Peer-to-Peer Netw Appl (2010) 3:83–99
DOI 10.1007/s12083-009-0047-5

SPACE: A lightweight collaborative caching for clusters

Mursalin Akon · Towhidul Islam · Xuemin Shen · Ajit Singh

Received: 24 September 2008 / Accepted: 22 April 2009 / Published online: 13 May 2009

Abstract In this paper, we introduce *S*ystematic *P*2P *A*ided *C*ache *E*nhancement or *SPACE*, a new collaboration scheme among clients in a computer cluster of a high performance computing facility to share their caches with each other. The collaboration is achieved in a distributed manner, and is designed based on peer-to-peer computing model. The objective is to provide (1) a decentralized solution, and (2) a near optimal performance with reasonably low overhead. Simulation results are given to demonstrate the performance of the proposed scheme. In addition, the results show that SPACE evenly distributes work loads among participators, and entirely eliminates any requirement of a central cache manager.

Keywords Cluster computing · Caching · Collaborative computing · Filesystem

1 Introduction

In recent years, computer memory and mass storage devices are becoming large and inexpensive. However, the data access time for storage devices has not been improved proportionally as compared with processor and computer memory. Hence, throughput of a data processing application is dominated by the performance of the data provider, and in most of the cases, those mass storage devices play this significant role. Caching is commonly used to improve the performance of mass storage devices by reducing access latency [21, 28, 35]. In a network environment, a cache eliminates frequent expensive storage device access and increases the efficiency in terms of service time and throughput [15]. However, to serve a pool of clients efficiently, the cache needs to be large enough. A way around is to distribute the cache among the clients. The smaller client caches can now serve many data requests locally, without involving the central file server. This not only helps in alleviating the server load but also reduces the network traffic.

Today's high speed and wider bandwidth communication hardware enables transfer of high volume of data among clients in a very short period. Due to this effluent technology, the possibility of coordination among the client caches emerges. The coordination can serve a requested block which is not available at the local cache but is present in the cache of a remote client. The collaboration mimics the integration of all the client caches together into a large cache, and can offset the drawback of the storage devices up to some extent. However, coordination in a network environment is abstruse, especially when response time is critical. Besides, efficient search mechanism is required to locate a cache block at a remote client. The simplest solution is to have a central database of references mapping the storage blocks to the client-side cache blocks.

In this research, we propose a distributed collaborative caching scheme for clusters (or for closely coupled

M. Akon · T. Islam · X. Shen (✉) · A. Singh
Department of ECE, University of Waterloo,
Waterloo, Ontario, Canada
e-mail: xshen@bbcr.uwaterloo.ca

M. Akon
e-mail: mmakon@ece.uwaterloo.ca

T. Islam
e-mail: mtislam@ece.uwaterloo.ca

A. Singh
e-mail: asingh@ece.uwaterloo.ca

network workstations), namely *SPACE* (*S*ystematic *P*2P *A*ided *C*ache *E*nhancement). Here, the clients create an environment which gives the perception of a large *pseudo global cache*. To accomplish this goal, the clients exchange information about their local caches through gossip messages. The disseminated information is used to unify the smaller client caches into the pseudo global cache. If a request can not be served from the local cache, it is looked up in the rest of the pseudo global cache without involving the file server or any central manager. When a locally unavailable block is found in the pseudo global cache, the clients cooperate in fetching the block. Moreover, through the coordination, the performance of a busy cache can be improved by properly utilizing a nearby idle cache. Here, an idle cache not only helps the busy cache in fetching blocks, but also preserves critical blocks to reduce the cost of retrieval from the mass storage. In addition, we propose that the clients introduce replicas of frequently accessed blocks. Replication of such blocks often reduces the bottleneck of the central server, distributes the service load among clients, and increases the chance of hits in the local as well as in the pseudo global cache. However, when the system gets busy, the clients coordinate in an elimination process to remove one or more replicas, and make space for newly introduced blocks.

Due to the collaboration, a client acts as a requester for services from other clients and at the same time, acts like a service point for other clients. As a result, the load of the system is distributed among the participators. In this scheme, the data server gets a service request, only if the request can not be served by the pseudo global cache, i.e., a miss happens in both the local and the global cache. To achieve this coordination, we introduce a peer-to-peer (P2P) client partnership to model the collaborative cache. In this partnership, the client-server relation becomes dubious, and cooperation among peers (i.e., clients of the file server) emerges to provide higher number of hits in the pseudo global cache. With this approach, we obtain three additional fundamental benefits: (1) low maintenance cost, (2) easy integration with existing software platforms, and (3) easy development platform. Note that, unlike the research on distributed P2P storage systems, such as CAN [32], in this research, we do not consider distribution of storage and concentrate on collaborated caching system.

Beside introducing our scheme, we also compare the proposed collaborated caching with an ideal Global LRU policy and investigate the practicality of the approach. Our simulation results show that this proposed scheme reasonably approximates the ideal Global LRU caching policy which has the instantaneous view of all the caches in the system. The results also show that the scheme performs better than existing centralized solutions. Additionally, the results demonstrate that the message communication and memory overhead for the maintenance operations are fairly low.

The reminder of the paper is organized as follows. In Section 2, the related works with some comparisons with the proposed caching scheme are briefly discussed. The system model is presented in Section 3. Each component of SPACE is elaborated in Section 4. In Section 5, we evaluate SPACE and explain simulation results. Finally, in Section 6, we conclude the paper with a discussion of possible improvements and future works.

2 Related works

We first explicitly distinguish our work by presenting research domains with keywords similar to those of collaborative caching. Significant research works have already been published on Web caching where the goal is to gracefully handle denial of service due to the bandwidth limitation between a server and a client. These works mainly concentrate on object location determination [8, 41, 46] and efficient lookup procedure [16, 34]. In [27], similar issue is addressed using P2P networks. In contrary, our research explores cooperation between caches along with object location determination and lookup mechanism. Research in caching of objects to facilitate multimedia on demand has received much attention [22]. P2P computing has also been considered to achieve efficient and low-cost solutions [12, 13]. To facilitate multimedia streaming, each peer caches few segments around its play offset. Then, peers collaborate to arrange themselves in a pipeline such that cached contents of a peer are useful to the next one. Our research is completely different where peers do not have such a linear relation. In other researches, distributed organization is used to enable extraordinarily large file system [17, 23, 33, 38] that are beyond the capacity of the most advance storage media. These researches include controlled placement of files, efficient fetching and data consistency. Some of these distributed file system projects exploits client caches, but none of them are collaborative. However, collaborative caching can increase the throughput significantly, as shown in [25]. In [26], P2P networks are used to locate the peer caching the smallest superset of a database to answer a specific range query. In [39] and [42], alternates to Distributed Hash Table (DHT) are proposed to cache indices among a group of peers

in a distributed manner. These techniques enable fast and efficient query processing in structured P2P networks. Our target platform is a more tightly coupled environment where a cache manages objects of same size (of disk block) within a very limited cache space without explicit control over placement. Moreover, the demand of faster response time prohibits the caches to employ an extra level of redirection through indexing or to employ a rigorous and highly intelligent module, as opposed to the Web and multimedia object caching.

The idea of distributing a huge server cache among cooperative clients was first proposed in [15]. Here, the server maintains a central database on precise information about the cache blocks located at different clients. A request to locate a block is forwarded to the server, if a local miss occurs. The server, in turn, redirects the request to the client caching the block. If such a client does not exist, the server serves the block by reading from the storage. A client can evict or replace a block only after consulting the server. In this scheme, a single copy of a block is allowed to relocate from a cache to another at most N times, before removing it entirely (hence, the name N-chance). In [36, 37], a hint based cooperative caching is proposed, where each client maintains references to special copies of all the blocks. The algorithm works reasonably well when the combined distributed cache is significantly smaller. In [14], a cooperative caching scheme for clusters is proposed. However, the scheme assumes that an instant view of all the collaborative caches is available to all the cluster members. As a result, the proposed procedures are not applicable in practice without having a central manager. In [25], a hierarchical cooperative caching scheme, called Hierarchical GreedyDual, is proposed. In this scheme, a coordinated placement and a replacement algorithms are described. A network of client caches is divided into clusters of caches. The representatives of the clusters form a higher level clusters and so on, and results in a hierarchical cluster tree. The representative helps in coordinating the caches of the cluster members. Object placement, look up and replacement take place in hierarchical manner starting from the bottom of the hierarchy. The proposed algorithms depends on availability of object access frequencies. However, in practice, this kind of information is rarely available. As a solution, it was proposed to predict the future access frequencies based on the past experiences. Maintaining all the frequencies are impractical in different situations, particularly when the number of objects are extremely large (for example, block based file caching). At the same time, the work load is high for a cluster representative and is higher for the representatives as the cluster hierarchy is climbed. Unlike the previous works, SPACE is a completely distributed scheme. It eliminates the need for a central database hosting locations of cache blocks. Each peer disseminates its local cache information using a gossip protocol. A peers chooses a collaborator based on the local information only.

3 System model

In this section, the system model for the proposed scheme is described. Then two examples are presented to illustrate the use of the scheme.

3.1 System model

The discussion on system model is divided into two parts: (1) the hardware model, and (2) the software model.

Hardware model Figure 1 shows the hardware part of the system model. The model reflects the hardware orientation of a typical computer cluster. Here, the master and all the slave nodes are connected through a high bandwidth, low latency network. Slave nodes may be equipped with local storages, but they are used for virtual memory and as storage of temporary files. Application data are served from a high speed disk (such as, RAID [30]), attached to the master node. Jobs are posted in the system through either terminal or remote log-in at the master node. It should be noted that in

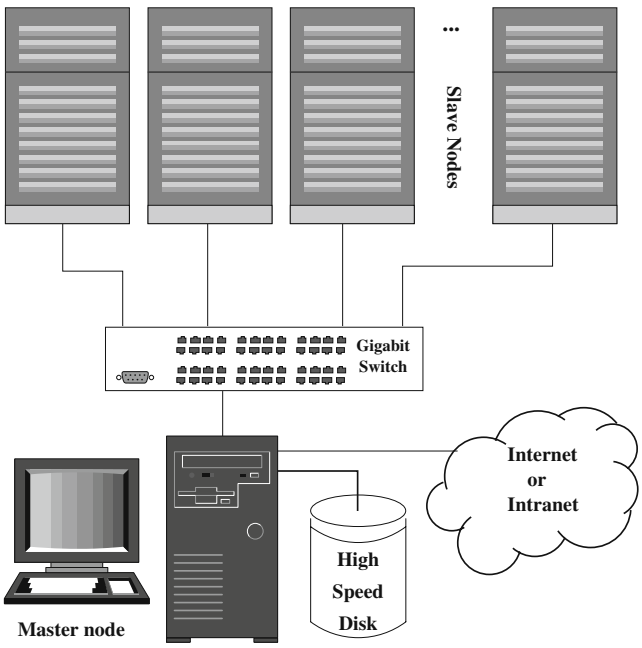


Fig. 1 A typical computer cluster

this model, one (parallel) application consists of one or more processes running on different master / slave nodes, and a job often consists of one or more such applications. The Abscus system [44] at the University of Waterloo is an excellent example of this model. It is a recent addition to the High Performance Computing (HPC) facilities to expedite computational chemistry. The typical communication networks used in today's clusters are much (often, more than ten fold) faster than the data access time of a storage device. This is the main thrust behind the idea of collaborative caching—it is considerably faster to retrieve data from the memory at a remote machine than retrieving the data from (as in this case, from remote) mass storage devices.

Software model We assume that a data server is executing on the master node. This allows the data server to access the data from the storage directly. The sole purpose of the data server is to respond to data requests from the clients by reading the data from the high speed storage. A client is defined as a collection of application processes running on a computer of the cluster. As a result, the slave nodes as well as the master node can act as clients.

In this model, a local cache is maintained by each of the clients. To have a collaborative caching, the clients at first form a peer-to-peer network. In this network, a client acts as a peer of the other clients.[1] The peers form a pseudo global cache by sharing information about their respective caches.

3.2 Illustrations

Consider an automated face recognition system, installed at an important facility. The system is shown pictorially in Fig. 2. Here, the cameras supply continuous or periodic images taken at different intersections of the facility. The face recognition system identifies persons of interest from the images. As the first step of the process, the captured images are analyzed and faces are detected (denoted as *Face Detection* in the figure). From the detected faces, different features are extracted (in the *Feature Extraction* phase). Then different face recognition algorithms can be applied to compare the features of faces from the captured images with those of the persons of interest. The information about persons of interest along with their face attributes are stored in the attached high speed storage device.

[1] In the rest of the paper, we use the terms client and peer interchangeably.

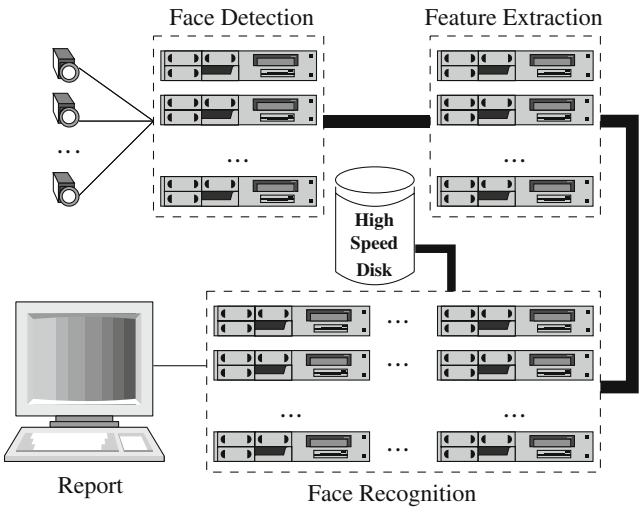


Fig. 2 A face recognition system at an airport

Till today, there exists no perfect face recognition algorithm. Hence, a consensus is made from the outputs of all the algorithms running on the cluster, and is presented via the monitoring device as a suggestion to the human operator.

Another example is an On-Line Analytical Processing (OLAP) application. This application provides real-time answers to the queries made by the Decision Support System (DSS) of an enterprize. To achieve this goal, an OLAP application pre-computes different multi-dimensional views of the enterprize data. Recent research efforts demonstrate that parallel computation of the views is the most effective solution [6, 11, 19].

During our previous researches on parallel computation for clusters [2–6], we have found that data retrieval time is still the most critical factor for the applications described above. The retrieval time determines how fast the face recognition system can work or how frequently the OLAP application can revise a view from the dynamic raw data. A collaborative caching scheme helps this kind of applications to become interactive and to act in realtime.

4 Collaborative caching scheme

In this section, we reveal different components of the proposed collaborative caching scheme, SPACE. To facilitate operations of SPACE, each peers manages several related data structures. Those data structures encapsulate the states of the local and remote caches. The states of remote caches are gathered from the gossip messages broadcasted in the P2P network. Other

maintenance operations are required to ensure consistency of this data structures.

For the sake of discussion, this section is divided into several subsections. At first, the organization of a cache is described. Then, other essential operations of the scheme such as cache lookup procedure, cache block eviction procedure, cache block placement procedure in the pseudo global cache, etc are presented. The discussion about the management of peers in the P2P network is out of the scope of this paper and hence is omitted.

4.1 The cache organization

In SPACE, the pseudo global cache, formed by combining all the caches of the participating peers, is perceived by the running applications only. In contrary, as a lower level service provider, a peer distinguishes between the local cache and the remote caches. A peer p maintains a cache memory of size M_p where the size of each cache block is m. The number of cache block available at the peer is denoted as $n_p = \lfloor M_p/m \rfloor$. The total disk size is D where the size of each disk block is d. The number of disk blocks available in the disk is $n = \lfloor D/d \rfloor$. Each cache block can store c consecutive disk blocks where c is a positive integer, i.e., $c \in \mathbb{Z}^+$ and $c \times d = m$.

A peer maintains a set, called Local Cache Information (LCI), to refer to the local cache. The set LCI includes an entry for each of the locally cached disk blocks. An entry x of LCI (denoted as $LCI(x)$) consists of the tuple $< a_x, clk_x, ob_x, RCB_x, ptr_x >$. Here, a_x is the disk address that the first block $LCI(x)$ is holding. The clk_x maintains the clock tick since the last reference of the cache block. The ob_x element is a simple flag and indicates the originality of a cache block. When a cache block is retrieved from the server, it is marked as *original* and replicas of an original cache block are designated as non-original. Information about remote cache blocks, having the same content, are maintained in RCB_x (detailed discussion follows). The last tuple element ptr_x holds the reference to the local cache memory where the actual disk data is physically cached.

Similar to LCI, another set, called Remote Cache Information (RCI), is maintained to refer to the cache blocks at the remote peers. An entry x of RCI (denoted as $RCI(x)$) consists of the tuple $< a_x, clk_x, RCB_x >$. The a_x and RCB_x elements have same semantics of similar elements in an LCI entry. But, unlike clk_x in $LCI(x)$, clk_x in $RCI(x)$ serves the purpose of a reference clock counter. The actual time since the last access to a block at a remote peer is computed relative to this counter.

RCB_x (Remote Cache Block) holds a set of entries for all peers caching the block with address a_x. An entry y in RCB_x (denoted as $RCB_x(y)$) consists of the tuple $< rclk_y, ob_y, pid_y >$. pid_y is the identification of the remote peer that caches the same block and ob_y indicates the status of originality of that block. The element $rclk_y$ is a relative clock tick that is used to compute the reported last access of the remote block at the peer pid_y. The computation of the last access time of a block at a remote peer is determined by the function $fclk(rclk_y, clk_x)$, where $rclk_y$ is in $RCB_x(y)$, and RCB_x is in either $LCI(x)$ or $RCI(x)$. Though $fclk(rclk_y, clk_x)$ can be implemented in different ways, a simple solution would be as follow:

$$fclk(rclk_y, clk_x) = rclk_y + clk_x \tag{1}$$

It should be noted that both LCI and RCI can not have entries for the same disk blocks. Formally, if $x \in LCI$, there must be no $y \in RCI$ such that a_x and a_y are the same.

Beside maintaining cache references, a peer also maintains some statistics about other peers (PS). An entry z of PS (designated as $PS(z)$) contains: identification of the remote peer (pid_z), the maximum number of cache blocks the peer can have (n_z), the number of original cache blocks maintained by the peer (no_z), the frequency of data read requests received by the peer from the local processes (λ_z), and the local miss rate (f_z).

A peer periodically broadcasts a digest about the local cache with an interval of t. The digest contains information about the local cache blocks along with some other statistics. A peer uses information from the digest to populate the PS structure. On receiving a digest broadcast from peer p, a peer recomputes the RCB sets. It also updates the associated peer statistics, i.e., $PS(z)$ where $pid_z = pid$.

At each read request from the executing processes, the corresponding peer decrements the clocks associated with all the LCI and RCI entries, i.e., all clk_x and clk_y where $x \in LCI$ and $y \in RCI$. If a local cache hit takes place at $x \in LCI$, clk_x is exempted from this decrement operation. In contrary, the clock is upgraded with the maximum allowable value. Note that different peers may face different rate of read requests. As a result, the clocks at different peers count down at different speed. To be consistent, each peer computes the highest read request frequency among all the peers from the locally available PS. To match with the peer with the most frequent read requests, a peer inserts dummy read requests between actual read requests.

As no cache hit occurs during a dummy read, all the clocks in the cache data structures are decremented by one. The clock counters of *RCB*s are never required to be decremented, as they are computed relative to the corresponding LCI or RCI clocks using the $fclk$ function. While computing frequency of read request at a peer, the dummy reads are excluded and only real read operations are counted.

We denote a local cache block $x \in LCI$ to be *alive*, when clk_x counts within a system-wide allowable limit. When the clock count goes below this limit, the block is considered to be *dead*. However, a dead block is not removed from the cache immediately. The detailed eviction procedure will be discussed later, in Section 4.3. For original blocks, the allowable time limit to be alive (without any reference) is γ times of the allowable time limit of a non-original block, where $\gamma \geqslant 1$. In our scheme, all dead blocks are considered non-original blocks, i.e., when an original block becomes dead, it is also marked as non-original. This ensures that no dead block gets the preference of an original block. If necessary, those dead blocks are considered to be the best candidate for evictions, and are never included in the digest to broadcast.

4.2 Lookup procedure

The control flow of the lookup procedure is presented in Algorithm 1. When a read request arrives to a peer for the block with address *da*, at first, the peer searches for that requested block in the local cache (line 1). If the search is successful, the block is served immediately. Otherwise, the peer searches for other peers who potentially have the requested block (line $4-17$). A *FETCH_REQUEST* message is sent to such a peer to retrieve the block. Due to the fact that the content of a cache may be changing at each read request, a fetch request may not always be satisfied. Therefore, in reply to a *FETCH_REQUEST*, a *FETCH_RESPONSE* message indicates either a success along with the

Algorithm 1: Control flow of the lookup procedure

```
   input  : da is the address of a data block at storage
   output: address of the data block in local memory
 1 Search for an x ∈ LCI, such that a_x = da
 2 if x = null then
       //the block is not cached locally
 3     found ← false
 4     Search for an y ∈ RCI, such that a_y = da
 5     if y ≠ null then
           //the block is possibly available at a remote cache
 6         Z ← ∅
 7         for i ← 1 to η do
 8             Find peer z ∈ RCB_y such that util(pid_z) ≥ util(pid_a) where a ∈ RCB_y − Z
 9             Sent FETCH_REQUEST to pid_z with a request to fetch the block
10             if FETCH_RESPONSE indicates a success then
                   //data is fetched from remote peer
11                 data ← extract data block from FETCH_RESPONSE
12                 found ← true
13                 break
14             end
15             Z ← Z ∪ {z}
16         end
17     end
       //data is not available in pseudo global cache -> get it from data server
18     if found = false then
19         Sent READ_REQUEST to data server with a request to fetch the block
20         data ← extract data block from READ_RESPONSE
21     end
       //search for an empty space to save the block locally
22     Search for an x ∈ LCI, such that a_x = null
23     if x = null then
24         x ← Evict()
25     end
26     Store data at x
27 end
28 return ptr_x
```

requested block or a failure. If these two steps fail, a *READ_REQUEST* message is sent to the data server to read the block from the storage device (line 18 – 21). The server then reads the block and returns the data within a *READ_RESPONSE* message.

While forwarding a block in *FETCH_RESPONSE*, a peer always tags the forwarding copy with the tag of the local copy and remarks the local copy as non-original. So, if the local copy is an original block, forwarded copy is tagged as original. This policy of forwarding an original block has two positive effects. Firstly, the original block gets a refreshed clock at the requesting peer, and as a result, the chance that it will be in the pseudo global cache for longer time becomes higher. Secondly, an idle peer gets rid of the original blocks at a faster pace which helps in having more free cache blocks in the pseudo global cache (see Sections 4.3 and 4.4 for further discussion).

$$util(pid_z) = (clk_{max} - clk_{min}) \times \left(1 - \min\left(1, \frac{\ln(fclk(rclk_z, clk_y) - clk_{min} + 1)}{\ln(clk_{max} - clk_{min} + 1)}\right)\right) \quad (2)$$

To find the most favorable remote collaborator (at line 8), utility of all other peers reported to store *da* block is computed. The peer with the maximum utility is considered to be the most favorable peer. For a specific block, the utility of a peer is considered as an increasing function of the (reported) clock count. In other words, it is a non-increasing function of the time elapsed since the block was referred last. As the clock count decrements, the chance that the associated block will be evicted (or will be dead) increases. So, the probability of existence of a block with lower clock count is less than a block with a fresher clock. Finally, the utility of a peer pid_z, for a $z \in RCB_y$, is computed according to Eq. 2. When a block is fetched from a remote peer or from the server, the block is cached locally along with a refreshed clock count of clk_{max}. When the count decrements to clk_{min} an alive block becomes dead. Hence, $(clk_{max} - clk_{min})$ is the maximum clock ticks a fresh block would remain alive without any reference. As $(clk_{max} - clk_{min})$ is larger for an original block (indicated through ob_z flag), a peer hosting an original block gets comparatively higher preference. Moreover, the utility function is such that it makes a peer less appealing at a faster pace as the associated clock count decrements linearly.

If the cache is full, one existing block is evicted to make space for the new block (line 24). This procedure will be discussed further in the next subsection.

4.3 Eviction procedure

The eviction procedure (the *Evict* operation) is illustrated in Algorithm 2. When eviction of a block is necessary, SPACE evicts a dead block without any further treat (line 1). Other than that, the oldest non-original and then original block are evicted (line 3 – 5). Note that, in practice the searches of line 1, 3 and 5 can be performed in a single round. SPACE gives preference towards preserving original blocks by evicting non-original blocks. In this way, the P2P cache benefits in two ways: (1) shortage of cache blocks is handled by removing the duplicates, and (2) every effort is made to keep at least one copy of a block, i.e., the original block. The last option to make space for a new block is to evict the oldest original block. However, before evicting such a block, the peer marks the block as *sticky* and keeps it for some more time to make further efforts to preserve it. A series of ζ attempts are made to mark a copy of a sticky block at a remote peer as original (line 8 – 19). If all the attempts fail, a final attempt is made to place the sticky block in one of the idle caches (line 20 – 22, see next subsection for details). Finally, the sticky block is permanently removed from the local cache.

Like a dead block, a sticky block is not included in the periodic digest broadcast but is used in serving read requests. The reason behind is that a sticky block will soon be phased out and the chance that a remote peer will be served by this block is very narrow.

4.4 Placement in the remote cache

To move and place a sticky block at one of the remote caches, a peer has to make careful decision, as sending a whole block is more expensive than sending simple request messages such as *MAKE_ORG_REQUEST*. The relocation operation is presented in Algorithm 3. At first estimated number of free blocks at each peer is computed according to Eq. 3 (line 1 – 3) and then a placement request is sent to peer z with a probability, proportional to the number of the estimated free blocks (line 4 – 7).

$$\max(0, n_z - f_z \times \lambda \times t_z) \quad (3)$$

where n_z is the reported number of free blocks. Parameters f_z, λ, and t_z are the miss rate, average read request arrival rate and the time epoch since pid_z reported its last status, respectively. All these parameters are available through peer status records, i.e., $PS(z)$.

After computing the potential collaborator, the peer sends over an asynchronous *STORE_REQUEST* message (line 7) and drops the sticky block immediately. On receiving the *STORE_REQUEST* message, the

Algorithm 2: The *Evict* operation

```
   output: address of a cache block is free
   //search for a dead block
 1 Search for an x ∈ LCI, such that clk_x < clk_min
 2 if x = null then
       //no dead block is found -> evict the oldest non-original block
 3     Search for an x ∈ LCI, such that clk_x ⩽ clk_y ∧ ob_x ≠ original; where y ∈ LCI ∧ x ≠ y
 4     if x = null then
           //no non-original block is found -> evict the oldest original block
 5         Search for an x ∈ LCI, such that clk_x ⩽ clk_y; where y ∈ LCI ∧ x ≠ y
 6         Mark x as sticky
           //try to mark another block in the pseudo global cache as original
 7         marking ← false
 8         Z ← ∅
 9         for i ← 1 to ζ do
10             Find peer z ∈ RCB_x such that util(pid_z) ≥ util(pid_a) where a ∈ RCB_x − Z
11             Sent MARK_ORG_REQUEST to pid_z with a request to mark the d_x block original
12             if MARK_ORG_RESPONSE indicates a success then
                   //the marking attempt is a success
13                 marking ← true
14                 break
15             end
16             Z ← Z ∪ {z}
17         end
18         if marking = false then
               //all the marking attempts failed -> try to relocate the data
19             Relocate(x)
20         end
21         Remove x from the cache
22         Move the record for x from LCI to RCI
23     end
24 end
25 return x
```

remote peer stores the accompanying block, if and only if it hosts an unused or a dead block in its local cache.

4.5 Duplicate originals

It is possible for more than one peers to have original copies of the same block. Duplicate original blocks are not desirable, as they make it difficult to get free cache blocks in a busy system. Having multiple original copies of the same block does not mandate any change to the procedure of fetching a block from a remote peer. However, phasing out a sticky block assumes existence of no duplicate original blocks. This observation leads us to an elegant distributed solution to the problem.

When a peer fetches an original block from the server, it includes that information in the next digest broadcast. After receiving such a digest, other peers are not going to get another original copy of the same block, because if needed, other peers can get copies from the peer stocking the original one. Besides, retrieving an original block from the server is more expensive than getting a copy from the peers.

In our duplicate resolving process, if a digest from a preceding peer[2] asserts about a duplicate original block, a peer simply marks its own original copy as non-original. On the other hand, if the digest is received from a non-preceding peer, the recipient updates the clock tick of the of the local copy with the maximum of that's of the duplicates. We propose the following theorem.

Algorithm 3: The *Relocate* operation

```
   input: x is the reference to local cache
 1 foreach peer z do
 2     n'_z ← Expected number of free blocks at z
 3 end
 4 foreach peer z do
 5     p_z^free ← n'_z / Σ_q n'_q
 6 end
 7 Send x to a peer z with the probability p_z^free
```

[2]Precedence is measured according to an identification which may be the peer ID, IP address, etc.

Theorem 1 *With the proposed duplicate resolving process all original duplicates are resolved within two gossip cycles after the first copy is fetched.*

Proof We demonstrate the proof with two peers—A and B; the proof with more than two peers is a simple extension. Let $A \prec B$, as A has an identification which precedes (possibly, lexicographically or numerically) the identification of B. Let both the peers cache original copy of the same block. Such a scenario happens if and only if both A and B report in their digest about not caching a specific block and then both retrieve the block from the data server before their next digest broadcast. Now, one of the following situations can arise before resolving duplicates.[3]

Case 1 $g(B) \mapsto f(B) \mapsto g(A) \mapsto f(A) \mapsto g(B) \mapsto g(A)$
Case 2 $g(B) \mapsto g(A) \mapsto f(A) \mapsto f(B) \mapsto g(B) \mapsto g(A)$
Case 3 $g(B) \mapsto g(A) \mapsto f(B) \mapsto f(A) \mapsto g(B) \mapsto g(A)$
Case 4 $g(A) \mapsto f(A) \mapsto g(B) \mapsto f(B) \mapsto g(A)$
Case 5 $g(A) \mapsto g(B) \mapsto f(A) \mapsto f(B) \mapsto g(A)$
Case 6 $g(A) \mapsto g(B) \mapsto f(B) \mapsto f(A) \mapsto g(A)$

In case 1, duplicate original blocks are introduced, if $f(B)$ and $f(A)$ of 2nd and 4th event fetch the same data block. However, duplicates are resolved as both B and A report their digest, i.e., 2nd instance of $g(B)$ and $g(A)$ occur. As observed, the resolve process is limited by $g(B) \mapsto g(B) \mapsto g(B)$ events, i.e., two gossip cycles. Similarly, other cases can be explained.

Hence, in all cases duplicates are resolved within two gossip cycles. □

Corollary 1 *After one gossip cycle of fetching an original block, no other original copy of the same block is fetched and after two gossip cycles only one copy of an original may exist in the entire pseudo global cache system.*

4.6 Bloom filter-based software solution

A direct translation of the idea presented in the previous subsections to an implementation yield to be efficient for a small scale cluster (refer to Section 5). However, a solution based on Bloom filter [16] provides scalable solution. A digest broadcast includes a Bloom filters computed using *ComputeBloom* of Algorithm 4

Algorithm 4: The *ComputeBloom* operation

```
input : D is a set of block identifier or address
output: A Bloom filter representation of D
//have an empty Bloom filter
1 Let, B = b_0, b_1, ... b_τ, where b_i = 0, ∀i
2 foreach da ∈ D do
      //add da to B using hash h_j, 1 ⩽ j ⩽ κ
3     for j ← 1 to κ do
          //set the proper bit in the filter:
          //use bitwise OR
4         b_{h_j(d_x)} ← 1
5     end
6 end
7 return B
```

with argument $(\bigcup_{\forall x \in LCI}\{ob_x \| a_x\})$. Here, the logical identification of a block is constructed by concatenating the originality information[4] and the address. Let, designate a filter from peer p as $Bloom_p$.

To check the availability of a block the *SearchBloom* operation of Algorithm 5 is used. At first, the item is looked up as a non-original item (line 1, 2) and later as original (line 3, 4). If both the lookup fail, the item is considered to be not available. It should be noted that Bloom filter suffers from false hit, where the filter gives positive conclusions about elements those are not added to the filter. Moreover, as the cache content changes in between two gossip broadcasts, the proposed scheme also suffers from false hit, where the filter gives positive conclusions about elements those are evicted from the cache.

Algorithm 5: The *SearchBloom* operation

```
input : da is a block identifier or address and B is a
        Bloom filter
output: {avail, orig} to indicate availability and
        originality
//compute Bloom filter for non-original item
1 B_x ← ComputeBloom({false||da})
//is da available in the filter as non-original? use bitwise
  AND
2 if B_x AND B = B_x then return ({true, false}) //now
  try for original item
3 B_x ← ComputeBloom({true||da})
4 if B_x AND B = B_x then return ({true, true})
5 return ({false, NULL})
```

Proposition 1 *Let, for a fully occupied cache, F^{+ve}_{best} and F^{+ve}_{worst} be the best and worst case number of blocks, not added to the filter, resulting in false positive availability, respectively. F^{-ve}_{best} and F^{-ve}_{worst} be the best and worst case*

[3] $g(P)$ and $f(P)$ are used for gossip and fetch events at peer P, and $a \mapsto b$ means a happens before b.

[4] The true/false status may be represented with a single bit.

number of blocks, added the filter, resulting in false positive availability (due to eviction), respectively. With $n_{rem} = \min(\lambda \cdot t_g, n_p)$ *and* $\beta = \frac{\tau}{n_p}$*, where* t_g *is a gossip period, in SPACE, following equalities hold for peer* p.[5]

$$F_{best}^{+ve} = (n - n_p) \cdot \left(1 - e^{-\kappa/\beta}\right)^{\kappa}$$

$$F_{worst}^{+ve} = \frac{n_{rem}}{n - n_p} + \left(1 - \frac{n_{rem}}{n - n_p}\right) \cdot F_{best}^{+ve}$$

$$F_{best}^{-ve} = 0$$

$$F_{worst}^{+ve} = \min\left((n - n_p) \cdot \left(1 - \left(1 - e^{-\kappa/\beta}\right)^{\kappa}\right), n_{rem}\right)$$

With Bloom filter-based solution, it is not possible to evaluate the utility of a peer for an specific block as shown in Eq. 2. Where required (such as line 8 of Algorithm 1), a peer is chosen randomly, instead of raking them based on their utility. In this solution, the *RCB* data structures are useless and hence omitted. The *RCI* structure is modified to include only the filters from all peers, i.e., $Bloom_p; \forall p$.

5 Simulation results

To evaluate the performance of SPACE we conducted extensive simulations.

5.1 Simulation environment

We develop an event driven simulation tool in C++ to investigate the performance of the collaborative cache. The presented results are collected from the logs generated by a batch of simulation runs, taking months of computation time of several computers. Each presented result is the average of ten simulation runs.

The chosen simulated hardware platform is due to the Abacus system [44]. We simulate a 32 node cluster as the underlying hardware. The physical communication medium for the cluster nodes is a Gigabit network, where the nodes are connected to each other through a Gigabit switch. The master node is connected with a RAID storage device, and the device can provide infinite parallel disk access, i.e., each request to read from the storage can be initiated immediately. This device should be the ideal RAID. In the simulations, the server component does not maintain any cache. There are two reasons for this choice. Firstly, a peer, executing at the server node, reduces the need of an extra level of cache. Secondly, as compared to the size of the global pseudo cache, the server cache would be considerably small. As a result, even if the server had a cache, most of the blocks would be evicted while they are still available in the pseudo global cache.

Five different synthetic traces (designated as *trace* $1 \sim 5$) of disk read requests are generated. Each trace consists of 100 million of disk read requests. Like real life, we assumed that the cache schemes do not have any prior knowledge about the trace behavior. First two synthetic traces are generated using *PQRS* algorithm [43], which relates time with the disk block access pattern, using four parameters: p, q, r and s. The parameters are chosen such that the first trace shows more burstiness than the second one, but the second trace shows a behavior with more spatio-temporal locality. The traces may mimic the behavior of a pipeline [31] and a mesh [18] computation, respectively.

The remaining three traces are generated using Zipf-like distribution [9]. The probability to access the i-th object, $P_N(i)$, is defined as,

$$P_N(i) = \frac{\Omega}{i^{\alpha}} \tag{4}$$

where $i = 1, 2, \ldots N$ and N is number of objects in the system. Ω is defined as,

$$\Omega = \left(\sum_{i=1}^{N} \frac{1}{i^{\alpha}}\right)^{-1} \tag{5}$$

The three traces are generated with $\alpha = 0.3, 0.4$ and 0.3, respectively. We assume that all objects are of the same size and fit entirely in a disk block. In situations where an object is larger than a block size, the object can be divided into smaller fragments, so that each fragment fits in a disk block. Zipf-like distribution does not capture spatial locality. Due to lack of proper model, we place object i subsequently after object $(i - 1)$ with probability p_{Zl}, given that the subsequent disk block is empty. Otherwise, the object is placed randomly. For the traces 3 and 4, $p_{Zl} = 0.3$, and for the trace 5, $p_{Zl} = 0.6$ are used. Read requests are considered to be exponentially distributed. Some other parameters of our simulations are listed in Table 1.

The probability that a request for a block, not in a filter, results in a false positive is $(1 - e^{-\kappa/\beta})^{\kappa}$. By carefully choosing a value for β and optimal κ, the false positive probability can be diminished. For example, for our simulated system, the probability is 0.00047. Moreover, due to the spatio-temporal locality of data access, most of requests are served directly from the pseudo global cache (and more about that discussion

[5] Refer to [16] for details about the math behind Bloom filters.

Table 1 Simulation parameters

Parameter name	Value
$\eta, \zeta, \gamma, \beta, \kappa$	∞, 1, 2, 16, 10
Cache size per peer (M)	32, 64, 96, 128 MB
Cache block size (m)	16, 32, 48, 64 kB
Storage Size (D)	2 TB
Storage block size (d)	4 kB
Avg. storage access time	5 ms
Avg. local cache search time	0.05 ms
Warm-up period	64000 read requests
Digest broadcast interval (t_g)	1 min

follows), i.e., most of the requests are for the objects those are already in the cache as well as added to the filter. From our simulations, we observe that the original and Bloom filter-based schemes produce almost identical results. So, we present the results from the Bloom filter-based scheme only, unless mentioned explicitly.

5.2 Simulation results

We compare our proposed scheme, SPACE, with several other algorithms. The first algorithm is an ideal one, we call it the *Global LRU* algorithm (designated as $GLRU$). The algorithm has instant global state information. Though GLRU is not realizable in practice, it gives the upper bound on the performance of any LRU based algorithm. We also compare SPACE with N-chance (designated as NC) and Hierarchical GreedyDual (designated as HGD) schemes. To have a realistic and practical HGD, we assume that no prior access frequencies are available and during runtime, frequencies of only those blocks that are available in the cache are computed. We also assume one level of cluster hierarchy consisting of only one cluster of caches. In this setup, the head of the cluster of caches provides a centralized solution for different cache operations. In our simulations, we do not compare SPACE with any of the local cache replacement algorithms. Interested

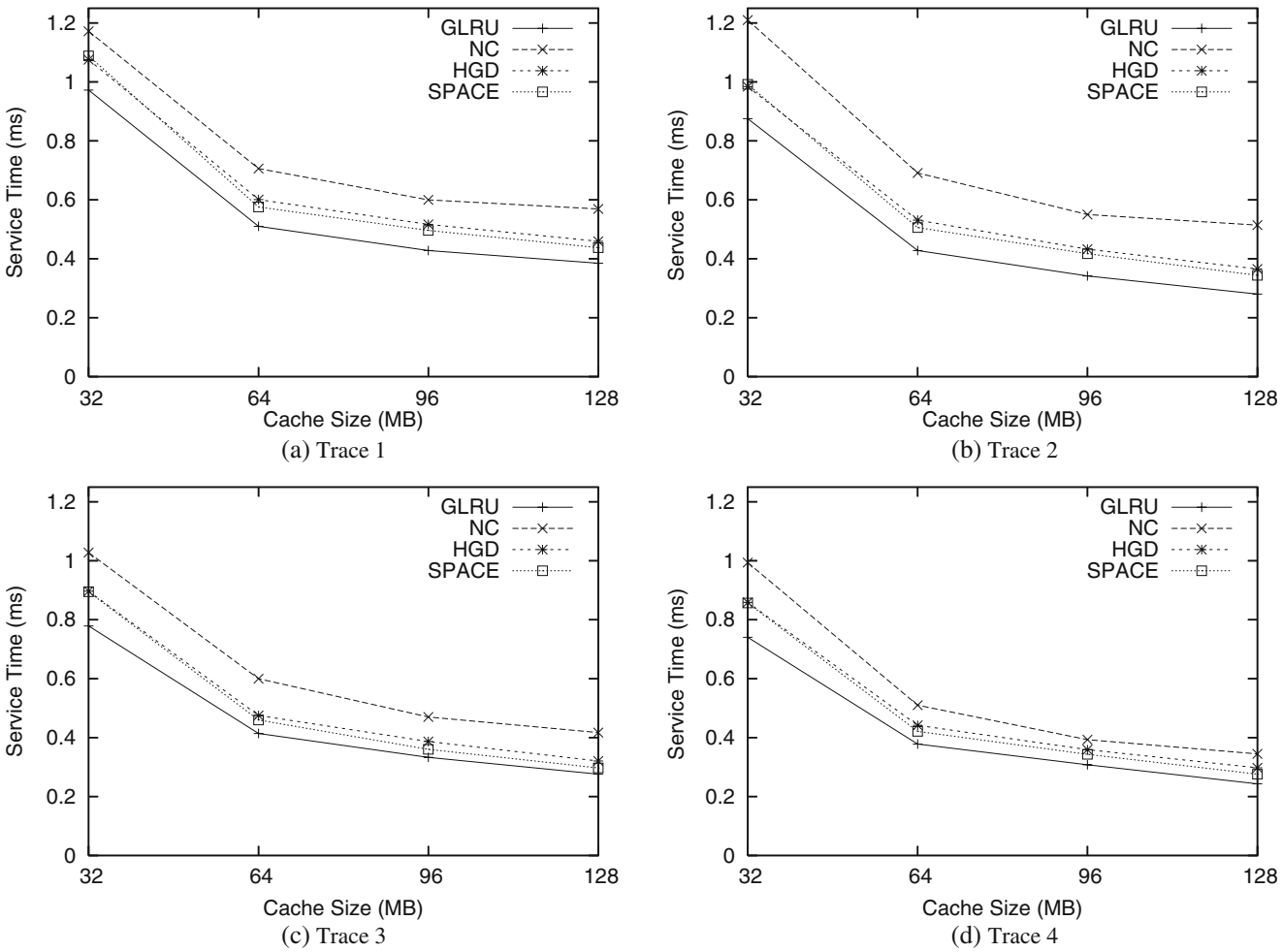


Fig. 3 Service time for different traces (**a**–**d**)

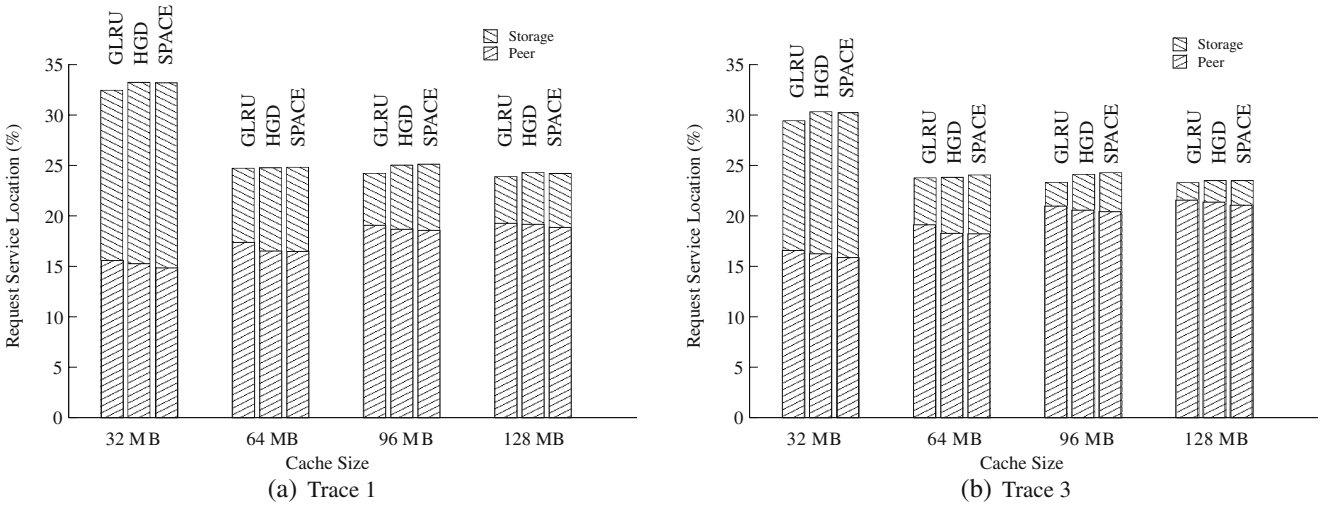


Fig. 4 Service location (**a**, **b**)

readers can find comparisons of such algorithms with HGD in [25].

The comparative results of service time for the traces $1 \sim 4$ are presented in Fig. 3. The figure presents the average time to serve a request for different cache sizes, keeping a single block size fixed to 32 kBs. As expected, in all traces, the GLRU performs the best of all. On the other hand, NC performs the worst of all. These results reflect the findings in a previous research [25]. For traces 1 and 2 (Fig. 3a and b), when the cache size is smaller, performances of HGD are similar to SPACE, but as the cache size grows, SPACE outperforms HGD. Those two traces are burstier than the others. During the busy time (i.e., request burst), the state of the caches changes significantly in a very short time. Therefore, the state information about a remote peer, available from the last broadcast, may become inconsistent with the current state. This inconsistency hurts the performance of SPACE. As the cache size increases, it becomes easier to accommodate newer blocks with more replicas. Then, the performance of SPACE becomes dominant among the candidates (except GLRU). This domination is due to the fact that SPACE eliminates the need for the communication with the central manager at each collaborative cache operation.

Traces 3 and 4 have similar spatial locality, but trace 4 shows higher temporal locality. As we look through the service times for traces 3 and 4, the performance of SPACE follows the performance of GLRU more closely, and outperforms both HGD and NC. With the increment of cache size, it becomes increasingly unnecessary to forward requests to the server. This issue can be easily perceived from Figs. 4 and 5. Figure 4 shows the percentage of read requests served at the remote

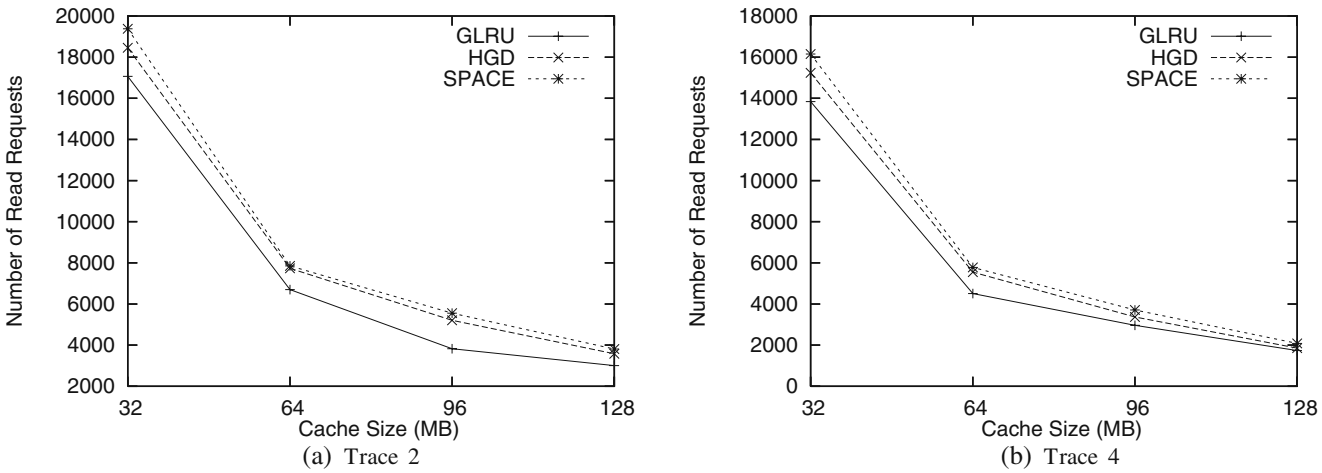
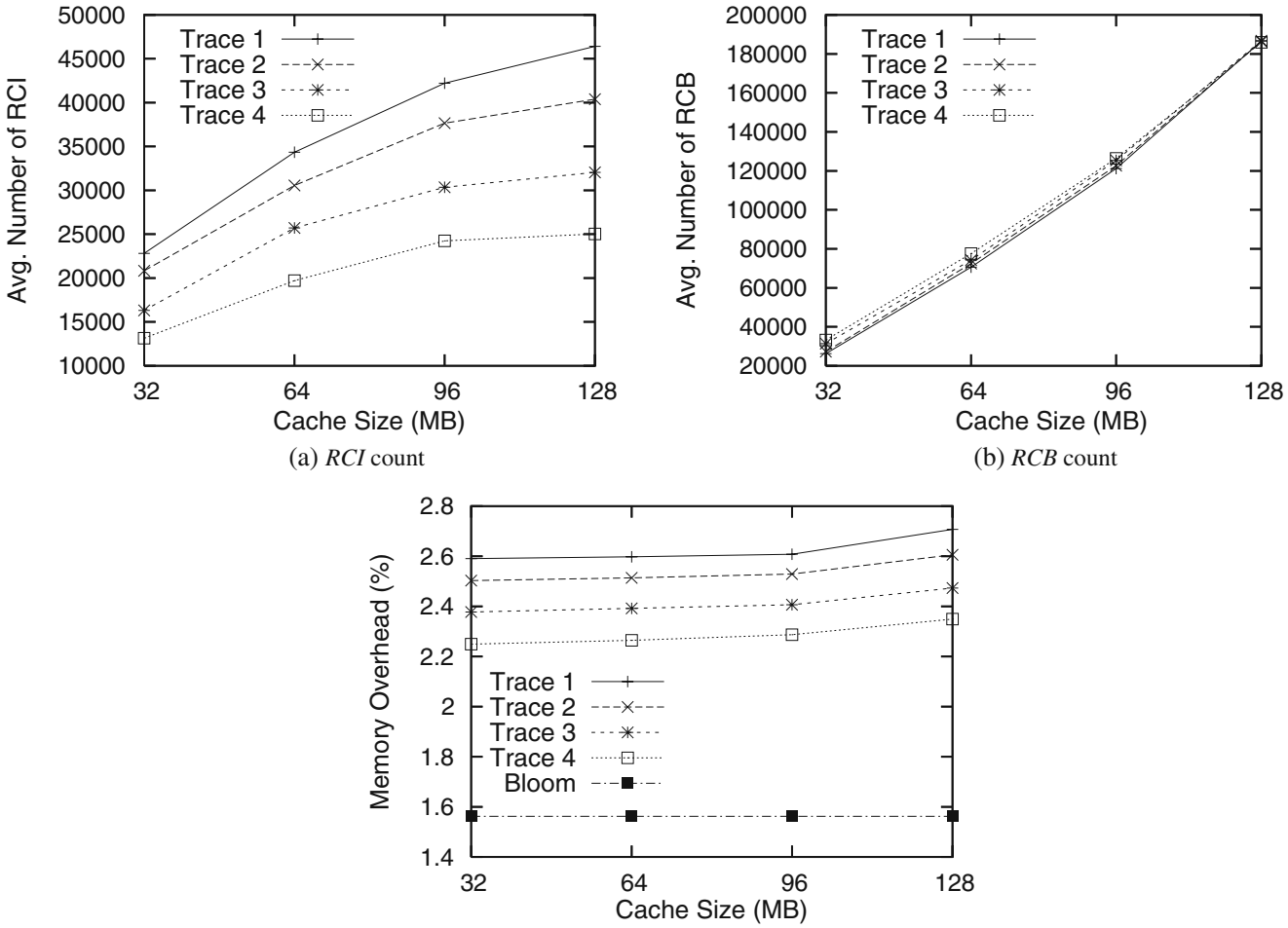


Fig. 5 Server load for trace 2 and 4 (**a**, **b**)



(a) *RCI* count

(b) *RCB* count

(c) Excess memory space

Fig. 6 Memory overhead (**a**–**c**)

peers as well as at the server (and the rest are served from the local cache)[6] for traces 1 and 3. For a clearer visualization, Fig 5 shows only the server load for the traces 2 and 4.

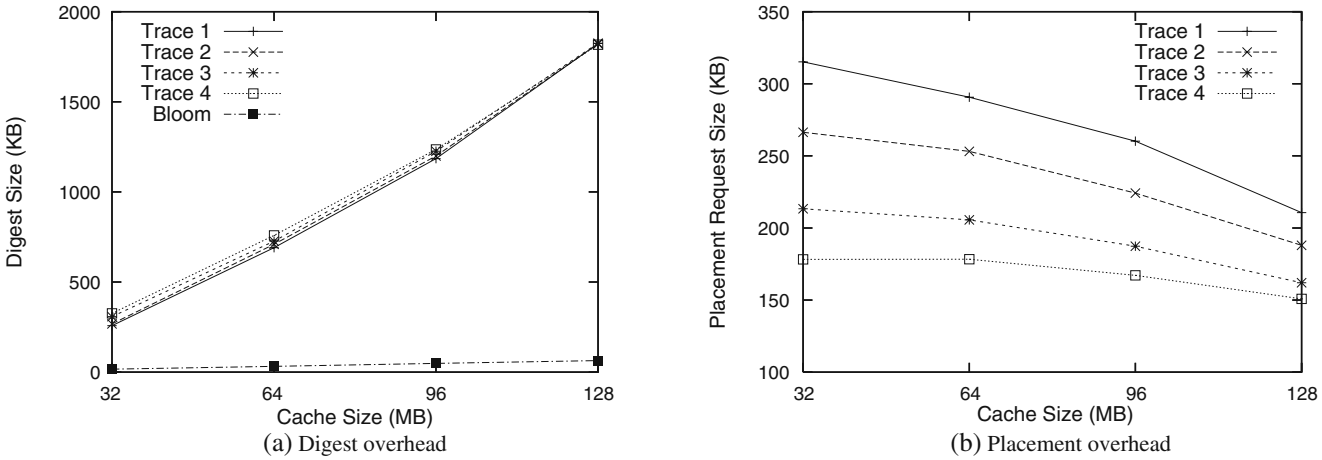
The average time to serve a request is dominated by the service time at the data server. It is evident from the figures that as the cache size increases, the number of service requests to the server decreases in a non-leaner fashion and hence, service time performance increment lags behind (i.e., the curve of Fig. 5 becomes less steeper). The phenomenon starts earlier given that the requests show increased locality. The users of the cache scheme need to find a trade-off between the performance and allocated space. However it should be noted that if the collaboration of caches were not present, most of the requests served by the peers would demand services from the server. That would drastically increase the average time to serve a single request.

The overhead of the proposed scheme is evaluated next. At first, we look into the additional space or memory requirements due to the additional data structures. In the original scheme additional data structures are *RCI* and *RCB* sets. Figure 6 illustrates different aspects of memory overhead. The average number of

Table 2 Size of data structures

Field name	Size
Storage block address (SBA)	8 bytes
Peer identification (PID)	4 bytes
Clock counter (CLK)	15 bits
Flag for original block (OB)	1 bit
Pointer	4 bytes

[6]Note that total percentage of requests served at the remote peers and at the server denotes the local cache miss ratio. The percentage of read requests served at the server denotes the pseudo global cache miss ratio.



(a) Digest overhead (b) Placement overhead

Fig. 7 Communication overhead of SPACE (**a**, **b**)

RCI and *RCB* entries maintained by each peer is shown in Fig. 6a and b, respectively. As the cache size increases, the number of replicas of a block in the the pseudo global cache also increases. As a result, the uphill curve of *RCB* (reference to all remote replicas) becomes more steeper. On the other hand, the curves for *RCI* flatten with the increased cache size.

To evaluate the real impact of the extra memory requirements, we consider a practical implementation of both of the data structures. Different fields from the data structures are presented in Table 2. A *RCI* entity is assumed to consist of one SBA, one CLK, one OB and three pointers (two to maintain the *RCI* set in a double-way linked list and the third one to point to the associated *RCB* set). A *RCB* entity consists of one PID, one CLK, one OB and two pointers (to maintain the *RCB* set in a double-way linked list). The total memory overhead is presented in Fig. 6c after normalizing with the cache size. In all the observations, the memory overhead is less than 3%. The memory overhead for the Bloom filter-based scheme is due to the filters. For a given cache size, the filter size is independent of the characteristics of the traces. Figure 6c also shows the overhead of the Bloom filter-based scheme under the legend *Bloom*. The overhead of this scheme is sufficiently smaller (less than 1.6%).

The second overhead of SPACE is the communication overhead and is illustrated in Fig. 7. During the simulation, we assume 100 bytes of protocol overhead for each message. In Fig. 7a, the average overhead from all the digest broadcasts for one minute period is presented. In fact, this overhead is directly proportional to the average number of *RCB*s maintained by each peer. The cost due to the placement of blocks at remote peers is shown in Fig. 7b. As the cache size increases, it becomes easier to accommodate new blocks and the number of remote placement operations declines. Even if the communication medium is considered as a bare broadcast channel, in the worst case, the total communication overhead in the entire system is insignificant (less than 1% and 0.1% for original and Bloom filter-based scheme, respectively) compared with the total bandwidth capacity of the network. In reality, the capacity of the network with a modern intelligent switch is far higher than the broadcast channel, and hence, the communication overhead can be ignored. It should be noted that the gossip period (t_g) determines two conflicting performance parameters—the communication overhead and the worst case false positive availabilities. However, a system designer may compute these parameters analytically (refer to Proposition 1) and decide an optimum value for the application in hand.

In our simulation, we compute the number of fetch services provided by each peer. To make a conclusion, we normalize each data set with the average where the average is considered to be 100. The standard

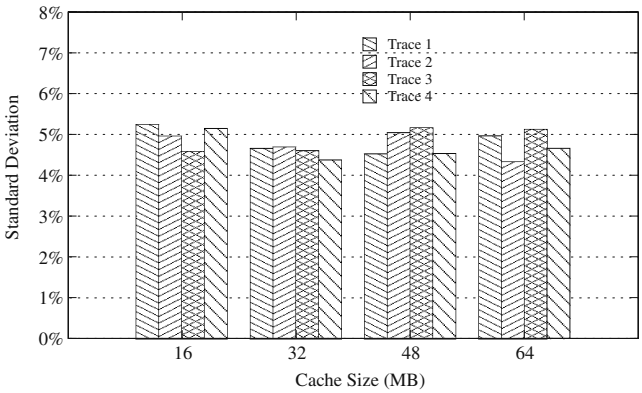


Fig. 8 Standard deviation of service time

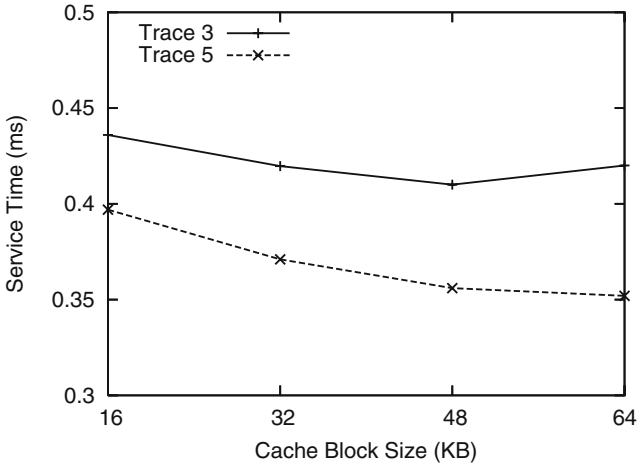


Fig. 9 Impact of depth on service time

deviations of the average number of fetch requests served by each peer are presented in Fig. 8. We observe that the loads of the peers are reasonably equal.

Finally, we present the impact of depth (designated as c in Section 4.1) of each cache block. We found that the performance of SPACE is similar to the performance of a sequential program on a single cache with different depths [20]. The results from the simulation is presented in Fig. 9 where the cache size is fixed to 64 megabytes. In general, a higher depth is preferable if the requests show more spatial locality.

6 Further discussion and conclusion

In this paper, we have considered a simple LRU scheme for local block eviction. However, the LRU-based improvements proposed in [24, 45] could easily be incorporated into our scheme. In addition, peer statistics are considered in our research to be distributed with the digest only. In practice, the statistics can always be piggy-backed with other kinds of messages to disseminate the most current status. Though clock based LRU cache for storage systems is an excellent design choice [40], the clock management can be improved using techniques discussed in [7]. Besides, at each access, update of all the clocks can be efficiently implemented using relative clocks, as described in [10]. Bloom filter used in this paper can be replaced with Optimal Bloom filter [29], which reduces memory and communication overhead due to digest by 30.6%. Finally, we have focused on read operations only, but a wide variety of parallel applications require write access to the data. Many of them may not require data consistency. The face recognition application, described in Section 3.2, requires only read access to the data. In contrast, the described parallel OLAP application mandates write access to store the computed views. Depending on the algorithm, it may not produce multiple copies of the same data, and no consistency issue arises. If an application requires write access where data consistency has to be ensured, an existing consistency mechanism can be employed. A comprehensive study of different consistency mechanisms can be found in [1].

In conclusion, we have proposed a collaborative caching scheme for clusters. Our scheme is based on peer-to-peer computing model. The peers form and maintain a collaborative cache in a distributed way, without a central manager. By combining individual caches of the peers, a large pseudo global cache can be obtained. We have demonstrated that the scheme is very efficient and can approximate the Global LRU scheme, yet with reasonably low communication and memory overhead. Our ongoing research includes an integrated cache consistency mechanism, and its suitable use in large LAN and WAN environments.

Acknowledgement Financial support of this research has been provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

1. Adve SV, Gharachorloo K (1996) Shared memory consistency models: a tutorial. Comput 29(12):66–76
2. Akon M, Goswami D, Li HF, Shen XS, Singh A (2008) A novel software-built parallel machines and their interconnections. J Interconnection Netw 9:1–29
3. Akon MM, Goswami D, Li HF (2004) SuperPAS: a parallel architectural skeleton model supporting extensibility and skeleton composition. In: International symposium on parallel and distributed processing and applications, pp 985–996
4. Akon MM, Goswami D, Li HF (2005) A model for designing and implementing parallel applications using extensible architectural skeletons. In: International conference on parallel computing technologies, pp 367–380
5. Akon MM, Singh A, Goswami D, Li HF (2005) Extensible parallel architectural skeletons. In: IEEE international conference on high performance computing, pp 290–301
6. Akon MM, Singh A, Shen X, Goswami D, Li HF (2005) Developing high-performance parallel applications using EPAS. In: International symposium on parallel and distributed processing and applications, pp 431–441
7. Bansal S, Modha D (2004) CAR: clock with adaptive replacement. In: USENIX conference on file and storage technologies, pp 187–200
8. Bowman CM, Danzig PB, Hardy DR, Manber U, Schwartz MF (1995) The Harvest information discovery and access system. Comput Netw ISDN Syst 28(1–2):119–125
9. Breslau L, Cao P, Fan L, Phillips G, Shenker S (1999) Web caching and zipf-like distributions: evidence and implications. In: INFOCOM, pp 126–134
10. Cao P, Irani S (1997) Cost-aware WWW proxy caching algorithms. In: Usenix symposium on internet technologies and systems, pp 193–206

11. Chen Y, Dehne F, Eavis T, Rau-Chapli A (2004) Parallel ROLAP data cube construction on shared-nothing multiprocessors. Distributed and Parallel Databases 15(3): 219–236
12. Chi HC, Zhang Q (2005) Deadline-aware network coding for video on demand service over P2P networks. HKUST 7(22–23):755–763
13. Chi HC, Zhang Q, Shen X (2007) Efficient search and scheduling in P2P-based media-on-demand streaming service. IEEE J Sel Areas Commun 25(1):119–130
14. Cuenca-Acuna FM, Nguyen TD (2001) Cooperative caching middleware for cluster-based servers. In: 10th IEEE international symposium on high performance distributed computing, pp 303–315
15. Dahlin M, Wang R, Anderson TE, Patterson DA (1994) Cooperative caching: using remote client memory to improve file system performance. In: Operating systems design and implementation, pp 267–280
16. Fan L, Cao P, Almeida J, Broder AZ (2000) Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans Netw 8(3):281–293
17. Ghemawat S, Gobioff H, Leung ST (2003) The Google file system. In: ACM symposium on operating systems principles, pp 29–43
18. Globisch G (1995) PARMESH—a parallel mesh generator. Parallel Comput 21(3):509–524
19. Goil S, Choudhary AN (1997) High performance OLAP and data mining on parallel computers. Data Mining and Knowledge Discovery 1(4):391–417
20. Hennessy JL, Patterson DA (2002) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann, San Francisco
21. Howard JH, Kazar ML, Menees SG, Nichols DA, Satyanarayanan M, Sidebotham RN, West MJ (1998) Scale and performance in a distributed file system. ACM Trans Comput Syst 6(1):51–81
22. Hu A (2001) Video-on-demand broadcasting protocols: a comprehensive study. In: IEEE INFOCOM, pp 508–517
23. IBM (2007) IBM general parallel file system. http://www.ibm.com/systems/clusters/software/gpfs.html
24. Kampe M, Stenstrom P, Dubois M (2004) Self-correcting LRU replacement policies. In: CF '04: proceedings of the 1st conference on computing frontiers, Ischia, pp 181–191
25. Korupolu MR, Dahlin M (2002) Coordinated placement and replacement for large-scale distributed caches. IEEE Trans Knowl Data Eng 14(6):1317–1329
26. Kothari A, Agrawal D, Gupta A, Suri S (2003) Range addressable network: a P2P cache architecture for data ranges. In: Third international conference on peer-to-peer computing, Sweden, pp 14–23
27. Linga P, Gupta I, Birman K (2003) A churn-resistant peer-to-peer web caching system. In: ACM workshop on survivable and self-regenerative systems, pp 1–10
28. Nelson MN, Welch BB, Ousterhout JK (1998) Caching in the Sprite network file system. ACM Trans Comput Syst 6(1):134–154
29. Pagh A, Pagh R, Rao SS (2005) An optimal bloom filter replacement. In: Annual ACM-SIAM symposium on discrete algorithms, pp 823–829
30. Patterson DA, Gibson G, Katz RH (1988) A case for redundant arrays of inexpensive disks (raid). In: International conference on management of data (SIGMOD), pp 109–116
31. Radenski A, Norris B, Chen W (2000) A generic all-pairs cluster-computing pipeline and its applications. In: Parallel computing: fundamentals & applications, pp 366–374
32. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S (2001) A scalable content-addressable network. In: ACM SIGCOMM, pp 161–172
33. Red Hat, Inc (2007) Red hat global file system. http://www.redhat.com/software/rha/gfs/
34. Rousskov A, Wessels D (1998) Cache digests. Comput Netw ISDN Syst 30(22–23):2155–2168
35. Sandberg R, Goldberg D, Kleiman S, Walsh D, Lyon B (1985) Design and implementation of the sun network filesystem. In: Proc. summer 1985 USENIX conf, pp 119–130
36. Sarkar P, Hartman J (1996) Efficient cooperative caching using hints. In: OSDI '96: proceedings of the second USENIX symposium on operating systems design and implementation, Seattle, pp 35–46
37. Sarkar P, Hartman JH (2000) Hint-based cooperative caching. ACM Trans Comput Syst 18(4):387–419
38. Satyanarayanan M, Kistler JJ, Kumar P, Okasaki ME, Siegel EH, Steere DC (1990) Coda: a highly available file system for a distributed workstation environment. IEEE Trans Comput 39(4):447–459
39. Skobeltsyn G, Aberer K (2006) Distributed cache table: efficient query-driven processing of multiterm queries in P2P networks. Tech rep LSIRRE-PORT-2006-010, EPFL, Lausanne, Switzerland
40. Tanenbaum AS, Woodhull AS (2006) Operating systems design and implementation, 3rd edn. Prentice Hall, Englewood Cliffs
41. Tewari R, Dahlin M, Vin HM, Kay JS (1999) Design considerations for distributed caching on the internet. In: International conference on distributed computing systems, pp 273–284
42. Wang C, Xiao L, Liu Y, Zheng P (2006) DiCAS: an efficient distributed caching mechanism for P2P systems. IEEE Trans Parallel Distrib Syst 17(10):1097– 1109
43. Wang M, Ailamaki A, Faloutsos C (2002) Capturing the spatio-temporal behavior of real traffic data. Perform Eval 49(1–4):147–163
44. University of Waterloo (2007) Abacus cluster. http://abacus.uwaterloo.ca/
45. Wong WA, Baer JL (2000) Modified LRU policies for improving second-level cache behavior. In: High-performance computer architecture, pp 49–60
46. Zhang L, Michel S, Nguyen K, Rosenstein A, Floyd S, Jacobson V (1998) Adaptive web caching: towards a new global caching architecture. In: 3rd international WWW caching workshop, pp 2169–2177

Mursalin Akon received his B.Sc.Engg. degree in 2001 from the Bangladesh University of Engineering and Technology (BUET),

Bangladesh, and his M.Comp.Sc. degree in 2004 from the Concordia University, Canada. He is currently working towards his Ph.D. degree at the University of Waterloo, Canada. His current research interests include peer-to-peer computing and applications, network computing, and parallel and distributed computing.

Towhidul Islam received his B.Sc.Engg. degree in 2001 from the Bangladesh University of Engineering and Technology (BUET), Bangladesh, and his M.Sc. degree in 2004 from the University of Manitoba, Canada. He is currently working towards his Ph.D. degree at the University of Waterloo, Canada. His current research interests include peer-to-peer computing and applications, service oriented architectures, and mobile computing.

Xuemin Shen received the B.Sc.(1982) degree from Dalian Maritime University (China) and the M.Sc. (1987) and Ph.D. degrees (1990) from Rutgers University, New Jersey (USA), all in electrical engineering. He is a Professor and the Associate Chair for Graduate Studies, Department of Electrical and Computer Engineering, University of Waterloo, Canada. His research focuses on mobility and resource management in wireless/wired networks, wireless security, ad hoc and sensor networks, and peer-to-peer networking and applications. He is a co-author of three books, and has published more than 300 papers and book chapters in different areas of communications and networks, control and filtering. Dr. Shen serves as the Technical Program Committee Chair for IEEE Globecom'07, General Co-Chair for Chinacom'07 and QShine'06, the Founding Chair for IEEE Communications Society Technical Committee on P2P Communications and Networking. He also serves as the Editor-in-Chief for Peer-to-Peer Networking and Application; founding Area Editor for IEEE Transactions on Wireless Communications; Associate Editor for IEEE Transactions on Vehicular Technology; KICS/IEEE Journal of Communications and Networks, Computer Networks; ACM/Wireless Networks; and Wireless Communications and Mobile Computing (Wiley), etc. He has also served as Guest Editor for IEEE JSAC, IEEE Wireless Communications, and IEEE Communications Magazine. Dr. Shen received the Excellent Graduate Supervision Award in 2006, and the Outstanding Performance Award in 2004 from the University of Waterloo, the Premier's Research Excellence Award (PREA) in 2003 from the Province of Ontario, Canada, and the Distinguished Performance Award in 2002 from the Faculty of Engineering, University of Waterloo. Dr. Shen is a registered Professional Engineer of Ontario, Canada.

Ajit Singh received the B.Sc. degree in electronics and communication engineering from the Bihar Institute of Technology (BIT), Sindri, India, in 1979 and the M.Sc. and Ph.D. degrees from the University of Alberta, Edmonton, AB, Canada, in 1986 and 1991, respectively, both in computing science. From 1980 to 1983, he worked at the R & D Department of Operations Research Group (the representative company for Sperry Univac Computers in India). From 1990 to 1992, he was involved with the design of telecommunication systems at Bell-Northern Research, Ottawa, ON, Canada. He is currently an Associate Professor at Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. His research interests include network computing, software engineering, database systems, and artificial intelligence.