Cooperative Caching with Return on Investment

Gala Yadgar
Computer Science Department, Technion
Haifa, Israel
Email: gala@cs.technion.ac.il

Michael Factor
IBM Research
Haifa, Israel
Email: factor@il.ibm.com

Assaf Schuster
Computer Science Department, Technion
Haifa, Israel
Email: assaf@cs.technion.ac.il

***Abstract*—Large scale consolidation of distributed systems introduces data sharing between consumers which are not centrally managed, but may be physically adjacent. For example, shared global data sets can be jointly used by different services of the same organization, possibly running on different virtual machines in the same data center. Similarly, neighboring CDNs provide fast access to the same content from the Internet. Cooperative caching, in which data are fetched from a neighboring cache instead of from the disk or from the Internet, can significantly improve resource utilization and performance in such scenarios.**

However, existing cooperative caching approaches fail to address the selfish nature of cache owners and their conflicting objectives. This calls for a new storage model that explicitly considers the cost of cooperation, and provides a framework for calculating the utility each owner derives from its cache and from cooperating with others. We define such a model, and construct four representative cooperation approaches to demonstrate how (and when) cooperative caching can be successfully employed in such large scale systems. We present principal guidelines for cooperative caching derived from our experimental analysis. We show that choosing the best cooperative approach can decrease the system's *I/O delay* by as much as 87%, while imposing cooperation when unwarranted might increase it by as much as 92%.

I. Introduction

Resource consolidation is a prevalent means for saving power, maintenance, administrative and acquisition costs. Traditionally, storage and compute resources were consolidated within organizations [1], [2]. Recently, however, resources are being consolidated on a much larger scale, often involving resources owned, or chartered, by different entities. Common examples include computational grids [3], clouds [4], and large scale data centers [5].

The widespread use of such large scale environments introduces new data sharing scenarios, where the same data are accessed by physically neighboring services or end users of different priority or ownership. The fast, high bandwidth network within these environments makes it substantially cheaper to access data from a neighbor's cache than it is to access the shared storage. Common scenarios involve separate services that use the same data repository [6], [7]. For example, customer data hosted on a DBaaS (database as a service) cloud may be accessed by a company's search engine, as well as by an external advertisement service, hosted on the same cloud. The services run on separate sets of virtual machines, but, augmented with a simple messaging protocol, they can access data stored in each other's cache.

Another example is content distribution networks, frequently used for large-scale content delivery, offsetting traffic from the content provider's infrastructure. Recently, several working groups have been discussing interfaces and specifications to facilitate federated CDNs [8], [9]. The feasibility of such federation depends on a suitable cooperation model that will enable sharing of cache content between these otherwise competing entities [10].

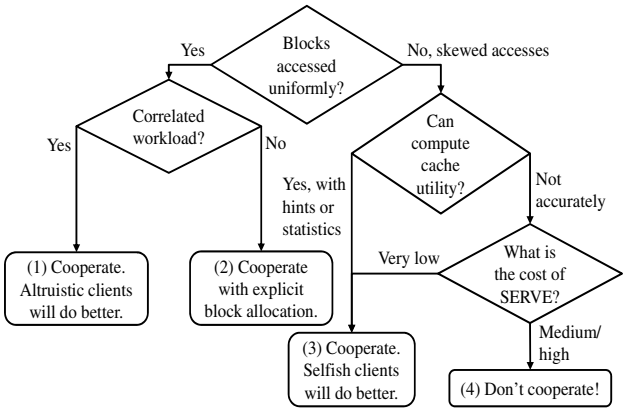


Figure 1. Cooperation guidelines derived from our experimental analysis (Section V).

Cooperative caching can eliminate unnecessary redundancy and greatly improve performance by avoiding slow access to content residing on disk or at WAN distances. However, cooperation incurs additional overheads that may degrade the performance of some participating caches. Thus, *selfish* cache owners, having separate, possibly conflicting, objective functions, will not cooperate.

Cloud users are a textbook example of selfish entities, since they are directly charged for the resources that they rent [11]. They may agree to "sublet" their caches within some cooperative framework [12], but will do so only if their *return on investment* is guaranteed. For example, consider two departments within the same organization, accessing the same data but paying separate cloud and electricity bills. Accessing one another's caches will increase their cache utilization, providing improved performance, or, alternatively, opportunity to scale down their caching tiers and reduce costs [13]. However, each department will agree to cooperate only if it is certain that its own bill will not increase as a result.

Existing frameworks for cooperative storage caching were designed for centrally owned caches, with the goal of minimizing global I/O response time [14], [15], [16], [17], [18], [19], and lack *incentives* for selfish cache owners to cooperate. While selfishness is well-studied in the network domain [20],

[21], [22], [23], the corresponding peer-to-peer mechanisms deal with short term cooperation. We will show in our analysis that these mechanisms are insufficient for efficient cooperation in stateful systems such as caches. At the same time, existing theoretical models for cooperation are computationally hard [24] and impractical for managing large scale dynamic systems.

The above limitations of existing approaches call for a new storage model. Such a model should satisfy two requirements: an explicit accounting for the **cost** of cooperation, and a way to compute the true **utility** provided by a cache in a collaborative system. This will allow cache owners to calculate their **return on investment**, by weighing both the work invested in cooperation and its effect on individual performance.

Our contribution is threefold. First, we define a new storage model that satisfies the above requirements. Second, we present four novel caching approaches whose variations cover a range of client behaviors, from selfish to *altruistic*, where clients always cooperate. Third, from the extensive analysis of these approaches, we derive basic rules of thumb for cooperative caching. These rules, depicted in Figure 1, expose the potential benefits and limitations of cooperation with selfish clients. For example, we show that choosing the best cooperative approach can decrease the time spent on I/O by as much as 87%, while imposing cooperation when unwarranted can as much as double this time.

The rest of this paper is organized as follows. In Section II, we motivate our new cooperative storage model, and formalize its operations and costs. We introduce our new cooperative approaches in Section III, and describe our evaluation methodology in Section IV. We present our results and analysis in Section V, with additional design considerations in Section VI. We discuss related work in Section VII, and conclude in Section VIII.

II. Cooperative Storage Model

Traditionally, cooperative caching was considered in systems with central ownership and management. Accordingly, the goal of cooperative global memory management algorithms was to optimize the entire system's I/O response time [14], [15], [16], [17], [18]. However, in the emerging resource consolidation models, caches belong to different owners and administrative domains, and should primarily be used for the purpose and benefit of their owners. Therefore, a new storage model is required, that addresses caches as *selfish* entities, which cooperate with one another only if the *benefit* of doing so exceeds the *cost*. In this section, we describe the two key requirements from such a model, followed by a formal definition of our proposed model and its operations. We refer to selfish cache owners as *clients*, while *server* refers to the system's centrally owned, shared, cache.

A. Service Cost

Previous research on cooperative caching focused on how cooperation affects the hit rate of participating caches, and consequently, on determining which data blocks to store in each cache. In this study, we consider, for the first time, the cost incurred on the *serving* cache; its CPU must initiate a copy of the requested data block, after which the network

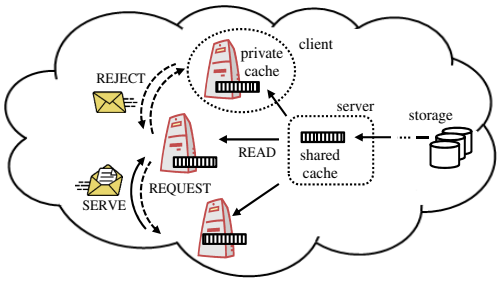


Figure 2. Cooperative storage model and operations

transfer takes place in the background, involving its NIC and possibly its DMA engine. This cost is usually negligible when compared to the cost of the disk or Internet access that the service replaces. However, the cost is incurred on the serving cache, while the benefit is gained by its peer. Thus, this cost may prevent selfish cache owners from participating in a cooperative caching scheme.

Therefore, the first requirement of the new model is to explicitly define a SERVE operation, similar to the way existing models define READ and WRITE. The cost of SERVE should be explicitly stated in terms of the cache owner's objective function, allowing it to choose whether to perform the operation. The cost of existing I/O operations is usually defined in terms of the disk accesses, network transfers, and queuing delays they incur. We describe the cost of SERVE for a few prevalent objective functions.

When the objective is to minimize I/O response time or application run time, the cost of SERVE is the *service delay* – the delay in its own compute tasks incurred by a client while its CPU is busy serving peer requests. The delay may be shorter if the client's CPU is idle, waiting for I/O, when the request arrives. Another cost, which is out of this paper's scope, is the additional energy consumed by the CPU when serving a peer, no matter when the request arrives – either the CPU was busy doing actual work and must now do the work of serving the request, or it was in an idle, "sleep" state, and must now "wake up" before its I/O operation is complete. This cost is regularly considered in the context of ad hoc networks of battery operated devices [23]. It is also closely related to objectives such as *Energy productivity* [25], [26] which are used to characterize individual applications' performance in data centers.

Serving peer requests consumes the client's upload bandwidth, whose cost implicitly motivates BitTorrent's "tit for tat" policy [20]. We elaborate on the relationship between upload bandwidth limitations and download objectives in Section IV-A. Finally, when a SERVE is performed to benefit a competing business as in the federated CDN vision, the financial implications should be carefully calculated and incorporated into its cost.

B. Utility

Cache owners should be able to compare the cost incurred by cooperation to the benefit it entails. Thus, the second requirement is that a client be able to measure the utility it derives from the content of its own cache, as well as the

utility it can derive from remote caches. We define the *utility* of a cache as the savings in I/O cost achieved by using the cache. These savings depend on the cache content as well as on the various I/O costs in each particular storage setting. Selfish clients aim to cooperate iff

$Utility$ (private cache content without cooperation) $<$
$[Utility$ (private cache content with cooperation)[1]
$+ Utility$ (content accessed from remote caches)
$- Cost$ (total accesses to remote caches)
$- Cost$ (total SERVES to peers)].

The system's incentive mechanism may include some form of credit transfer for each cooperative transaction between pairs of clients. In that case, the utility of each such transaction can be computed on the basis of the change in cache content and credit transfer of that transaction.

To compute the utility of a cache, its owner must be familiar with the relative costs of operations in the system, e.g., *how expensive is a disk access compared to an access to a peer cache*. In addition, it must be able to evaluate the data blocks stored in the cache – the expected hit rate derived from the cache's contents for the duration of cooperation. Hit rate can be calculated, for example, on the basis of query execution plans in relational databases, or other forms of application hints [27] and workload attributes [2]. When accurate hints are unavailable, the hit rate can be estimated via methods such as statistics gathering in separate queues [28] or ghost caches [29], analytic models [13] and active sampling [4].

Two parameters determine the selfish behavior of a client. One is the decision which blocks to cache, and the other is the decision which blocks to SERVE. Figure 3 depicts the schematic *scale of selfishness*, which is affected by these two parameters. At one end, perfectly altruistic clients cooperate entirely according to the system's global objective. At the other end, perfectly selfish clients cooperate strictly according to the above inequality. We expect most realistic implementations to reside throughout this parameter space, indicating that clients may be willing to risk some extra costs for potential benefits in the future. We expect the clients' behavior to be affected, for example, by their ability to predict their utility, the system's ability to guarantee the expected benefit from cooperation, their trust in their peers, etc. We discuss the relative importance of the two parameters in the following sections.

A model based on costs and utilities is particularly appealing in the context of cloud environments. Cloud services are explicitly priced, and users are expected to continuously analyze their costs, benefits, and alternatives, to determine their required resources and SLAs [4], [11], [13], [30], [31], [32]. They can leverage the same calculations to estimate their utility from cooperation. In addition, cloud providers may enhance their services by providing a framework for cooperation, thus increasing the utility customers can derive from their resources.

C. Model Definitions

Our model, depicted in Figure 2, consists of several clients with access to a shared, centrally owned, storage server. Clients are separate, possibly virtual, machines, each with a private cache. For simplicity, we assume that each client is running a single service or application.[2] The server provides access to data residing on persistent *storage*, either on a disk attached to it or on the Internet. It also manages its own cache of blocks fetched from the storage.

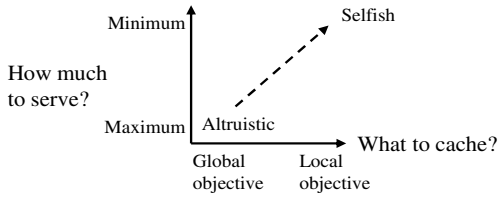


Figure 3. Parameters that affect client selfishness. Selfish clients aim to minimize the work they do for others and to cache the blocks most valuable to them.

In addition, clients may access blocks stored in caches of neighboring clients, called *peers*. We assume that the clients and server are placed at uniform distances, in terms of access time, from one another, while the storage is much further away. The clients and the server can be trusted in terms of data integrity and adherence to caching and cooperation protocols. However, clients are *autonomous* – they decide whether to *participate* in the protocol, according to their selfish objectives.

The basic I/O operation in our model is READ, with which clients fetch blocks from the server. We add to it three cooperative operations. On a cache miss, a client may REQUEST a block from one of its peers. The peer either SERVES the request, sending the block to the requesting client, or REJECTS the request. The cooperative operations can be implemented by a simple messaging protocol, and do not depend on the client's physical location. The decisions whether to SERVE or REJECT a request, and whether to store or discard SERVED blocks, are not defined by the operation. Rather, they depend on the cooperative caching approach adopted by the clients.

Our model defines three basic costs, whose values are determined by the clients' objective functions:
– C_{net} is the cost of performing a local network transfer. This cost is incurred on a client whenever it receives a block from the server cache or from a peer.
– $C_{storage}$ is the cost of performing a disk or Internet access. Since all data blocks first arrive at the server cache, even if they are not stored there, the cost incurred by a client for bringing a block from persistent storage is $C_{storage} + C_{net}$.
– C_{serve} is the cost incurred by a client when it serves a peer request. We assume that the cost of sending or rejecting a request is negligible.

We focus our analysis on minimizing application run time, and use the following terms. The *average I/O response time* of a client is the average time this client waits for a block required by its application. The *service delay* is the total delay imposed on a client's computation while the CPU is busy serving peer requests. To compute the *average service delay*, we divide the client's service delay by the number of its own block accesses. The *average I/O delay* is the sum of average I/O response time and average service delay. It represents the average time a client "wastes" on I/O operations. The server

[1]Possible reduction in hit rate as a result of cooperation is incorporated into the utility of the private cache with cooperation.

[2]In the following, we use the term 'application' to distinguish it from the service caches perform when sharing their data blocks.

is centrally owned, and is therefore not considered a selfish entity. Its objective is to minimize the overall I/O delay in the system.

III. Caching Approaches

We first describe state-of-the-art caching approaches optimized for multiple clients. We use those approaches to define the baseline performance achievable without cooperation. We then describe our cooperative approaches. We suggest four approaches for coordination and management of the cooperating clients, with an increasing degree of selfishness. A cooperative caching algorithm is composed of two logical components. The *cache management component* is responsible for allocation and replacement. The *cooperation component* is responsible for selecting peers to REQUEST blocks from, and for deciding whether to SERVE or REJECT peer requests. Parts or all of the logic of the different components can be implemented at the server or the clients, depending on the algorithm. Figure 4 summarizes the characteristics of our cooperative policies, and places them on the scale between selfish and altruistic.

A. Noncooperative Approaches

The purpose of cooperative caching is to improve cache utilization by *exclusivity* – eliminating data redundancy between client caches. Therefore, our cooperative approaches are also designed to eliminate redundancy between the client and server caches. To evaluate their efficiency, we compare them to three non-cooperative approaches that vary in their degree of exclusivity.

With **LRU** replacement, the least recently used block in the cache is evicted to make room for a new block. LRU is an *inclusive* policy, thus blocks can be stored in both client and server caches. The second policy, **ARC** [33], distinguishes between new blocks that are stored for a trial period in the $L1$ list, and useful blocks that have been seen more than once and are stored in the main, $L2$ list. Metadata of recently evicted blocks is stored in a dedicated *ghost cache*. In ARC, the server is responsible for achieving exclusivity: when serving a READ request, it probabilistically decides whether to cache the block, or to PROMOTE [34] it to the requesting client's cache. The PROMOTE operation does not incur an additional cost over the cost of the READ request that triggered it.

The third policy, $\boldsymbol{MC^2}$ [35], uses application hints to divide blocks into *ranges*, each with a known access frequency and pattern. Cache partitions are allocated to ranges in one of the cache levels, according to the access frequency of their blocks, and managed according to their access patterns. Clients achieve exclusivity by using two operations: they DEMOTE [36] evicted blocks by sending them to the server for second level caching. In addition, they request some blocks by using READ-SAVE (instead of READ), indicating that they do not wish to store them, instructing the server to keep them in its cache. Demoting a block requires an additional network transfer, thus the cost of DEMOTE is C_{net}. In contrast, READ-SAVE incurs the same cost as the READ it replaces.

B. Cooperative Distributed Hash Table

C-DHT is constructed for perfectly *altruistic* clients. It serves to test the applicability of distributed storage techniques to caching. The cooperation component generates a hash key for each block, and distributes the key space evenly so that each client is responsible for an agreed upon portion of the keys, as in Chord [37]. Clients request blocks from the responsible peer, which in turn always serves requests that hit in its cache. If a request is rejected, the client fetches the block from the server.

Cache management is based on **Demote** [36], the exclusive version of LRU, with the restriction that clients only store blocks they are responsible for. When a block is requested by a peer, it is moved to the MRU (most recently used) position, as if it was locally accessed by the client. If a client is responsible for more blocks than can fit in its cache, then the LRU blocks are DEMOTED to the server upon eviction. The server only stores blocks DEMOTED to it, or blocks requested by clients not responsible for them. This guarantees perfect exclusivity between all caches in the system.

Clients dedicate a small, *private*, portion of their cache to an LRU partition of recently accessed blocks. This allows clients to duplicate blocks other clients are responsible for, if they are now being used repeatedly. We experimented with various sizes of this partition, but omit the details for lack of space. We fixed its size at 3 blocks for the TPCH workloads, and at 20% the size of the client cache for the OLTP and video workloads. Alternatively, this size can be adjusted dynamically at run time [38].

C. Peer-to-Peer Cooperative Caching

C-P2P is constructed for moderately *selfish* clients. It adapts peer-to-peer techniques, and specifically BitTorrent [20], to storage caching. In BitTorrent, peers query a dedicated *tracker* to learn about other peers downloading the same file. The set composed of such peers is called a *swarm*. Peers upload data to peers in a *tit for tat* manner, favoring peers that upload to them.

In C-P2P, the server tracks accesses and DEMOTES to each block range. All clients that currently cache blocks from each range compose its *swarm*. Clients periodically query the server for the swarms of the ranges they currently access. On a cache miss, a client requests the block from a random peer in the swarm of the block's range. If the request is rejected, then another random peer is chosen from the swarm. If max_p requests are rejected, then the block is fetched from the server.

As in BitTorrent's tit for tat policy, clients maintain a counter which serves as a *credit balance* with each of their peers. A peer's balance is incremented whenever a block is received from that peer, and decremented whenever a block is sent to that peer. Clients serve peer requests that hit in their cache only if that peer's balance is within limits. A non zero *account limit* allows peers to initialize cooperation in the first place, as the receiving peer may accumulate some negative credit. Similarly, a peer may accumulate positive credit to use for future requests.

Cache management is based on Demote, and peer requests move blocks to the MRU position in a client's cache. Blocks received from peers are stored in a private partition similar to that in C-DHT. Thus, replication is limited by the size of the private partition. In that sense, clients participate in a global

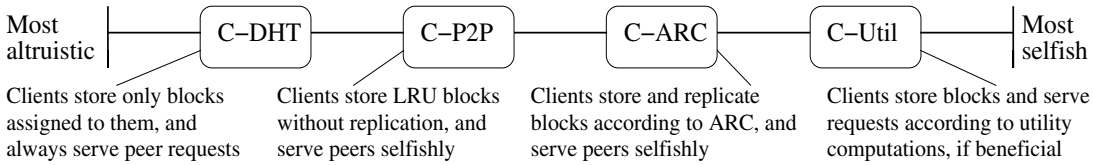


Figure 4. Cooperative policies on the scale of selfishness.

optimization when they cooperate. The metadata of a block received from a peer is stored in a dedicated *ghost cache*. If a block misses in the client cache but hits in its ghost cache, then the first REQUEST is sent to the peer that previously supplied the block. The size of the ghost cache equals that of ARC.

D. Cooperative ARC

C-ARC is constructed for strictly *selfish* clients and minimal server involvement. Clients store blocks in $L1$ or $L2$ according to ARC, without distinguishing between blocks received from peers or from the server. When a peer request for a block hits in $L2$, it does not alter the block's LRU position. However, when a block from $L1$ is sent to a peer, it is discarded. This serves as extending the trial period to all the peers, while avoiding duplication of blocks that are not necessarily useful. We consider the clients in C-ARC more selfish than those in C-P2P because they maintain control of their cache content, replicating their own useful blocks.

Clients use the ghost cache maintained by ARC to store information about the peers that previously supplied recently evicted blocks, and attempt to request these blocks again from the same peer. If the request misses in the ghost cache, or if this peer no longer stores the block, a random peer is chosen to request the block from. As in C-P2P, max_p attempts are made to receive the block from random peers before the block is fetched from the server. Clients maintain a credit balance with each of their peers, as in C-P2P, and serve peer requests that hit in their cache as long as this balance is within the predefined account limit.

E. Utility Based Cooperative Caching

C-Util represents the rightmost end of the selfishness scale (Figure 4), where clients use explicit utility calculations to cooperate only when their performance is guaranteed to improve. Clients use information, hinted or derived, about their future accesses, and are willing to share it with a trusted central server. The cooperation component at the server uses this information to construct a *configuration* that determines, for all clients, which blocks to store and which peers to serve. Each client is associated with an account balance, which is updated upon serving or receiving blocks. The cooperation component ensures all participating clients benefit from cooperating by keeping their balances close to zero. The configuration is updated periodically according to measured behavior or client updates.

Initialization. Cache management in C-Util is based on MC^2, and assumes that each client C can estimate, for each block range R it accesses, $F_C(R)$, the frequency of accesses of C to blocks in R in a given time frame[3]. Initially, each client computes the *utility* it can derive from storing ranges in its cache without cooperating with others. Namely, the utility from range R is $C_{storage} \times F_C(R)$, the cost of storage accesses saved by storing R. The *base utility* of a client is the sum of utilities from the ranges this client would store in its cache according to MC^2. Clients agree to cooperate only if their *expected* utility is greater than their base utility.

The client access frequencies are communicated to the server, which initializes a non-cooperative *base configuration*, where the cache content of all clients is determined according to MC^2. The base configuration serves as an initialization for the construction of a cooperative configuration according to the following guidelines.

Balanced Credit. Each client is associated with a counter, incremented whenever this client serves a peer request and decremented whenever it receives a block from a peer. This credit balance is maintained without distinguishing which peers the client serves. A very high credit balance indicates that the corresponding client has invested too much of its resources in helping others without getting enough in return, and is therefore undesirable. A very low (negative) credit balance is also undesirable because it indicates that the corresponding client has enjoyed the help of others without contributing its own resources. Therefore, the cooperation component keeps client balances within a predefined account limit, attempting to keep them as close to zero as possible.

Maximal Utility. $\langle C, R, P \rangle$ is a *cooperation transaction* in which client C agrees to store range R and serve all requests for blocks in that range originating from peer P. The utility of P from this transaction is $(C_{storage} - C_{net} - C_{send}) \times F_P(R)$, corresponding to the cost of storage accesses saved by receiving R from C, less the cost of receiving R from C and the cost of performing a corresponding amount of SERVES in order to accumulate the credit necessary for the transaction. The utility of C from the transaction is identical. Although in the current transaction C only performs work, it accumulates credit that it will use in another transaction as the receiving peer. Thus, when the value of the credit balance is taken into account, and as long as balances are guaranteed to stay within limits, both the serving client and the receiving peer benefit from cooperation, and their utilities increase. The cooperation component attempts to maximize the *global utility* – the sum of all client utilities.

The cooperation component greedily constructs a cooperative configuration, by iteratively adding cooperation transactions to the base configuration. The final configuration is communicated to the clients in the form of several bit arrays. Clients populate their caches *lazily* according to the configuration, fetching blocks only when they are requested by peers or accessed by their application. Clients REQUEST blocks from peers and SERVE peer requests according to the cooperation transactions specified by the configuration. A high level description of this process appears in Figure 5.

[3]The application hints used in MC^2 can be converted to these frequencies by normalizing them according to the client computation speeds. They can also be updated periodically according to the observed relative access frequency of the clients.

Greedy construction step
01. C = client with lowest account balance (within limits)
02. R = range with lowest $F_C(R)$ stored by C
03. P = peer with highest account balance (within limits)
04. R' = range with highest $F_P(R')$ not stored or received by P
05. **if** $(F_P(R) > 0)$ // P needs R
06. **if** (P does not store or receive R) **or**
07. ((P stores R) **and** // benefits from saving R' instead:
08. $(C_{storage} \times F_P(R') > (C_{net} + C_{serve}) \times F_P(R))$)
09. add $\langle C, R, P \rangle$ to configuration
10. update account balances
11. // continue to next P, R, C until all options are exhausted.

Client C accesses block X of range R:
12. Cache hit:
13. update stacks
14. Cache miss:
15. **if** C should store R in current configuration
16. **if** $(\exists P, \langle P, R, C \rangle \in$ previous configuration)
17. REQUEST X from peer P
18. **else** // or if request was rejected
19. READ X from server
20. store X in cache
21. **else**
22. **if** $(\exists P, \langle P, R, C \rangle \in$ current configuration)
23. REQUEST X from peer P // P must SERVE
24. **else**
25. READ-SAVE X from server // X will not be stored

Figure 5. High level pseudocode for greedy construction, lazy population and cooperation protocol in C-Util.

The cooperation component updates the configuration when a client provides new information which alter its utility or whenever a client exceeds its credit limit. This can happen because the balances are rarely exactly zero, and positive or negative credit is accumulated in client accounts. Another reason is inaccuracy of the access frequencies, causing clients to serve more requests than intended by the cooperation component. The new configuration is constructed greedily according to the current credit balances. The clients update their cache content in a lazy manner, similar to the initial population.

IV. Evaluation Methodology

In this section we describe the methodology used for our analysis. We used trace driven simulations to evaluate and compare the caching approaches described above. We use two different measures to evaluate their performance, on four scenarios of data sharing and access patterns.

A. Workloads

We experimented with several workload characteristics. In *uniform* workloads, block accesses follow uniform distributions, whereas in *skewed* workloads, accesses exhibit non-uniform, "long tail" distributions. *Stable* workloads exhibit the same access pattern for a long period, while *dynamic* workloads change over time. In *correlated* data sharing, different clients simultaneously access the same blocks with similar access patterns and distributions. In *non-correlated* sharing, clients access the shared data at different times or with different access characteristics.

Workload	TPCH queries	TPCH sets	OLTP	Video
Correlation	varied	medium	strong	medium
Access type	uniform	uniform	skewed	skewed
'Hint' accuracy	perfect	short term	low	medium
Stability	stable	dynamic	stable	dynamic
Num. requests	400K-6M	14M	5M	1.8M
Unique blocks	36K-202K	220K	45K	770K
Num. clients	2-4	2-20	2-20	50

TABLE I. Summary of workload characteristics

Database I/O traces. We first consider a scenario in which one data set is accessed by several applications, possibly launched by different departments or end users. TPCH [39] is a decision support benchmark, where queries access large amounts of data uniformly. In our first workload, each client runs a different query, executing multiple times, with different search parameters. We experimented with 7 combinations of 2 to 4 clients and queries, with varying degrees of data correlation between the queries. In the second workload, clients run all 22 queries in the TPCH benchmark, but in a different order, representing dynamic data sharing, where access patterns and pattern combinations change continuously, and blocks turn from *hot* to *cold* and vice versa.

We also consider a scenario of online transaction processing (OLTP), such as in web auctions [40] or other e-commerce applications. In our third workload, clients run the TPCC [41], [42] benchmark. Utility information for MC^2 and C-Util was generated in the form of application hints, using the database *explain* mechanism [43]. Table I summarizes the characteristics of our traces.

Video playback. Our last workload is composed of viewing requests to video sharing Web sites, such as YouTube. The traces were collected over a period of one month, at a medium sized ISP in Latin America (not exposed for reasons of privacy and business sensitivity), serving both private and business customers. We used request traces of the 50 most active IP addresses. The duration of the video was not always available in the traces, so we used the average values of 5 minutes and 512 Kbps for all the videos. Clients have an upload bandwidth of 1 Mbps, as in the traced ISP's network. Although in practice the bandwidth between the server and the clients or the Internet is limited, we assumed, for simplicity, that the server can serve all clients concurrently.

Utility information for MC^2 and C-Util is based on viewing history, and is generated on the fly, without relying on external hints. We assumed the ISP cache collects access statistics to the most popular videos. At the beginning of each week, the server calculates the access frequencies based on statistics collected on the last day of the previous week, for 3% of the most accessed videos on this day. These frequencies *estimate* the future aggregate access distribution at the ISP, and are not necessarily accurate for each individual client.

B. Objective Functions

We use two different objective functions that suit our workloads. One quantifies the delay experienced by the client, while the other quantifies the Internet accesses it incurs. We

explain how the costs defined in Section II are computed for each of the objective functions.

Time. To compute the I/O response time and I/O delay of database clients, we express the costs in terms of time.

– T_{net} is the time it takes to transfer a data block from one cache to another. It represents a combination of computation, network, and queuing delays.
– $T_{storage}$ is the average time spent waiting for a disk access, including queuing and seek times. A block request from the disk may complete in less than $T_{storage}$ if the block was already being fetched when it was queued.
– T_{serve} is the time the CPU spends sending a block to a peer. During this time, the client cannot perform computation. A client experiences a delay $< T_{serve}$ if it is idle, waiting for I/O, when the request arrives.

Throughout our evaluation we assume that $T_{serve} < T_{net} < T_{Disk}$. For our basic setup we assign $T_{serve} = 50\mu secs, T_{net} = 200\mu secs$ and $T_{Disk} = 5msecs$, corresponding to a local network transfer of an 8KB block and the average seek time of a SAS disk. We elaborate on more setups in Section V-G.

Internet access. In the context of video playback, average I/O response time is irrelevant – the video's bit rate determines when each frame is played, and data is buffered in advance. Therefore, for the video workload, we measured the portion of requested videos served from the Internet, as well as the upload bandwidth used by the clients.

For utility calculations in C-Util, we used the relative operational costs of the ISP where the traces were collected: bandwidth within the ISP network (between all clients and between the clients and the server) was 100 times cheaper than the connection to the Internet. Although these costs represent the objective of the ISP and not of the individual clients, we assume they can be incorporated into the clients' service level agreements. Therefore, it is reasonable to base the clients' selfish decisions on this metric. Thus, $C_{net} = 1$ and $C_{storage} = 100$. $C_{serve} = 0$, because the transfer within the ISP network is accounted for in C_{net}. In addition, *uploading* a video to a peer does not affect the client's viewing performance. However, the client's limited upload bandwidth (1 Mbps) limits the number of concurrent peers it can serve.

C. Simulator

We used a simulation environment to experiment with a variety of storage configurations, described in detail in Section V. We used a simulator that was used in previous studies [35] and manages I/O event queues for all clients and caches in the system. It computes the time each event spends in each queue, taking into account the interactions between requests in different queues.

We ran our simulations with a wide range of client cache sizes. We assumed that all clients have the same cache size. For the database traces, the size of the server cache is the same as one client cache, and the sizes are expressed as a fraction of the data set size in each experiment, to capture the interaction between the two. For the video traces, we fixed the server cache size at $\frac{1}{4}$ GB per client, and varied the size of the client caches from $\frac{1}{4}$ GB to 8 GB. Note that in the context of Web caching, this storage need not necessarily be entirely in RAM.

V. Evaluation

Clients that serve peer requests invest work, and possibly valuable cache space, hoping to avoid expensive disk or Internet accesses. Our goal is to evaluate whether cooperative caching is helpful with selfish clients, and the effect the degree of selfishness has on performance.

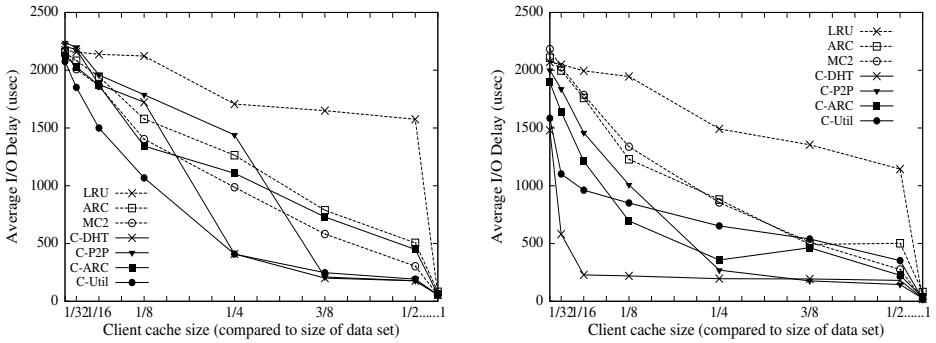
A. TPCH query sets workload

Figure 6 shows the average I/O delay incurred by the different policies on the TPCH sets workload (due to space limitations, we present here results only for setups with 2 and 20 clients). All clients run the same set of TPCH queries in different order, resulting in a dynamic mix of workloads. Clients recalculate their utilities whenever their workload changes. Since the changes in workload are small and there is considerable data sharing between the clients, cooperation always improves performance. However, the different approaches exhibit different behavior.

When the consolidated client cache is smaller than the data set (small client caches or few clients), selfish clients perform better because they keep their most important blocks in their private cache. C-Util, the best approach in this case, reduces the average I/O delay by as much as 32% for 2 clients and a cache size of $\frac{1}{8}$. Without utility calculations, C-ARC provides the best cache management, reducing the average I/O delay of ARC by as much as 15%. With large caches, clients cooperating altruistically perform better because they manage to fit the entire data set in the consolidated client cache, thanks to exclusivity. C-DHT reduces the average I/O delay of ARC by as much as 87%, with caches of size $\frac{1}{16}$ and 20 clients.

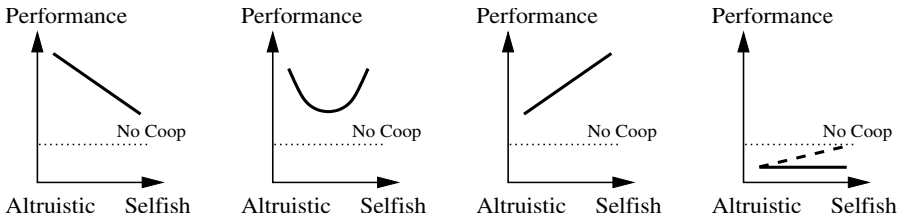
Selfish clients fail to achieve perfect exclusivity with large caches. The main reason is that clients duplicate frequently accessed blocks and don't rely on their peers to supply them. In C-ARC this also interferes with locating less frequently accessed blocks – a peer that served them before is not obligated to store them for future accesses. Additionally, C-P2P and C-ARC are limited by their credit balance. A client that accumulates credit with some peers is unable to "cash" it if different peers hold the blocks it needs, and has to replicate those blocks in its own cache. The aggregate credit balance and the configurations constructed by the server in C-Util address these issues.

In contrast, clients in C-Util refuse to cooperate when they think they have nothing to gain. A client will decline a cooperation opportunity if the blocks it can fit in its cache as a result are not valuable enough. In the dynamic query set workload, clients do not wish to store blocks they will access only when their workload changes. With a cache of size $\frac{1}{16}$ and 20 clients, the benefit of C-Util, the best selfish approach, is 41% less than the benefit of the altruistic C-DHT. This limitation can be addressed in two orthogonal ways. First, if the application can supply hints about future workloads, the client can incorporate them into its utility function. Second, the system can pay "extra credit," i.e., by means of SLA benefits, to clients that store blocks they don't currently need. Such



(a) 2 clients (b) 20 clients

Figure 6. TPCH query sets. With uniform, dynamic workloads, cooperation is always good. Exclusivity and cache replacement are the dominant factors with large and small caches, respectively.



(a) TPCH sets (b) TPCH queries (c) Video (d) OLTP

Figure 7. Schematic representation of the effect of client selfishness on performance with different workload characteristics: (a) uniform distribution, medium correlation (b) stable, uniform distribution (c) heavy tail distribution, medium correlation (d) non-negligible SERVE and high access skew.

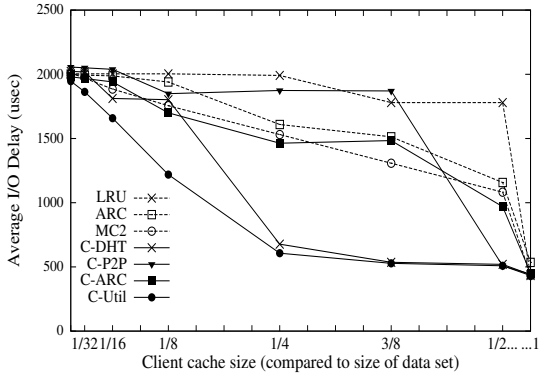


Figure 8. TPCH queries 3, 10, and 18. With predictable, stable workloads, cooperation is always good. Exclusivity and cache replacement dominate with large and small caches, respectively. C-Util achieves both.

incentive schemes are addressed by Algorithmic Mechanism Design [44].

Figure 7(a) shows, schematically, that cooperation is always beneficial with large caches and uniform, strongly correlated workloads. Our results show that client selfishness limits the benefits from cooperation in such scenarios, corresponding to Rule 1 in Figure 1.

B. TPCH query combinations workload

In our last workload, each client runs one query repeatedly, with varying parameters, as explained in Section IV-A. In such a scenario, the access frequencies are highly accurate, and the stability of the working sets provides good opportunity for cooperation. Figure 8 shows the average I/O delay incurred by the different cache management policies on three clients running TPCH queries 3,10 and 18. These queries access the same database tables with different priorities and access patterns.

Cooperation almost always improves performance with this workload, with different behaviors exhibited by different caching approaches. C-DHT and C-Util achieve the best performance improvements with exclusive caching. In C-DHT, exclusivity results from dividing the hash key space statically between clients. In C-Util the server constructs exclusive configurations, with minimal replication of very popular blocks. Similar to the query set workload, this replication is especially important when caches are small. The other policies, C-P2P and C-ARC achieve significant improvement over their non-cooperative versions only when the caches are almost as large as the entire data set. Their selfishness, combined with lack of central management, causes excessive data replication.

Figure 7(b) depicts the trend we observed in this workload; the most selfish and the most altruistic approaches achieve the best results for non-correlated workloads – these approaches include explicit block allocation and do not depend on correlation between clients. Cooperation with approaches in the middle of the scale that lack explicit block allocation, provides only modest performance improvements. While each query combination resulted in a different access pattern and priority mix, the overall conclusion from all combinations (summarized in Figure 9) was the same, corresponding to Rule 2 in Figure 1.

C. Video workload

Figure 10(a) shows the portion of videos that can be served without accessing the Internet, when the different caching approaches are used. Cooperation in the video workload does not affect viewing performance, since it does not utilize the client's download bandwidth. Additionally, viewing a video directly from a peer cache is equivalent to viewing it from the server, as long as bandwidth limitations are not exceeded. Therefore, cooperation is always beneficial in this workload. The selfish C-Util is the best cooperative approach, improving its relative performance as cache sizes increase. It achieves a hit rate as high as 32% with 8 GB – an increase of 10%,

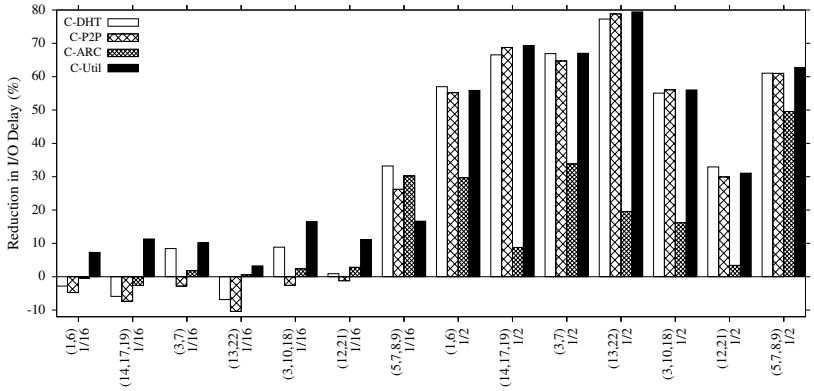
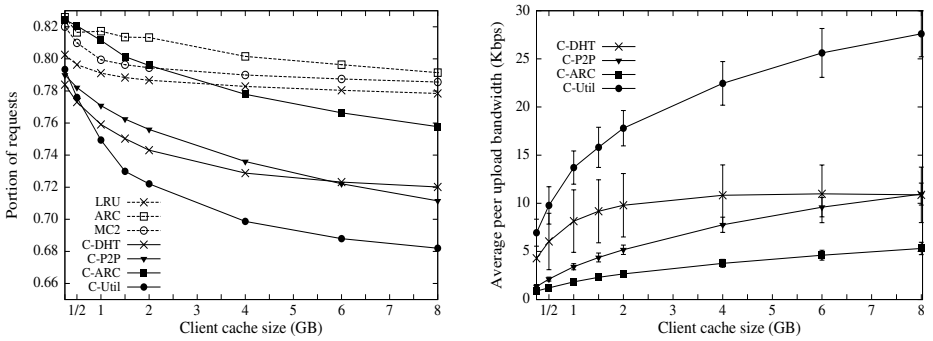


Figure 9. Performance of cooperative policies on TPCH query combinations when compared to the best non-cooperative policy. The combinations are ordered according to the difference between the queries, from almost identical access patterns (1,6) to different data ranges and different access patterns and block priorities (5,7,8,9). We present results for two representative cache sizes: $\frac{1}{16}$ and $\frac{1}{2}$. Negative numbers represent an *increase* in I/O delay.



(a) Internet accesses (ISP misses)

(b) Average client upload

Figure 10. 50 clients viewing videos. Cooperation reduces the portion of requests that incur an Internet access (a), while incurring modest load on client upload bandwidth (b). The error bars depict the standard deviation, demonstrating that credit balances are useful for evenly distributing the load on the clients.

14% and 44% over the hit rates of C-P2P, C-DHT, and LRU, respectively[4].

C-DHT performs well with small caches, but its performance remains constant even when client cache sizes increase beyond 4 GB. This is the result of combining altruistic behavior with uncorrelated accesses. At 4 GB, the popular videos are already stored in the cumulative client cache, but unpopular videos fail to "enter" it: unpopular videos are often not accessed by the client responsible for them, and are therefore repeatedly fetched from the server.

Figure 10(b) shows the average upload bandwidth consumed by individual peers. The error bars, depicting standard deviation, show the load distribution between clients. The per-peer credit balances in C-P2P ensure even distribution, but also limit cooperation opportunities. C-Util is more susceptible to imbalance, but, surprisingly, achieves better load balancing than the hash keys in C-DHT. Clients using C-Util selfishly replicate extremely popular videos, while in C-DHT, arbitrary peers become "hot spots."

With small caches ($\frac{1}{4}$-$\frac{1}{2}$ GB), the roles are reversed. Small caches can store only the most popular videos, which are easily recognized by LRU. C-DHT is the best cooperative policy in this situation, outperforming C-Util. Although the most popular videos are identified by the access frequencies, the workload changes faster than the frequencies are updated. Videos which are most popular when utilities are computed thus become less popular as new videos appear in the workload, but still consume a large portion of the cache (this also explains MC^2's poor performance on this workload). Larger caches mask this "error" and store both old and new popular videos.

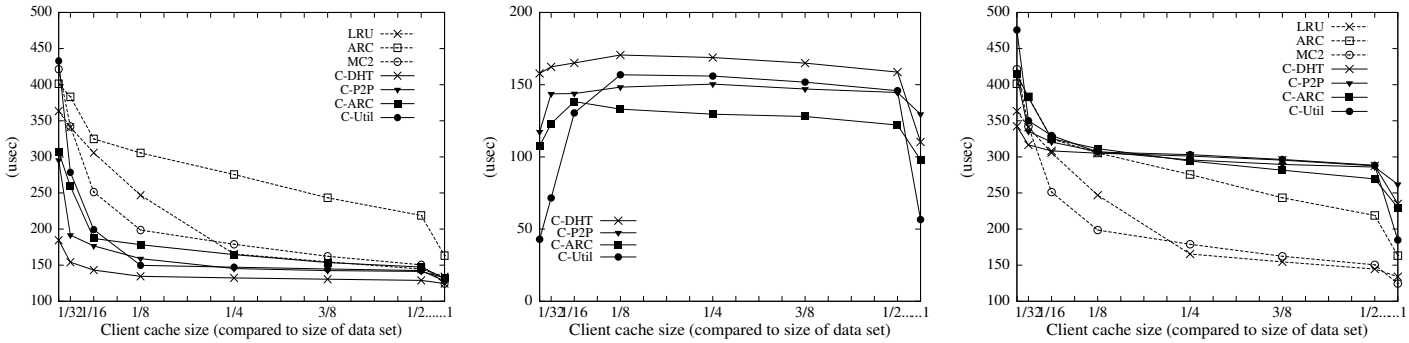
Figure 7(c) schematically shows that cooperation is always helpful when its cost is negligible, even with long tail distributions and medium data correlation. Selfish considerations improve performance in this case, corresponding to Rule 3 in Figure 1.

D. OLTP workload

Figure 11(a) shows the reduction in I/O response time achieved by cooperation between 20 clients running the OLTP workload (the results are similar for fewer clients). This workload has great cooperation potential – clients run the same workload continuously, so their data set is stable, and they agree on block priorities. Thanks to the skew in block accesses and strong temporal locality, all cooperative approaches reduce the average I/O response time. Clients gain access to a larger portion of the data without accessing the disk. However, this measure does not take into account the work clients invest in serving peers.

The cost of serving peers, the service delay, is depicted in Figure 11(b). Comparing Figure 11(b) to Figure 11(a), it is evident that the time saved by avoiding the disk accesses is less than that spent serving other peer requests. The workload skew causes the space freed in client caches to be used for

[4]ARC, and subsequently, C-ARC, are specifically optimized for storage I/O, explaining their inferior performance on these traces. We present their results for completeness, but omit them from the discussion.



(a) Average I/O response time (b) Average service delay (c) Average I/O delay

Figure 11. 20 clients run the OLTP workload, with high skew and data correlation. Cooperation reduces the average I/O response time (a) at the price of long service delays (b). The average I/O delay (c) increases with all cooperative policies.

infrequently accessed blocks, whose utility does not mask the cost of serving popular blocks to peers.

The average I/O delay takes this cost into account. Figure 11(c) shows that the cooperative policies almost double the I/O delay of LRU in the large cache sizes. This demonstrates the importance of explicitly addressing the cost of service when evaluating the benefits from cooperation.

The solid line in Figure 7(d) represents our conclusion that with non-negligible cost of SERVE and high access skews, cooperation always degrades performance. However, the similarity between altruistic and selfish approaches is counter intuitive. We expected the behavior to resemble that depicted by the dashed line, indicating that selfish clients refuse to cooperate if their utility decreases.

Specifically, the utility calculations in C-Util were supposed to help it detect that cooperation is not beneficial with OLTP (Figure 11(c)). However, when block ranges are accessed in a nonuniform distribution, the overall utility of the range does not accurately reflect the utility of each of its blocks. With MC^2, clients selectively store only the more frequently accessed blocks within such ranges [45]. However, with C-Util a client agrees to be responsible for a range, and is obligated to store its low priority blocks as well, at the expense of its own performance. Rule 4 in Figure 1 states that when non-uniform accesses are combined with inaccurate hints and non-negligible cost of SERVE, cooperation should be avoided. To detect such situations, clients relying on application hints may use online statistics, and refuse to cooperate until their workload changes.

E. Lookup Mechanisms

Lookup mechanisms for distributed caches have been suggested in previous studies [14], [15], [18], [46], and were not the focus of our evaluation. However, insufficient lookup capabilities limit cooperation. While C-DHT and C-Util have built-in mechanisms that do not incur additional overhead, cooperation with C-ARC and C-P2P is limited by max_p, the number of peers a client is allowed to query before requesting a block from the server. When we increased max_p, their performance improved continuously. The average I/O delay of C-ARC and C-P2P decreased by as much as 63% and 76%, respectively, when max_p equaled the number of peers. Querying all the peers in the system is clearly infeasible in a practical implementation. Instead, such policies should be augmented by some external mechanism to fully utilize the cumulative caches.

F. Account Limits and Selfishness

The willingness of clients to cooperate is determined by their credit balance limit. Recall that clients using C-P2P and C-ARC SERVE peer requests only if their corresponding credit balance is within limits. In C-Util, the aggregate credit balance maintained by each client restricts the cooperation transactions added to the configuration. We varied the aggregate account limits in order to evaluate their effect on performance. As we increased the balance limit, we expected performance to improve as a result of additional cooperation opportunities, and the load balance to degrade as a result of increased credit accumulation. This was indeed the case in almost all policies and workloads, although the effect was minor. The difference in average I/O delay was 0%, 1%, and 3% on average, with C-ARC, C-P2P and C-Util, respectively.

With C-Util, increasing balance limits *improved* the load balance, because clients had more opportunities to "cash" the credit they accumulated. However, in some cases, the I/O delay *increased* because, due to the insufficient accuracy of the access frequencies, clients agreed to cooperate on less valuable blocks, whose utility did not mask the additional cost of serving peers.

Account balances have a fundamental role as incentives for selfish clients to cooperate. Their non-zero limits allow clients to initiate their cooperative transactions. In existing peer-to-peer systems, proper incentives are the main contributor to the overall system's performance [47], [48]. In contrast, our analysis shows that in the context of cooperative caching, the precise value of account limits has little effect on the performance of most cooperative approaches, especially when compared to cache management and lookup capabilities. Thus, an existing peer-to-peer mechanism cannot be applied "as is" to a cooperative caching system – it will not guarantee performance improvements without a suitable cache management algorithm.

G. Storage Access Costs

The benefit from cooperation between clients, like that of other techniques that incur space [34] and transfer [36]

overhead, is sensitive to the storage access cost. We evaluated this sensitivity with several storage setups. While the rest of our results are for a high performance SAS disk, here we consider a SATA disk drive, representing long access delays, an SSD drive, representing short delays, and a controller with DRAM storage, representing high-end fast storage access [49]. Figure 12 shows the performance of the best two cooperative approaches on these setups.

We expected the benefit from cooperation to increase with $T_{storage}$. Indeed, with the SATA drive all policies benefit more from cooperation than they do with SAS. In fact, since T_{serve} was so much smaller than $T_{storage}$, the cooperative policies were able to improve performance even with the OLTP workload (not presented for lack of space), corresponding to Rule 3 in Figure 1.

We also expected some benefit from cooperation with faster storage, as long as $T_{serve} < T_{storage}$. In practice, all cooperative policies except C-Util *increased* the average I/O delay, for all our workloads, both with SSD and with DRAM. Cooperation improves exclusivity and frees up cache buffers, but these buffers are populated with blocks that are accessed less frequently. Thus, disk accesses saved are fewer than the number of peer REQUEST and SERVE operations, canceling any benefit from cooperation. C-Util suffers less as a result of decreases in $T_{storage}$ because clients estimate their utility based on the access costs and selfishly avoid cooperation in those setups.

H. Upload Bandwidth

To evaluate the effect of the upload bandwidth bottleneck, we varied it from 1 Mbps to 4 Mbps. Recall that serving a peer request consumes 512 Kbps of the client's upload bandwidth for the duration of the video. At 2 Mbps, the average load on the peers increased by a maximum of 0.15%, 2%, and 9% for C-DHT, C-P2P and C-Util, respectively, while the number of Web accesses decreased by 0.01%, 0.1% and 1%. Further increasing the bandwidth to 4 Mbps had no effect. We conclude that while upload bandwidth was a bottleneck for C-Util, the dominating factor in the performance of C-DHT and C-P2P was the hit rate in the client caches.

VI. Other Design Considerations

Write traffic. Data sharing between multiple clients introduces conflicts when the workloads include WRITE traffic, requiring explicit cache coherency protocols to ensure correctness. Cooperative caching in itself does not introduce additional coherence issues. The precise effect of write traffic on cooperative approaches is out of this paper's scope. However, some are more suited to handle write traffic than others.

Approaches with built-in lookup mechanisms which ensure exclusivity, such as C-DHT and C-Util, can easily be augmented with a cache coherency protocol – a client responsible for a group of blocks can be in charge of their master copies. Policies such as C-P2P, which allow replication only through access to the shared cache, do not introduce coherency issues beyond those of a shared server cache. However, clients that selfishly replicate SERVED blocks, as in C-ARC, present an additional challenge.

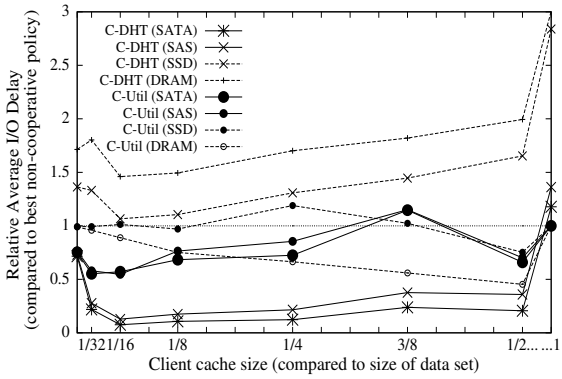


Figure 12. Increasing storage access costs increases the benefit from cooperation. 20 clients run the TPCH query sets with SATA, SAS, SSD, and DRAM drives, where $T_{storage}$ equals $10ms$, $5ms$, $260\mu s$ and $120\mu s$, respectively. T_{serve} is $50\mu s$. We omit the network cost because it is also incurred by requesting a block from a peer.

Multiple Servers. Clients in large, consolidated systems, may access data from multiple storage servers with heterogenous access costs. The cache replacement mechanism in such clients must take these costs into account when deciding which block to evict. This is equivalent to allocating a separate cache partition to blocks from each server, and dynamically adjusting the sizes of these partitions [50]. While the replacement decisions alone are orthogonal to the cooperation protocols, clients in all cooperative approaches should have some means of identifying peers that access the same servers they do.

Clients that adhere to a configuration constructed by the server, as in C-Util, cannot adjust their partition sizes independently. Instead, we suggest that partitions be adjusted only at the end of predetermined time frames. Each server will construct its configuration for the next time frame according to the adjusted size of its respective partitions. The configurations of the different servers do not interfere with one another – they guarantee utility in independent cache partitions. Clients may use these utilities to determine their partitioning in subsequent time frames.

Trust. Clients in our model trust their peers in terms of data integrity and adherence to caching and cooperation protocols. In realistic environments, malicious behavior can have a detrimental effect on cooperating clients. There, existing protection mechanisms can be added to the cooperative approaches we presented. For example, Shark [51] uses encryption and opaque tokens to enable cooperation, CloudViews [6] suggests unforgeable tokens to ensure privacy in data sharing scenarios, and credit balances can prevent *free riding*, where peers maliciously receive service without contributing their own resources [20].

VII. Related work

Traditional storage caching approaches assumed altruistic clients in centrally managed systems. Most of them consider the global system's optimization as the only objective [15], [17], [18]. Some exploit locality of reference within a client's workload by allowing the clients to manage a private LRU partition [14], [38]. None of the above approaches consider the delay incurred by a client serving peer requests. Our results

show that this delay may mask, or even exceed, the delay saved by cooperation.

Recent cooperative caching studies address load balancing between clients. Experiments with the Shark network file system [51] show significant differences in the bandwidth served by different proxies. In NFS-CD [52], the load of serving popular files is split between "delegates" when it is detected. LAC [16] equalizes cache utilization between all clients. Similarly, distributed storage systems [37], [53] and NoSQL databases [54] use hash keys to evenly distribute replicas and load between servers.

Load distribution, or "fairness", is not enough in heterogenous environments, where cache owners are autonomous and have selfish objectives. Customers that have paid for resources are expected to share them only within frameworks that guarantee "return on investment", and are likely to prefer platforms that guarantee their objectives are met. We have demonstrated that cooperative caching is possible and beneficial despite these limitations, as long as the selfish objectives are considered explicitly.

Peer-to-peer (P2P) systems constitute a large number of independent, autonomous peers, and an incentive mechanism which motivates peers to invest resources in serving others. System efficiency is evaluated by measuring its global throughput. In BitTorrent [20], peers upload data to other peers in a *tit for tat* manner. The *reputation* system in [22] enhances this mechanism. Alternatively, currency and balance based mechanisms are used for packet forwarding [23] and multicast [21] in ad hoc networks.

In such systems, peers cooperate according to their location or group assignment. However, these are short-term cooperative transactions that usually involve a single operation. Our analysis shows that incentives alone do not guarantee performance benefits in stateful systems such as caches. To benefit from cooperation, clients must use a suitable cache management algorithm that allows them to evict blocks from their cache, and rely on their peers to store these blocks and provide them when requested, for an agreed upon period of time.

Auction based mechanisms were suggested for managing resources in distributed systems [30]. Similarly, economic models for setting the price of services on a supply-and-demand basis were suggested for job scheduling in grid systems [3]. These models capture the interaction between conflicting objectives of different entities, and allow for long-term agreements. However, the resulting mechanisms are prohibitively difficult to implement and carry significant computation and network overheads. Similarly, other models such as cooperative game theory or detailed market models, are generally computationally hard, and do not guarantee a stable system state [24]. In contrast, the selfish cooperative approaches we presented provide sufficient incentives for selfish caches to cooperate while employing only simple or greedy heuristics.

VIII. Conclusions and Future Work

Resource sharing is becoming increasingly popular, entailing significant benefits for users in large scale distributed systems. Cooperative caching has long been suggested as a means to increase cache utilization and improve performance. In consolidated systems, and particularly in infrastructure-as-a-service clouds, it can also reduce operational costs by enabling dynamic resource scaling. We introduced a new model for cooperative caching in such environments, where clients are selfish and cooperate based on their expected return on investment.

We proposed four cooperative approaches for clients of varying degrees of selfishness. Our analysis shows that when applied correctly, cooperative caching can greatly improve performance in many system and workload combinations. However, in some scenarios, imposing cooperation can significantly degrade performance. We summarize our findings in the form of basic guidelines for identifying *when* caches should cooperate, and *how*.

Our guidelines and basic cooperation schemes can be combined into a general management policy that will automatically detect whether cooperative caching can improve performance, and recommend (or apply) the most suitable scheme according to the workload and system characteristics. This high level policy can also detect workload and topology changes, and instruct clients how to adjust their behavior accordingly.

Acknowledgments

We wish to thank Gabi Kliot, Zvika Guz and Michael Shapira for valuable discussions. We also thank Ofir Amir from Allot for his insightful comments. Jenya Pergament helped with the workload traces. This work was supported in part by the Israel Ministry of Industry, Trade, and Labor under the Net-HD Consortium, and by Microsoft. Gala Yadgar's work was supported in part by the Levi Eshkol scholarship from the Israel Ministry of Science, and by a PhD scholarship from the Hasso Plattner Institut.

References

[1] L. N. Bairavasundaram, G. Soundararajan, V. Mathur, K. Voruganti, and S. Kleiman, "Italian for Beginners: The Next Steps for SLO-based Management," in *USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[2] J. Wilkes, "Traveling to Rome: a retrospective on the journey," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 10–15, January 2009.

[3] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in Grid computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1507–1542, 2002.

[4] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal, "Pesto: Online Storage Performance Management in Virtualized Datacenters," in *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[5] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *ACM SIGOPS Operating Systems Review*, vol. 43, pp. 92–105, January 2010.

[6] R. Geambasu, S. D. Gribble, and H. M. Levy, "CloudViews: Communal Data Sharing in Public Clouds," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[7] M. Mihailescu, G. Soundararajan, and C. Amza, "MixApart: Decoupled Analytics for Shared Storage Systems," in *USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2012.

[8] B. Niven-Jenkins, F. Le Faucheur, and N. Bitar, "Content Distribution Network Interconnection (CDNI) Problem Statement," IETF, Internet-Draft, January 2012.

[9] D. Rayburn, "Telcos and carriers forming new federated CDN group called OCX," StreamingMedia, June 2011.

[10] C. Osika, "Cisco Keynote: The Future Is Video," in *CDN Summit*, May 2011.

[11] Y. Chen and R. Sion, "To Cloud Or Not To Cloud? Musings On Costs and Viability," in *ACM Symposium on Cloud Computing (SOCC)*, 2011.

[12] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The Resource-as-a-Service (RaaS) Cloud," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[13] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. Kozuch, "Saving Cash by Using Less Cache," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[14] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," in *USENIX conference on Operating Systems Design and Implementation (OSDI)*, 1994.

[15] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, "Implementing Global Memory Management in a Workstation Cluster," in *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 1995.

[16] S. Jiang, F. Petrini, X. Ding, and X. Zhang, "A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance," in *International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[17] M. R. Korupolu and M. Dahlin, "Coordinated Placement and Replacement for Large-Scale Distributed Caches," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, pp. 1317–1329, 2002.

[18] P. Sarkar and J. Hartman, "Efficient Cooperative Caching Using Hints," in *USENIX conference on Operating Systems Design and Implementation (OSDI)*, 1996.

[19] B. Tang, H. Gupta, and S. Das, "Benefit-Based Data Caching in Ad Hoc Networks," in *IEEE International Conference on Network Protocols (ICNP)*, 2006.

[20] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Workshop on Economics of Peer to Peer Systems (P2PECON)*, 2003.

[21] I. Keidar, R. Melamed, and A. Orda, "Equicast: Scalable Multicast with Selfish Users," in *Annual Symposium on Principles of Distributed Computing (PODC)*, 2006.

[22] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson, "One Hop Reputations for Peer to Peer File Sharing Workloads," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[23] S. Zhong, J. Chen, and Y. R. Yang, "Sprite: A Simple, Cheat-Proof, Credit-Based System for Mobile Ad-Hoc networks," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2003.

[24] V. V. Vazirani and M. Yannakakis, "Market Equilibrium Under Separable, Piecewise-Linear, Concave Utilities," *J. ACM*, vol. 58, pp. 10:1–10:25, May 2011.

[25] "A Framework for Data Center Energy Productivity," The Green Grid, White Paper 13, 2008.

[26] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini, "Statistical Profiling-Based Techniques for Effective Power Provisioning in Data Centers," in *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2009.

[27] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," in *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 1995.

[28] B. S. Gill and D. S. Modha, "SARC: Sequential Prefetching in Adaptive Replacement Cache," in *USENIX Annual Technical Conference*, 2005.

[29] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch," in *USENIX Annual Technical Conference*, 2007.

[30] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, "CA-NFS: A Congestion-aware Network File System," *ACM Transactions on Storage (TOS)*, vol. 5, pp. 15:1–15:24, 2009.

[31] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger, "Using Utility to Provision Storage Systems," in *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[32] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," in *Cloud Computing Technology and Science (CloudCom)*, 2011.

[33] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[34] B. Gill, "On Multi-Level Exclusive Caching: Offline Optimality and Why Promotions are Better Than Demotions," in *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[35] G. Yadgar, M. Factor, K. Li, and A. Schuster, "Management of Multilevel, Multiclient Cache Hierarchies with Application Hints," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, pp. 5:1–5:51, 2011.

[36] T. M. Wong and J. Wilkes, "My Cache or Yours? Making Storage More Exclusive," in *USENIX Annual Technical Conference*, 2002.

[37] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, pp. 17–32, 2003.

[38] X. B. He, L. Ou, M. J. Kosa, S. L. Scott, and C. Engelmann, "A Unified Multiple Level Cache for High Performance Storage Systems," *International Journal of High Performance Computing and Networking*, vol. 5, pp. 97–109, 2007.

[39] "TPC Benchmark H Standard Specification, Rev. 2.1.0."

[40] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *USENIX Annual Technical Conference*, 2009.

[41] "TPC Benchmark C Standard Specification, Rev. 5.6."

[42] D. R. Llanos and B. Palop, "An Open-Source TPCC Implementation for Parallel and Distributed Systems." in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[43] G. Yadgar, M. Factor, and A. Schuster, "Karma: Know-It-All Replacement for a Multilevel Cache," in *USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[44] N. Nisan and A. Ronen, "Algorithmic Mechanism Design (extended abstract)," in *ACM Symposium on Theory of Computing (STOC)*, 1999.

[45] G. Yadgar, M. Factor, K. Li, and A. Schuster, "MC^2: Multiple Clients on A Multilevel Cache," in *International Conference on Distributed Computing Systems (ICDCS)*, 2008.

[46] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, pp. 281–293, 2000.

[47] E. J. Friedman, J. Y. Halpern, and I. Kash, "Efficiency and Nash Equilibria in a Scrip System for P2P Networks," in *ACM Conference on Electronic Commerce (EC)*, 2006.

[48] S. Jun and M. Ahamad, "Incentives in BitTorrent Induce Free Riding," in *Workshop on Economics of Peer to Peer Systems (P2PECON)*, 2005.

[49] "Kaminario K2 All Solid-State SAN Storage," 2011.

[50] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USENIX Symposium on Internet Technologies and Systems (USITS)*, 1997.

[51] S. Annapureddy, M. J. Freedman, and D. Mazières, "Shark: Scaling File Servers via Cooperative Caching," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[52] A. Batsakis and R. Burns, "NFS-CD: Write-Enabled Cooperative Caching in NFS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 323–333, March 2008.

[53] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[54] A. Lakshman and P. Malik, "Cassandra: a Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, April 2010.