and only need a single connection, whereas the document size is larger and may require contacting many nodes. Nevertheless, each technique copes with cache management from complementary perspectives and the throughput of the combined system is up to 1.90. Our statistical model proves that although the absence of caching and load balancing penalizes severely the system throughput, both techniques do not collide when they are simultaneously enabled and they can be activated simultaneously for a better performance.

Moreover, we statistically study some design decisions that influence the system performance. We noticed that the number of scheduling points in a question answering system is relevant, and it is not statistically significant to include more than two: one for PR and other for AE. More scheduling points do not deteriorate the throughput but do not provide additional benefit. We also realized that the delay between load updates may create small unbalances and it is advisable to let the system forward each query twice for a balanced computation.

References

[1] D. Roussinov, W. Fan, and J. Robles-Flores, "Beyond keywords: automated question answering on the web," *Commun. ACM*, vol. 51, no. 9, pp. 60–65, 2008.

[2] Y. Xie and D. R. O'Hallaron, "Locality in search engine queries and its implications for caching," in *INFOCOM*, 2002.

[3] D. Dominguez-Sal, J. Larriba-Pey, and M. Surdeanu, "A multi-layer collaborative cache for question answering," in *Euro-Par*, 2007, pp. 295–306.

[4] M. Surdeanu, D. Dominguez-Sal, and P. Comas, "Design and performance analysis of a factoid question answering system for spontaneous speech transcriptions," *Interspeech*, 2006.

[5] J. Callan, "Distributed information retrieval," *Advances in Information Retrieval*, pp. 127–150, 2000.

[6] D. Dominguez-Sal, J. Aguilar-Saborit, M. Surdeanu, and J. Larriba-Pey, "On the use of evolutive summary counters in distributed retrieval systems," *Technical report.UPC-DAC-RR-DAMA-2008-1*, 2008.

[7] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE Trans on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[8] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Comm. of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[9] D. Puppin, F. Silvestri, and D. Laforenza, "Query-driven document partitioning and collection selection," in *Infoscale*, 2006, p. 34.

[10] D. Wessels and K. Claffy, "Internet cache protocol: protocol specification, version 2," *RFC 2186*, 1997.

[11] D. Dominguez-Sal, M. Surdeanu, J. Aguilar-Saborit, and J.-L. Larriba-Pey, "Cache-aware load balancing for question answering," in *CIKM*, 2008, pp. 1271–1280.

[12] M. Surdeanu, D. Moldovan, and S. Harabagiu, "Performance analysis of a distributed question/answering system," *TPDS*, vol. 13, no. 6, pp. 579–596, 2002.

[13] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *IPM*, vol. 24, no. 5, pp. 513–523, 1988.

[14] NIST, "TREC question answering track," *http://trec.nist.gov/*, 1999-2007.

[15] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Riberio-Neto, "Rank-preserving two-level caching for scalable search engines," *ACM SIGIR*, pp. 51–58, 2001.

[16] R. Baeza-Yates, "Web usage mining in search engines," in *Web Mining: Applications and Techniques*, A. Scime, Ed. Idea Group, 2005, pp. 307–321.

[17] E. Markatos, "On caching search engine query results." *Computer Communications*, vol. 24, no. 2, pp. 137–143, 2001.

[18] B. S. Everitt, *The Cambridge Dictionary of Statistics*, 3rd ed. Cambridge University Press, 2006.

[19] D. Mongomery, *Design and Analysis of Experiments*, 5th ed. Wiley, 2000.

[20] SPSS Inc., "SPSS version 17.0," 2008.

[21] L. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[22] M. Dahlin, R. Wang, T. Anderson, and D. Patterson, "Cooperative caching: Using remote client memory to improve file system performance." in *OSDI*, 1994, pp. 267–280.

[23] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath, "Implementing global memory management in a workstation cluster," in *SOSP*, 1995, pp. 201–212.

[24] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang, "Serverless network file systems," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 41–79, 1996.

[25] S. Annapureddy, M. J. Freedman, and D. Mazières, "Shark: Scaling file servers via cooperative caching," in *NSDI*. USENIX, 2005.

[26] M. Raunak, "A survey of cooperative caching," *Technical report*, 1999. [Online]. Available: http://citeseer.ist.psu.edu/raunak99survey.html

[27] V. Holmedahl, B. Smith, and T. Yang, "Cooperative caching of dynamic content on a distributed web server," in *HPDC*, 1998, p. 243.

[28] L. Yin and G. Cao, "Supporting cooperative caching in ad hoc networks," *IEEE Trans. Mob. Comput.*, vol. 5, no. 1, pp. 77–89, 2006.

[29] K. Lillis and E. Pitoura, "Cooperative xpath caching," in *SIGMOD*, 2008, pp. 327–338.

[30] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu, "The state of the art in locally distributed web-server systems," *ACM Comp. Surveys*, vol. 34, no. 2, pp. 263–311, 2002.

[31] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," *ACM SIGPLAN Notices*, vol. 33, no. 11, pp. 205–216, 1998.

[32] L. Cherkasova and M. Karlsson, "Scalable web server cluster design with workload-aware requestdistribution strategy WARD," *WECWIS 2001*, pp. 212–221, 2001.

[33] X. Qin, H. Jiang, Y. Zhu, and D. Swanson, "Dynamic load balancing for I/O-intensive tasks on heterogeneous clusters," *HiPC 2003*, pp. 300–309, 2003.

[34] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *TPDS*, vol. 16, no. 3, pp. 219–233, 2005.

[35] R. B. Bunt, D. L. Eager, G. M. Oster, and C. L. Williamson, "Achieving load balance and effective caching in clustered web servers," in *Int. Web Caching Workshop*, 1999.

[36] H. Andrade, T. Kurç, A. Sussman, and J. Saltz, "Optimizing the execution of multiple data analysis queries on parallel and distributed environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 520–532, 2004.

[37] M. Rodriguez-Martinez and N. Roussopoulos, "Mocha: A self-extensible database middleware system for distributed data sources," in *SIGMOD*, 2000, pp. 213–224.