Multilevel Cache Management Based on Application Hints

Gala Yadgar
Multilevel Cache Management Based on Application Hints

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Gala Yadgar

Sumbitted to the Senate of
Technion - Israel Institute of Technology

Iyar, 5772 Haifa May, 2012
This research thesis was done under the supervision of Prof. Assaf Schuster in the Department of Computer Science, and Dr. Michael Factor from IBM Haifa Research Labs.

First and foremost, I would like to thank my advisors, Prof. Assaf Schuster and Dr. Michael Factor. They have patiently guided me in my journey as a new researcher, assisting whenever possible, while allowing me to grow and practice my independence. They have complemented one another in their points of view, experience, and wisdom, and it was my privilege to work with both of them.

I would like to thank Prof. Kai Li for providing me with an additional perspective on systems research. His invaluable insights and advice during our collaboration have guided me throughout my studies.

I would like to thank my parents, Igal Golan and Hadara Ben Yosef, who have taught me to question, explore, and persist. They supported me in every aspect of my studies, from elementary school to graduate school. I thank my parents in law, Menashe and Yardena Yadgar, who went out of their way to help and encourage me. I am in debt to my children, Naama and Erez, for their unconditional love, joy and inspiration that accompanied my research.

Last but not least, I thank my husband, Avi Yadgar, for his endless love and support, and his priceless help with everything, from the small details to the big dilemmas. He is my rock.

The generous financial help of The Technion, Hasso Plattner Institut, and The Israeli Ministry of Science and Technology is gratefully acknowledged.
List of Publications


# Table of Contents

Table of Contents  
Abstract  
Symbols and Abbreviations  

1 Introduction  
1.1 The challenges of multilevel caching  
1.2 Our approach to multilevel caching  
1.3 The challenges of cooperative caching  
1.4 Our approach to cooperative caching  

2 Non Cooperative Storage model  

3 Marginal Gain  

4 Single Client: Karma  
4.1 Overview  
4.2 Hints  
4.3 Allocation  
4.4 Replacement  

5 Sharing Scenarios and Caching Goals  

6 Multiple Clients: $MC^2$  
6.1 Local allocation and replacement  
6.2 Global allocation  
6.3 Global replacement
7 Cooperative Storage Model ........................................... 43
  7.1 Motivation .................................................................. 43
  7.2 Requirements ................................................................ 45
  7.3 Model definitions ......................................................... 47
  7.4 Objective functions ....................................................... 48

8 Cooperative Caching Approaches ..................................... 52
  8.1 Cooperative distributed hash table .................................. 52
  8.2 Peer-to-peer cooperative caching .................................... 54
  8.3 Cooperative ARC ......................................................... 55
  8.4 Cooperative caching with hints ...................................... 56

9 Experimental Testbed ....................................................... 62
  9.1 Simulation environment ................................................ 62
  9.2 PostgreSQL database .................................................. 65
  9.3 Workloads ................................................................... 66
    9.3.1 TPC benchmark traces ............................................ 66
    9.3.2 Video playback ...................................................... 68
    9.3.3 Synthetic Zipf workload ......................................... 68
  9.4 Comparative framework ............................................... 69
  9.5 Application controlled file caching ............................... 71

10 Results ........................................................................ 73
  10.1 Karma ...................................................................... 74
  10.2 MC² .......................................................................... 86
  10.3 Cooperative approaches ............................................. 96
    10.3.1 TPCH query sets workload .................................... 96
    10.3.2 Video workload .................................................. 99
    10.3.3 TPCC workload .................................................. 100
    10.3.4 TPCH query combinations workload ...................... 102
    10.3.5 Energy considerations ........................................... 104
    10.3.6 Storage access costs ............................................ 106
    10.3.7 Upload bandwidth ............................................... 108
    10.3.8 Lookup mechanisms ............................................. 108
    10.3.9 Account limits .................................................... 109
    10.3.10 Hint generation .................................................. 111
    10.3.11 Configuration search .......................................... 111
    10.3.12 Is cooperative caching really helpful? .................... 113
11 Discussion
11.1 Write traffic .......................................................... 115
11.2 The quality of hints ................................................. 117
11.3 Multiple servers ..................................................... 119
11.4 Security ................................................................. 120

12 Related Work
12.1 Multiple cache levels ............................................... 121
12.2 Application based caching .......................................... 125
12.3 Prefetching as a form of data sharing ......................... 129
12.4 Offline replacement .................................................. 130
12.5 Storage system design ............................................. 132
12.6 Cooperative P2P and storage systems ....................... 133
12.7 Theoretical models for cooperation ............................. 136
12.8 Energy management in storage systems ..................... 137

13 Conclusions ................................................................. 139

Bibliography ................................................................. 155
List of Figures

2.1 Non-cooperative storage model ........................................ 13
4.1 Karma’s allocation of buffers to ranges in two cache levels .... 22
4.2 Pseudocode for allocation and helper functions in Karma .... 23
4.3 Pseudocode for I/O operations in Karma (Part I) ............... 24
4.4 Pseudocode for I/O operations in Karma (Part II) ............... 25
5.1 Constructive sharing example ......................................... 31
5.2 Symbiotic and destructive sharing examples ..................... 33
6.1 Allocation for multiple clients ....................................... 36
6.2 Pseudocode for management of shared caches in $MC^2$ ........ 37
7.1 Cooperative storage model and operations ......................... 45
8.1 Cooperative policies on the scale of selfishness ................. 53
8.2 C-Hints Pseudocode (Part I) ........................................ 59
8.3 C-Hints Pseudocode (Part II) ...................................... 60
8.4 C-Hints Pseudocode (Part III) .................................... 61
10.1 What effect does client selfishness have on performance? .... 74
10.2 Karma’s improvement over LRU on Query 3 run multiple times. 74
10.3 Average I/O response time for all policies on Query 3 ........ 75
10.4 Average I/O response time on repeated runs of all queries ... 77
10.5 Average I/O response time on 20 query sets .................... 81
10.6 Average I/O response time on a Zipf workload ................ 82
10.7 Karma’s performance with different disk access delays ... 85
10.8 Average I/O response time with cache levels of different sizes 87
10.9 TPCH queries 3 and 7 ............................................. 88
10.10 TPCH query sets on 5 clients .................................... 89
10.11 Ten clients running TPCC ........................................ 91
10.12 Cooperative approaches on the TPCH query sets ............ 97
10.13 The effect of client selfishness on performance .............. 98
10.14 Cooperative approaches with the video workload ............ 99
10.15 Cooperative approaches with the TPCC workload .......... 101
10.16 Cooperative approaches with TPCH queries 3, 10, and 18 ... 103
10.17 Reduction in I/O delay on TPCH query combinations ........ 104
10.18 Energy consumption and Energy Delay Product ............ 106
10.19 Cooperation with increasing storage access costs .......... 107
10.20 The effect of client selfishness on performance ............ 114

12.1 Relaxed list update problem example ............................ 131
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Summary of workload characteristics</td>
<td>66</td>
</tr>
<tr>
<td>9.2</td>
<td>Non cooperative policies in our comparative framework</td>
<td>70</td>
</tr>
<tr>
<td>10.1</td>
<td>Average I/O response time for each query with LRU</td>
<td>76</td>
</tr>
<tr>
<td>10.2</td>
<td>Reduction in average I/O response time due to sharing</td>
<td>93</td>
</tr>
<tr>
<td>10.3</td>
<td>Increase in average I/O response time due to sharing</td>
<td>95</td>
</tr>
<tr>
<td>10.4</td>
<td>Average and worst-case fairness</td>
<td>95</td>
</tr>
<tr>
<td>10.5</td>
<td>Power specifications</td>
<td>105</td>
</tr>
<tr>
<td>10.6</td>
<td>Changes in load balance and performance</td>
<td>110</td>
</tr>
<tr>
<td>10.7</td>
<td>Prediction accuracy for the video workload</td>
<td>112</td>
</tr>
<tr>
<td>12.1</td>
<td>Summary of non cooperative cache management policies</td>
<td>134</td>
</tr>
</tbody>
</table>
Abstract

Modern storage systems are designed as standalone platforms, separated from their users and applications by strict protocols. This modularity allows for complex and dynamic system layouts, combining hardware from a range of manufacturers. However, the separation between the storage and the application layers precludes inter-layer information sharing that is crucial for cooperation between the system’s components – cooperation we believe will lead to substantial performance gains.

We propose to share information about the cached disk blocks and use it to make informed caching decisions. This information is currently available only in the application level, but is highly useful for managing the cache in all levels of the hierarchy. We define a new storage model in three stages.

In our initial model, the multilevel cache hierarchy is formed of a single client accessing a single storage server. Multilevel caching introduces new challenges to traditional cache management: data must be kept in the appropriate cache and replication avoided across the various cache levels. In the second model, lower levels of the hierarchy are shared by multiple clients, introducing additional challenges. While data fetched by one client can be used by another client without incurring additional delays, clients competing for cache buffers can evict each other’s blocks and interfere with exclusive caching schemes.

We present a global non-centralized, dynamic and informed management policy for multiple levels of cache, accessed by multiple clients. Our algo-
rithm, \( MC^2 \), combines local, per client management with a global, system-wide scheme, to emphasize the positive effects of sharing and reduce the negative ones. Our local scheme, *Karma*, uses readily available information about the client’s future access profile to save the most valuable blocks, and to choose the best replacement policy for them. The global scheme uses the same information to divide the shared cache space between clients, and to manage this space. Exclusive caching is maintained for non-shared data and is disabled when sharing is identified.

Our final model enables cooperative caching, where clients may access blocks directly from each other’s caches. Previous studies treated all the cooperating caches as a single pool, maximizing overall system performance at the price of possibly degraded performance for individual clients. In light of the popularity of many P2P mechanisms, we re-evaluate the concept of cooperative caching, considering *selfish* clients that cooperate only if they benefit from the interaction.

We present and analyze several novel cooperative caching approaches that are inspired by known collaborative mechanisms for varying degrees of client selfishness. Our evaluation focuses on the performance as well as the energy requirements of these approaches, on a wide range of systems and two workload domains. Our results provide initial insights regarding the effect client selfishness has on the performance of a cooperative storage cache.
Symbols and Abbreviations

\[ cache_n \]  Cache at level \( n \)
\[ C_i \]  The access cost from level \( i - 1 \) to cache level \( i \)
\[ D_i \]  The cost of demoting a block from level \( i - 1 \) to level \( i \)
\[ Hit(m) \]  The expected hit rate for storing \( m \) blocks in the cache
\[ MG(m) \]  The marginal gain for an access trace and a cache of size \( m \)
\[ F_r \]  The percent of all accesses in the trace, which address range \( r \)
\[ NMG_r(m) \]  Normalized marginal gain \((MG(m) \times F_r)\)
\[ C_{disk} \]  The cost of a disk access
\[ C_{net} \]  The cost of performing a network transfer
\[ C_{serve} \]  The cost incurred by a client when it serves a peer request
\[ P_{busy} \]  The maximal energy consumption rate of the CPU
\[ P_{net} \]  The energy consumption rate of the CPU while performing a NIC or DMA transfer
\[ P_{idle} \]  The energy consumption rate of an idle CPU
\[ max_p \]  Maximal attempts a client is allowed to receive a block peers, before requesting from the server (in C-ARC and C-P2P)
\[ F_C(R) \]  The frequency of accesses of client \( C \) to blocks in range \( R \)
\[ ET \]  Energy-Delay product
Chapter 1

Introduction

Caching is used in storage systems to provide fast access to recently or frequently accessed data, with non-volatile devices used for data safety and long-term storage. Much research has focused on increasing the performance of caches as a means of improving system performance. In many storage system configurations, client and server caches form a two-layer (or more) hierarchy, introducing new challenges and opportunities over traditional single-level cache management. These include determining which level to cache data in and how to achieve exclusivity of data storage among the cache levels given the scant information available in all but the highest-level cache. Addressing these challenges can significantly improve overall system performance.

A cache replacement policy is used to decide which block is the best candidate for eviction when the cache is full. The hit rate is the fraction of page requests served from the cache, out of all requests issued by the application. Numerous studies have demonstrated the correlation between an increase in hit rate and application speedup [28, 31, 32, 48, 56, 63, 107, 114, 121]. This correlation motivates the ongoing search for better replacement policies. The most commonly used online replacement policy is LRU. Pure LRU has no notion of frequency, which makes the cache susceptible to pollution from looping or sequential access patterns [93, 105]. Various LRU variants,
e.g., LRU-K [83], 2Q [60], LRFU [68], LIRS [58] and ARC [75], attempt to account for frequency as well as temporal locality.

A different approach is to manage each access pattern with the replacement policy best suited for it. This is possible, for example, by automatic classification of access patterns [32, 48, 63], or by adaptively choosing from a pool of policies according to their observed performance [12, 51]. In informed caching, replacement decisions are based on hints disclosed by the application itself [28, 33, 87]. Although informed caching has drawbacks for arbitrary applications (see Chapter 12), these drawbacks can be addressed for certain classes of applications, such as database systems [34, 80, 94]. File systems can also derive access patterns from various file attributes, such as the file extension or the application accessing the file. The Extensible File System [61] provides an interface which enables users to classify files and the system to derive the files’ properties. In IBM’s GPFS [6], applications may explicitly disclose aspects of their access pattern to improve performance. Recent tools provide automatic classification of file access patterns by the file and storage systems [42]. Despite the proven advantage of informed caching, it has been employed only in the upper level cache.

In cooperative storage caching, clients may access blocks directly from each other’s caches. Previous studies treated all the cooperating caches as a single pool, maximizing overall system performance at the price of possibly degraded performance for individual clients. In light of the popularity of many P2P mechanisms, we re-evaluate the concept of cooperative caching, considering selfish clients that cooperate only if they benefit from the interaction.

1.1 The challenges of multilevel caching

The above approaches attempt to maximize the number of cache hits as a means of maximizing overall performance. However, in modern systems
where both the server and the storage controller often have significantly large caches, a multilevel cache hierarchy is formed. Simply maximizing cache hits on any individual cache in a multilevel cache system will not necessarily maximize overall system performance. Therefore, given a multilevel cache hierarchy, we wish to minimize the I/O response time experienced by all the clients in the system, which is the sum of delays incurred by all data transfers between the caches and the delay of accessing each.

Multilevel cache hierarchies introduce three major problems in cache replacement. The first is the hiding of locality of reference by the upper cache [121]. The second is data redundancy, where blocks are saved in multiple cache levels [31, 77]. The third is the lack of information about the blocks’ attributes (e.g., their file or the application that issued the I/O request) in the lower level caches [100].

Accesses to the low level cache are misses in the upper level. Thus, these accesses are characterized by weak temporal locality. Since LRU is based on locality of reference, its efficiency diminishes in the second level cache. Policies such as FBR [91], MultiQ [121], ARC [75] and CAR [18] attempt to solve this problem by taking into account frequency of access in addition to recency. MultiQ, for example, uses multiple LRU queues with increasing lifetimes. ARC [75] and its approximation CAR [18] distinguish between blocks that are accessed once and those that are accessed more than once. None of the above policies address the cache hierarchy as a whole, but rather manage the cache levels independently, assuming the upper level cache is managed by LRU.

In exclusive caching [69, 114, 121], a data block should be cached in at most one cache level at a time. One way to do this is by means of the demote operation [114]. The lower level deletes a block from its cache when it is read by the upper level. When the upper level evicts an unmodified block from its cache, the block is sent back to the lower level using demote. The lower level tries to find a place for the demoted block, evicting another
block if necessary.

When the lower levels of the hierarchy are accessed by several clients, data sharing may occur. Sharing introduces both positive and negative effects on the performance of a multilevel cache. On the positive side, blocks fetched by one client may later be requested by another client. The second client experiences the effect of prefetching, when blocks are present in the shared cache and do not have to be fetched synchronously. On the negative side, sharing introduces two major challenges. The first is maintaining exclusivity: a block may be cached in the first level cache by one client and in the second level by another. Furthermore, exclusivity may deprive clients of the benefit of sharing described above, since blocks fetched by one client are not available for use by others. Several previous studies acknowledge this problem [114, 122], while others assume that clients access disjoint data sets [69, 100, 112].

The second challenge is meeting the demands of competing clients for space in the shared cache, while minimizing their interference. A common approach for allocation in shared caches is partitioning, where each client is allocated a portion of the cache buffers according to a global allocation scheme [59, 112]. This approach is problematic when blocks are accessed by several clients and might belong to more than one partition.

1.2 Our approach to multilevel caching

This study analyzes four basic data sharing scenarios in a multilevel cache hierarchy. We show that the positive effects can be enhanced and the negative ones reduced, by choosing the right replacement policy. This analysis leads to our main contribution: $MC^2$, a novel approach to the management of multilevel cache systems which attempts to address all of the above issues in concert. $MC^2$ is built on top of Karma, an exclusive, hint based approach for managing a multilevel cache hierarchy used by a single client. $MC^2$ further addresses the challenges introduced by multiple clients. In the rest of this
study, we use $MC^2$ to refer to our comprehensive solution, and Karma to describe its per-client functionality.

Each client provides application hints in the form of a list of block *ranges* it will access. The expected access pattern and access frequency to each range is also disclosed. $MC^2$ allocates a cache partition for each range. The size of the partition depends on the frequency of access to the range, the range’s size, and the access pattern of its blocks. Partitions accessed with higher frequency are placed at a higher cache level. Each partition is managed by the replacement policy most suited for its range.

The shared cache space in the lower levels is dynamically divided among competing clients. Our local per-client scheme, Karma, manages the space allocated for each client according to the hints it provided, as described above. The same hints are used to direct the management of shared partitions, which store ranges that are accessed by more than one client. The combination of access patterns determines the replacement policy and whether caching in each partition is exclusive or inclusive.

$MC^2$ maintains partitioning and exclusiveness in the cache with three I/O operations. We distinguish between a *read*, which deletes the read block from a lower level cache, and a *read-save*, which instructs a lower level to save a block in its cache. We also use *demote* to maintain exclusiveness in partitions of ranges that are split between multiple cache levels.

$MC^2$ is applicable to any application that is able to provide general hints about its access patterns. Databases are a classic example of such applications, where the access pattern is decided in advance by the query optimizer. We base our experimental evaluation both on real database traces and on synthetic traces with Zipf distribution. For the database traces, we used the *explain* mechanism of the PostgreSQL database as a source of application hints. For the synthetic workload, we supplied $MC^2$ with the access frequency of the blocks in the data set. We simulated a hierarchy of two cache levels and one storage level for comparing $MC^2$ to LRU, ARC [75],
MultiQ [121], LRU-SP [28] Demote [114], and a lower bound on optimal I/O response time [44]. We also defined and implemented extensions to these policies to apply to multiple levels of cache.

Our results show that $MC^2$ reduces I/O response times below those of all other online policies, in almost all cache sizes, both for the single client setting and for multiple clients. $MC^2$ is the best policy to use in all sharing scenarios, with performance close to the optimal lower bound. Its use of application hints enables matching the optimal policy to each access pattern or combination of access patterns. Its dynamic repartitioning eliminates sensitivity to changing access patterns, and its choice of exclusive caching allows every increase in the aggregate cache size to be exploited.

1.3 The challenges of cooperative caching

Resource consolidation is a prevalent means for saving power, maintenance, administrative and acquisition costs. Traditionally, storage and compute resources were consolidated within organizations [17, 113]. Similarly, cooperative caching was considered between centrally owned caches, with the goal of minimizing global I/O response time [37, 40, 57, 65, 95, 108].

Recently, however, resources are being consolidated on a much larger scale, often involving resources owned by different entities. Common examples include computational grids [27], elastic clouds, and large scale data centers [17, 86]. The increasing scale entails an increasing number of users and applications, possibly with conflicting goals. As a result, the management of these systems becomes increasingly complex, not only because of the mass of metadata and administrative concerns, but also because of the need to balance users’ conflicting goals and to comply with agreed upon service level objectives (SLOs).

Cooperative caching is a test case for management in such systems. Caches are a valuable resource and are usually fully utilized by their owners. Data
sharing may occur even between separately owned clients, for example in access to popular images, executables or Web pages, when different departments analyze the same data, and when separate services use the same data repository [43]. We focus on environments with multiple servers and a shared remote storage. In these environments, it is substantially cheaper to access data from another client’s cache than it is to access the shared storage. Ideally, in such a setting, data will only be replicated in multiple caches when doing so is necessary to avoid contention or improve locality.

Cooperation in such scenarios can eliminate unnecessary redundancy and greatly improve performance. However, cooperative caching incurs additional overheads that may degrade the performance of some participating caches. Their owners, being autonomous, will in turn refuse to cooperate.

Existing algorithms address performance isolation among users of shared resources. However, cooperating caches are privately owned resources, and the selfish needs and objectives of their owners must be addressed explicitly in order for them to cooperate willingly. While selfishness is well-studied in the network domain [36, 62, 79, 88], the corresponding peer-to-peer mechanisms are not sufficient for enabling cooperation in stateful systems such as caches. At the same time, existing theoretical models for cooperation are computationally hard [110] and impractical for managing large scale dynamic systems.

The above limitations of existing models call for a new storage model. Such a model should satisfy two requirements: an explicit cost of cooperation, and a way to compute the true utility provided by a cache in a collaborative system. This will allow selfish clients to weigh both the work invested in cooperation and its effect on individual performance.
1.4 Our approach to cooperative caching

Our first goal was to define a new model for cooperative caching, that satisfies the above requirements. Our second goal was to evaluate, based on the new model, the effect that client selfishness has on performance. In order to do so, we propose several novel caching approaches, sampling the range of client behaviors, from selfish to altruistic, where clients always cooperate. The analysis of these approaches exposes the potential benefits and limitations of cooperative caching with selfish clients.

In this work, we first point out the limitations of existing approaches in managing cooperative caching between selfish entities. Second, we propose a new model that satisfies the above requirements. Last, we present experimental results that demonstrate the limitations of existing models, the benefits of our proposed model, and the effect that various aspects of client selfishness have on performance.

Our evaluation focuses on the performance as well as the energy requirements of our proposed cooperative caching approaches, on a wide range of systems and two workload domains. 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%. We identify the situations that are especially vulnerable to such choices, as well as those which preclude cooperation altogether. Our analysis provides important new guidelines for managing complex systems with numerous caches.
Chapter 2

Non Cooperative Storage model

Our model, shown in Figure 2.1(a), consists of several clients, each with its own cache. There are $n$ cache levels, organized in a tree rooted at $cache_n$, which is attached to the storage level, or disk array (in short, Disk). The access cost from level $i - 1$ to cache level $i$ is $C_i$. The cost of a disk access is $C_{Disk}$. The cost of demoting a block from level $i - 1$ to level $i$ is $D_i$. We assume that a lower cache level carries an increased access cost, and that demoting and access costs are equal for a given level. Namely,

$$C_1 = D_1 < C_2 = D_2 < \ldots < C_n = D_n < C_{Disk}.$$

Typically, the caches in the first level reside in the clients’ memory and $cache_n$ resides on the storage controller. Additional cache levels may reside in either of these locations, as well as in additional locations in the network. The access costs, $C_i$ and $D_i$, represent a combination of computation, network, and queuing delays. $C_{Disk}$ also includes seek times. While these delays may vary in a real system, we do not attempt to model them in detail, and use fixed, average access costs.

The model is demand paging, read-only (for the purpose of this work, we assume a separately managed write cache, discussed further in Chapter 11),
Figure 2.1: Our storage model (a) consists of $n$ cache levels, with possibly several caches in each level. READ and READ-SAVE fetch a block from level $i$ to level $i-1$ and DEMOTE sends a block from level $i-1$ to level $i$. We pay special attention to the case of a single client (b).

and defines three operations:

- **READ** $(x, i)$ – move block $x$ from $\text{parent(cache)}$ in level $i+1$ to $\text{cache}$ in level $i$, removing it from $\text{parent(cache)}$. If $x$ is not found in $\text{parent(cache)}$, READ$(x, i+1)$ is performed recursively, stopping at $\text{Disk}$ if $x$ is not found earlier.

- **READ-SAVE** $(x, i)$ – copy block $x$ from $\text{parent(cache)}$ in level $i+1$ to $\text{cache}$ in level $i$. If $x$ is not found in $\text{parent(cache)}$, READ-SAVE $(x, i+1)$ is performed recursively, stopping at $\text{Disk}$ if $x$ is not found earlier.

- **DEMOTE** $(x, i)$ – move block $x$ from $\text{cache}$ in level $i$ to $\text{parent(cache)}$ in level $i+1$, removing it from $\text{cache}$.

In other words, a READ operation in $\text{cache}$ instructs $\text{parent(cache)}$ to discard the block in order to maintain exclusivity. If READ-SAVE is used instead, $\text{parent(cache)}$ can continue storing the block without interfering with
exclusivity. The distinction between the **READ** and **READ-SAVE** operations can be realized using the existing SCSI command set, by setting or clearing the **disable page out** (DPO) bit in the **READ** command [3].

A **DEMOTE** occurs only on a cache miss, when a new block is brought into the cache. It is therefore modeled as a synchronous operation, blocking the **READ** request of the new block. The model can be expanded to enable **early DEMOTE**, which occurs in the background and therefore does not necessarily incur the extra delay. This expansion and the required decisions of when to **DEMOTE** are not currently part of our model and algorithm.

Our goal is to minimize the **average I/O response time** for each client. While the load on the disk, the distribution of requests, and the access pattern all affect the disk access costs, we focus on the performance of the cache hierarchy. Therefore, we assume a constant cost for all disk accesses. However, we note that when data sharing occurs, a cache miss does not always incur the same delay; when two clients fetch the same block, the second client issuing the request may experience a shorter response time than the first, as the request is already being processed.

In the non-cooperative model, each client may retrieve blocks only from the cache directly attached to it, or its ancestors in the hierarchy. Furthermore, we assume a client has no information about applications running on other clients. Thus, each client chooses the most valuable blocks to save in its own space, regardless of blocks saved by other clients.

Ideally, each client is oblivious to the presence of other clients and operates under the illusion that it has the lower level caches all to itself, although they may seem smaller than they are in practice. This simplified model, depicted in Figure 2.1(b), is discussed in Chapter 3. We discuss the implications of the general model in Chapter 5.
Chapter 3

Marginal Gain

The optimal offline replacement policy for a single cache is Belady’s MIN [23]. Whenever a block needs to be evicted from the cache, MIN evicts the one with the largest forward distance – the number of distinct blocks that will be accessed before this block is accessed again. To develop our online multilevel algorithm, we have opted to use application hints in a way which best approximates this forward distance. To this end, we use the notion of marginal gains, which was defined in previous work [80].

The marginal gain for an access trace is the increase in hit rate that will be seen by this trace if the cache size increases by a single block:

\[ MG(m) = Hit(m) - Hit(m - 1), \]

where \( Hit(m) \) is the expected hit rate for storing \( m \) blocks in the cache. Below we show how \( MG(m) \) is computed for three common access patterns: looping, sequential, and random. Although we focus on these three patterns, similar considerations can be used to compute the marginal gain of any other access pattern for which the hit rate can be estimated [33, 63, 83, 87].

Obviously, the marginal gain depends on the replacement policy of the cache. We assume that the best replacement policy is used for each access pattern: MRU (Most Recently Used) is known to be optimal for sequential and looping references, whereas LRU is usually employed for random
references [28, 32, 33, 34, 63, 94].

Sequential accesses. For any \( m \), since no block is previously referenced, the hit rate for a sequential access trace is \( \text{Hit}_{\text{seq}}(m) = 0 \). Thus, the resulting marginal gain is 0 as well.

Random (uniform) accesses. For an access trace of \( R \) blocks of uniform distribution, the probability of accessing each block is \( \frac{1}{R} \) [80]. For any \( m \leq R \), the hit rate is thus \( \text{Hit}_{\text{rand}}(m) = \frac{m}{R} \). The resulting marginal gain is:

\[
MG_{\text{rand}}(m) = \begin{cases} 
\frac{m}{R} - \frac{m-1}{R} = \frac{1}{R} & m \leq R \\
0 & m > R.
\end{cases}
\]

Looping accesses. The loop length of a looping reference is the number of blocks being re-referenced [63]. For a looping reference with loop length \( L \), the expected hit rate for \( m \) blocks managed by MRU is \( \text{Hit}_{\text{loop}}(m) = \frac{\min(L,m)}{L} \). Thus,

\[
MG_{\text{loop}}(m) = \begin{cases} 
\frac{m}{L} - \frac{m-1}{L} = \frac{1}{L} & m \leq L \\
\frac{L}{L} - \frac{L}{L} = 0 & m > L.
\end{cases}
\]

In other words, the marginal gain is constant up to the point where the entire loop fits in the cache and zero from there on.

We deal with traces where accesses to blocks of several ranges are interleaved, possibly with different access patterns. Each range of blocks is accessed with one pattern. In order to compare the marginal gain of references to different ranges, we use the frequency of access to each range. Let \( F_r \) be the percent of all accesses which address range \( r \). Define the normalized expected hit rate for range \( r \) as \( \text{Hit}_r(m) \times F_r \), and the normalized marginal gain for range \( r \) as \( NMG_r(m) = MG_r(m) \times F_r \).

Although marginal gains are defined for a single level cache and measure hit rate rather than I/O response time, normalized marginal gains induce an order of priority on all ranges – and thus on all blocks – in a trace. This
order is used by our online management algorithm to arrange the blocks in a multilevel cache system: the higher the range priority, the higher its blocks are placed in the cache hierarchy. This strategy maximizes the total normalized marginal gain of all blocks stored in the cache.

Note that when all blocks in a range have the same access frequency, there is a correlation between the normalized marginal gain of a range and the probability that a block of this range will be accessed. A higher marginal gain indicates a higher probability. For any two ranges, \( a \) and \( b \), if \( NMG_a(m) > NMG_b(m) \), then all the blocks of range \( a \) will be accessed with higher probability than all the blocks of range \( b \). Cache space should thus be allocated for all the blocks of range \( a \). Only if there is space left (possibly in lower cache levels), should it be allocated to range \( b \). If \( NMG_a(m) = NMG_b(m) \), the order of allocation is arbitrary. Note, however, that partial allocation for both ranges yields no benefit.

Ranges accessed with a non-uniform distribution are divided into smaller ranges according to their distribution, as described below.

**Non-uniform accesses.** A skew of a range is defined by the pair \((x, y)\), where \( x\% \) of the accesses to this range request \( y\% \) of its blocks. The skew implies that some blocks in the range have a higher access frequency than others, and are thus more valuable, but it does not reveal which blocks. As opposed to the above access patterns, the marginal gain of skewed ranges is not equal for all blocks. We do not attempt to compute the marginal gain of each block. Rather, we characterize the entire range according to its skew, as follows.

A skew of \((x, y)\) splits range \( r \) into two subranges in the following manner. The size of the first subrange is \( y\% \) of the size of \( r \), and the frequency of access to it is assumed to be \( x\% \) of the frequency of \( r \). The second subrange is its complement, both in size and in frequency. The marginal gain is computed for each subrange as if it were accessed uniformly. When the access skew of a data set is available to the application or to an administrator (as...
in some well-tuned database systems), more accurate hints can be used. A skew with \( s-1 \) pairs of parameters, \((x_1, y_1), \ldots, (x_{s-1}, y_{s-1})\), divides a range into \( s \) subranges, enabling refined allocation decisions. When no information is available, we follow Pareto’s principle and divide the range according to a skew of \((80,20)\).

Partitioning the cache according to marginal gains is optimal if the “gain” from storing a range of blocks in the cache depends solely on the normalized hit rate in its partition. This holds as long as the cost of fetching a block from the lowest level (Disk) is equal for all cache misses. This is true in our storage model, where all disk accesses have equal cost, but does not necessarily hold in more complicated ones. For example, clients can access several data sources instead of just one Disk. If these sources incur different access costs, such as the different costs of SSD, SAS and SATA, the computation of marginal gains should be adjusted to reflect those costs. This could be done, for example, by normalizing the marginal gain according to those costs. If the access costs change over time (due to load balancing, etc.), this should be indicated by external hints or monitoring.

Access costs may vary even within the same data source. For example, seeking and queuing delays may vary within the same storage array, but are very hard to predict; incorporating them into the computation of marginal gains is a nontrivial task.

Finally, if caches in the same level cooperate and fetch blocks from one another, then the gain from storing a block depends on whether it should be fetched from its source, or from a cache in the same level. Thus, the marginal gain of a block for one client will depend on the cache content of other clients. Therefore, in our cooperative caching model, described in Chapter 7, we use access frequencies rather than marginal gains.

A major advantage of basing caching decisions on marginal gains is the low level of detail required for their computation. Since only the general
access pattern and access frequency are required, it is much more likely that an application will be able to supply such information. Our experience shows that databases can supply this information with a high degree of accuracy. We expect that it will also be possible to derive our hints from information available to the file system [42, 61]. Our approach is not limited to the access patterns described above. \( MC^2 \) supports any access pattern for which the marginal gain can be computed without online monitoring.
Chapter 4

Single Client: Karma

4.1 Overview

Karma (Know-it-All Replacement for a Multilevel Cache) is our algorithm for managing the blocks of one client in a multilevel cache hierarchy. The pseudocode for Karma appears in Figures 4.2, 4.3 and 4.4. The line numbers in this section refer to these figures. Our algorithm for multiple clients is described in Chapter 6.

Karma calculates the normalized marginal gain of a range of blocks (which corresponds to each of the sets described in Chapter 1.2) and then uses it to indicate the likelihood that the range’s blocks will be accessed in the near future. To calculate the normalized marginal gain, as described in Chapter 3, Karma requires that all accessed disk blocks be classified into ranges. The following information must be provided (by means of application hints) for each range: an identifier for the range, its access pattern, the number of blocks in the range, and the frequency of access to this range. Each block access is tagged with the block’s range identifier, enabling all cache levels to handle the block according to its range.

Karma allocates for each range a fixed cache partition in a way that maximizes the normalized marginal gain of the blocks in all cache levels. It places
ranges with higher normalized marginal gain in higher cache levels (closer to the application), where the access cost is lower. In other words, space is allocated in $Cache_i$ for the ranges with the highest normalized marginal gain that were not allocated space in any $Cache_j$, $j < i$ (lines 28-33). Hints describing the remaining ranges are sent to the lower level. For each level $i$ there can be at most one range that is split and allocated space in both level $i$ and the adjacent lower level $i + 1$. Figure 4.1 shows an example of Karma’s allocation.

Each range is managed separately, with the replacement policy best suited for its access pattern. When a block is brought into the cache, a block from the same range is discarded, according to the range’s policy (line 46). This maintains the fixed allocation assigned to each range.

The space overhead of Karma depends on the number of ranges accessed by the client and the number of partitions in the cache. The corresponding data structures are very small and the total space required is in the order of a few hundred bytes, typically less than one cache block.

### 4.2 Hints

The ability to propagate application information to the lower cache levels is essential to Karma. Specifically, the range identifier attached to each block access is crucial for associating the block with the knowledge about its range. These identifiers are very small and do not incur significant overhead. A method for passing information (sufficient for Karma) from the file system to the I/O system was suggested [19] and implemented in a Linux 2.4.2 kernel prototype.

For the two tables joined in the example in Figure 4.1, Karma will be supplied with the division of the blocks into tables and index tree levels, as in Figure 4.1(c). Additionally, each cache level must know the aggregate size of all cache levels above it. Such information can be passed out-of-band,
Figure 4.1: Karma’s allocation of buffers to ranges in two cache levels. Each rectangle represents a disk data block. The application is running a database query. (a) Two database tables are joined by scanning one of them sequentially and accessing the second one via an index. (b) The resulting access pattern is an interleaving of four ranges: one sequential (S), two loops (L), and one random (R). (c) This partitioning into ranges is supplied to Karma at the beginning of query execution. (d) Karma allocates one buffer for the sequential accesses (see text), three to hold the root and inner nodes of the index, and the remaining space to the randomly accessed blocks.
Definitions:

- \( \text{partition}(r) \) - The cache partition of blocks belonging to range \( r \)
- \( \text{reserved} \) - A reserved buffer of size 1 in the cache level \( == 1 \)
- \( \text{readBlocks} \) - A partition holding blocks that were READ by an upper level cache and are candidates for eviction

Input:

- \( R \) ranges
- \( |\text{Cache}| = n \)
- \( \text{List}_{\text{seq}} \) A list of the ranges accessed in a sequential pattern (when level \( \neq 1 \) \( \text{List}_{\text{seq}} \) is empty).
- \( \text{List}_{\text{nonseq}} \) The remaining ranges sorted by NMG, descending (ties are broken arbitrarily).

(Re)Partition:

- \( \text{rangeAlloc} \[ R \] = \{0\} //allocation for each range, initialized to zeros
- \( \text{priorities} \[ R \] = \{0\} //priority of each range (bigger is better), initialized to zeros
- if (level \( == 1 \))
  - allocated = 1 //size of \( \text{reserved} \)
- else
  - allocated = 0
  - priority = 0
- foreach \( r \in \text{List}_{\text{seq}} \)
  - \( \text{rangeAlloc} \[ r \] = 1 \)
  - allocated++
  - priorities[\( r \)] = priority
  - priority++
  - priority = \( R \)
- foreach \( r \in \text{List}_{\text{nonseq}} \)
  - size = \( \min(n - \text{allocated}, |\text{Range}_r|) \)
  - \( \text{rangeAlloc} \[ r \] = \text{size} \)
  - allocated += size
  - priorities[\( r \)] = priority
  - priority--

LowestPriority(Range \( r \))

- Return range with lowest priority exceeding its allocated size.
- If none exists return \( r \).

Evict(Blocks \( X \), Range \( r \))

- if \( (X \text{ was in } \text{reserved}) \) or \( (r \in \text{List}_{\text{seq}}) \)
  - discard \( X \)
- else
  - DEMOTE \( X \)
  - discard \( X \)

Insert (Block \( X \), Range \( r \))

- if (cache is full)
  - \( LP = \text{LowestPriority}(r) \)
  - remove block \( Z \) from \( LP \) according to policy of \( LP \)
  - Evict (\( Z \))
  - put \( X \) in \( \text{partition}(r) \)

Figure 4.2: Pseudocode for allocation and helper functions in Karma
49. READ(Block X, Range r)
50.   Cache hit:
51.      if (level == 1)
52.         update place in LRU stack of partition(r)
53.      else
54.         remove X from partition(r)
55.         put X in readBlocks
56. Cache miss:
57.      if (level == 1)
58.         if (allocRanges[r] > 0) or ((cache not full) and (r \notin List_seq))
59.            READ X
60.            Insert(X, r)
61.       else // allocRanges[r] == 0
62.           remove block Y from reserved // Y belongs to range r_y
63.           if(priorities[LowestPriority(r_y)] < priorities[r_y])
64.              Insert(Y, r_y)
65.           else
66.              discard Y
67.           READ-SAVE X
68.           put X in reserved
69.       else // (level \neq 1)
70.           READ X
71.           return X to upper level
72.       if (cache is full)
73.           discard X
74.       else
75.           put X in readBlocks

Figure 4.3: Pseudocode for I/O operations in Karma (Part I)
76. **READ-SAVE**(Block X, Range r)
77. Cache hit:
78. update place in LRU stack of \(\text{partition}(r)\)
79. Cache miss:
80. if \((\text{allocRanges}[r] > 0)\) or (cache not full)
81. READ X
82. return X to upper level
83. Insert(X,r))
84. else // \(\text{allocRanges}[r] == 0\)
85. if \(\text{priorities}[\text{LowestPriority}(r)] < \text{priorities}[r]\))
86. READ X
87. return X to upper level
88. Insert(X,r)
89. else // no space for X
90. READ-SAVE X
91. return X to upper level
92. discard X

93. **DEMOTE**(Block X, Range r)
94. Cache hit:
95. remove X from \(\text{readBlocks}\)
96. put X in \(\text{partition}(r)\)
97. Cache miss:
98. if \((\text{allocRanges}[r] > 0)\) or (cache not full) or
99. \(\text{priorities}[\text{LowestPriority}(r)] < \text{priorities}[r]\))
100. Insert(X,r))
101. else // no space for X
102. DEMOTE X

Figure 4.4: Pseudocode for I/O operations in Karma (Part II)
without changing current I/O interfaces. Each block access will be tagged with its range identifier, enabling all cache levels to classify it into the correct partition.

Applications should define ranges for which they can easily describe the access pattern and predict the access frequency. Files are natural candidates for ranges. However, if many small files are accessed with the same characteristics, they can be hinted as a single range. Similarly, if parts of a large file are accessed with different patterns or frequencies, the file should be divided into several corresponding ranges. Determining the access pattern of applications that do not provide hints has been handled by other approaches (described in Chapter 12.2) and is outside the scope of this work.

As in all informed management policies, Karma’s performance depends on the quality of the hints. However, thanks to its exclusive caching, even with imperfect hints Karma will likely outperform basic LRU at each level. We discuss the effects of imperfect hints in Chapter 11. Our experiments with skewed ranges, described in Chapter 10, show that Karma can significantly reduce I/O response times even when only approximate hints are supplied.

4.3 Allocation

Allocating cache space to blocks according to their normalized marginal gain would result in zero allocation for sequential accesses. Yet, in such patterns the application often accesses one block repeatedly before moving on to the next block. In some database queries, for example, a block may be accessed a few times, until all tuples in it have been processed. Therefore, ranges accessed sequentially are each allocated a single block in the upper level cache (lines 22-26). If prefetching is added to the model, several blocks may be allocated to allow for sequential read-ahead.

We allow for locality within the currently accessed block by always bringing it into $Cache_1$. When this block belongs to a range with no allocated
space in $Cache_1$, we avoid unnecessary overhead by reading it using READ-SAVE (lines 61-68) and discarding it without using DEMOTE (lines 38-39).

**Lazy repartitioning.** When Karma is supplied with a new set of hints which indicate that the access patterns are about to change (e.g., when a new query is about to be processed), it repartitions all cache levels (lines 14-33). Cache blocks in each level are assigned to the new partitions. As a result, disk blocks which do not belong to any of the new partitions in the cache level where they are stored become candidates for immediate eviction. Note that the partitioning is logical, not physical: the cache blocks are not actually moved in the cache during repartition.

A block is rarely assigned a new range. For example, in a database blocks are divided according to their table or index level. Therefore, when a new query is processed, the division into ranges will remain the same; only the access frequency and possibly the access pattern of the ranges will change. As a result, the blocks will remain in the same partition, and only the space allocated for this partition will change. This means that the sizes of the partitions will have to be adjusted, as described below.

Blocks are never discarded when there is space in the cache or when it contains blocks that are immediate candidates for eviction (blocks from old ranges or blocks that were READ by an upper level). Karma ensures that even in transitional phases (between the time a new set of hints is supplied and the time when the cache content matches the new partitioning), the cache will keep the blocks with the highest marginal gain. As long as a cache (at any level) is not full, non-sequential ranges are allowed to exceed the size allocated for them (lines 58-60, 80-83). When no space is left and blocks from ranges with higher marginal gain are accessed, blocks from ranges which exceeded their allocated space are first candidates for eviction, in reverse order of their marginal gain (lines 63-66, 85-92).

The importance of handling this transitional stage is that it makes Karma less vulnerable to the order in which, for example, several queries are pro-
cessed in a row.

4.4 Replacement

Karma achieves exclusive caching by partitioning the cache. This partitioning is maintained by use of DEMOTE and READ-SAVE, where each cache level stores only blocks belonging to its assigned ranges. For each \( i, 1 \leq i \leq n - 1 \), \( \text{Cache}_i \) demotes all evicted blocks which do not belong to sequential ranges (lines 40-42). When \( \text{Cache}_i \) is about to read a block without storing it for future use, it uses READ-SAVE in order to prevent \( \text{Cache}_{i+1} \) from discarding the block (lines 61-67, 90-92). Only one such block is duplicated between every two cache levels at any moment (see Figure 4.2). For each \( j, 2 \leq j \leq n \), \( \text{Cache}_j \) does not store any READ blocks, but only those demoted by \( \text{Cache}_{j-1} \) or read using READ-SAVE.

Special attention must be given to replacement in ranges which are split between adjacent cache levels \( i \) and \( i+1 \). The LRU (or MRU) stack must be preserved across cache levels. \( \text{Cache}_i \) manages the blocks in a split range with the corresponding policy, demoting all discarded blocks. \( \text{Cache}_{i+1} \) inserts demoted blocks at the most recently used (MRU) position in the stack and removes them from the MRU or LRU position, according to the range’s policy. Blocks READ by \( \text{Cache}_i \) are removed from the stack and become immediate candidates for eviction (lines 54-55, 74-75). This way, the stack in \( \text{Cache}_{i+1} \) acts as an extension of the stack in \( \text{Cache}_i \).

A similar method is used to manage the partitions of subranges of a skewed range. In order to determine how much space should be allocated to skewed ranges, previous studies computed the marginal gain of different partition sizes at run time, using ghost caches [33, 63]. Karma avoids such space and computational overheads by allocating space only to the subranges with the highest marginal gain, as computed in Chapter 3. The corresponding partitions are managed in a continuous LRU stack: blocks always enter the
partition of the first subrange, and are always evicted from the last one in that cache. When the partition of one subrange is full, blocks are “demoted” to the partition of the next subrange, which may be in the same cache level or in a lower one. This allows LRU to keep in the cache the most frequently accessed blocks from this range.
Chapter 5

Sharing Scenarios and Caching

Goals

When several clients share the same cache, the effects of sharing can be positive, negative, or neutral. In many cases, a positive effect (scenario) can be attained and a negative one avoided by correct cache management decisions. We describe and analyze the different types of sharing and show, for some of them, how they can result from basic caching schemes.

No data sharing. In the most trivial case, the clients access disjoint data sets and compete for cache space. The main concern in this case is unfair allocation of cache buffers. For example, one client may page in blocks that repeatedly page out blocks used by another client. Several studies [28, 33, 63, 87, 94, 112] show that when the cache is partitioned among clients, either statically or dynamically, such interference can be avoided.

Neutral sharing. In neutral sharing, performance for all clients is unchanged regardless of the number of clients accessing the shared cache. For example, consider two clients iterating over the same loop, with similar think times between I/O requests. The clients request the same blocks at the same time. As a result, cached blocks incur the same benefit for both clients.

Constructive sharing. In constructive sharing, one or more clients
Figure 5.1: A range of blocks is accessed as a loop by two clients, both managing their cache with LRU. Each block is accessed 10 times before the loop iterator is advanced to the next block. The average think time (time between I/O requests) of Client_B is 10 times that of Client_A. The baseline performance is identical for both clients, since they request the same blocks in the same order. When the clients access the loop concurrently, Client_A is always the first to request the next block in the loop, experiencing the maximal delay for fetching blocks from the disk.
benefit from sharing while the performance of the rest is unimpaired. An example is shown in Figure 5.1, where two clients iterate over the same loop, advancing at different speeds. The “fast” client, with shorter think times, is always the first to request the next block in the loop. Thus, it always experiences the maximal delay for fetching blocks from the disk. In this example, the cache is large enough for the other client to always find the blocks in the cache. Thus, it enjoys an effect similar to prefetching. Its performance is better than the optimal offline lower bound for a single client (described in detail in Chapter 12.4), which assumes a demand paging model.

**Symbiotic sharing.** In *symbiotic sharing*, all clients experience improvement over their baseline performance, although some may benefit more than others. For example, in Figure 5.2, a range of blocks is accessed randomly by two clients, each with its own cache in the first level. The clients share a second level cache. Both clients access the blocks with the same non-uniform distribution and the same average think time. With LRU, sharing improves exclusivity in the cache hierarchy, since the clients do not access the exact same blocks at the same time. More unique blocks are saved in the aggregate cache, resulting in a symbiotic sharing scenario.

**Destructive sharing.** In *destructive sharing*, all the clients suffer increased I/O response times due to sharing. This is the case when the clients in Figure 5.2 use Demote. In the baseline case, exclusivity enables better utilization of the aggregate cache. However, exclusivity suffers when the clients run together, resulting in destructive sharing. Note that the I/O response times of both clients are shorter with Demote than with LRU, despite the destructive sharing in Demote. This demonstrates that symbiotic sharing does not necessarily correspond with better performance.

**Caching goals.** An ideal cache management policy should avoid destructive sharing scenarios whenever they degrade performance, and leverage constructive and symbiotic scenarios to guarantee the shortest I/O response times in as many situations as possible. In Chapter 10, we show that existing policies
are able to make such guarantees either for a single client or for multiple ones, but not for both cases. Although inclusive policies, such as LRU, ARC and MultiQ, lead to constructive sharing scenarios, they are unable to fully utilize the cache when it is used by a single client. Exclusive policies such as Demote, however, usually perform poorly for multiple clients, causing destructive sharing scenarios. In Chapter 10 we show how our algorithm, \( MC^2 \), outperforms existing policies in all scenarios.

While reducing I/O response times is our main goal, we wish to maintain a “fair” distribution of space between clients whenever possible. “Fairness” refers to the performance improvement experienced by the clients and not necessarily to the amount of space allocated to them. We rely on a definition of fairness for processor multithreading [41] which we adapt to cache performance. We define the speedup of a client as the ratio between the completion time of its application when running alone and its completion time when running in a shared system. Fairness is defined as the minimum ratio between speedups in the system: \( \text{Fairness} \equiv \text{speedup}_j / \text{speedup}_k \), where Client\(_j\) is the client with minimal speedup in the system, and Client\(_k\) is the one with
maximal speedup. It follows from this definition that $0 < Fairness \leq 1$. Perfect fairness is achieved when $Fairness = 1$, which means that all clients experience the same speedup (or slowdown). We use speedup instead of I/O response time in the definition of fairness in order to reflect the proportion of I/O compared to computation in each client.

When clients run different workloads, some may be more important than others. When a storage system defines workload priorities, an alternative goal may be to maintain the order of priorities in the distribution of cache space among clients. Priorities are not part of our model, but we describe in Chapter 6.2 how our allocation scheme can be extended to handle them.
Chapter 6

Multiple Clients: \(MC^2\)

\(MC^2\) (Multiple Clients on a Multilevel Cache) is our algorithm for the management of a multilevel cache hierarchy with multiple clients. Like Karma, \(MC^2\) consolidates the management of blocks that share the same access characteristics, managing each range of blocks in its own cache partition. The size of each partition is dynamically adjusted, as explained below. For each client in the system, the application running on it supplies the same hints used by Karma, namely, the division of blocks into ranges, along with each range’s size, access pattern, and the frequency of access to it. Each I/O request (in all cache levels) is tagged by the client running the application with the range identifier of the requested block.

Whenever a new block is brought into the cache and an old one must be evicted, the above information is used to answer three basic questions: Which client must give up a block? Which of its partitions must grow smaller? Which is the least valuable block in this partition?

In other words, for the allocation aspect of the problem, \(MC^2\) determines the quota of cache space for each range, and for the replacement aspect it manages the blocks within each range. Each aspect is handled on two levels: globally (system-wide) and locally (per client). The global allocation scheme determines the quota of each client in the shared cache space, according to
its frequency and efficiency of cache use. The local scheme determines the amount of space each range is allocated from its client’s quota, according to its marginal gain. The local replacement scheme chooses the best replacement policy for each client’s range, according to its hinted access pattern. The global replacement scheme chooses the best policy for shared ranges according to the combination of their access patterns.

6.1 Local allocation and replacement

The space allocated for each client consists of the client cache, which is the first level cache used only by this client, and the space allocated exclusively for this client in shared caches in lower levels. Note that this space can be viewed as a multilevel cache hierarchy with a single client, as depicted in Figure 2.1(b). Therefore, this space is managed by each client independently using Karma, as described in Chapter 4.
### Definitions:

103. **Definition definitions:**

104. \( Max_c \) - number of blocks allocated for \( Client_c \)
105. \( Alloc_c \) - number of blocks used by \( Client_c \)
106. \( Alloc_{cr} \) - number of blocks from \( Range_r \) allocated by \( Client_c \)
107. \( Global(P,C) \) - the global LRU-(partition,client) pair

### Input:

108. **Input:**

109. \( R \) ranges
110. \( C \) clients accessing Cache 2
111. \( |Cache_2| = m \)
112. \( List_c \) A list of ranges accessed by \( Client_c \), sorted by MG, descending
113. All clients use the same IDs for the ranges

### Initialization:

114. **Initialization:**

115. \( Ranges[R] = \{0\} \) //total allocation for each range
116. \( ClientsRanges[C][R] = \{0\} \) //allocation by each client
117. **foreach** \( c \in C \)
118. \( Alloc_c = 0, Max_c = m/C \)
119. **foreach** \( r \in List_c \)
120. **if** \( Alloc_c \geq Max_c \)
121. "break"
122. \( size = \min(Max_c - Alloc_c, |Range_r| - Ranges[r]) \)
123. \( Alloc_c += size \)
124. \( ClientsRanges[c][r] = size, Ranges[c] += size \)

### Client \( c \) accesses block \( X \) of \( Range_r \):

125. **Client \( c \) accesses block \( X \) of \( Range_r \):**
126. Cache hit:
127. update stacks
128. Cache miss:
129. let \( Client_k \) be the client in \( Global(P,C) \)
130. let \( Range_s \) be the range \( Client_k \) chooses as victim
131. **if** \( ShouldEvict(Client_c, Range_r, Range_s) \)
132. \( Max_c += 1, ClientsRanges[c][r] +=1, Ranges[r] +=1 \)
133. \( Max_k -=1, ClientsRanges[k][s] -=1, Ranges[s] -=1 \)
134. evict block from \( Range_s \) //according to replacement policy
135. save block \( X \) in \( Range_r \)
136. update stacks
137. **else** // \( Client_c \) does not wish to save \( X \)
138. update stacks

### Boolean \( ShouldEvict \) (Client \( c \), Range current, Range victim)

139. **Boolean \( ShouldEvict \) (Client \( c \), Range current, Range victim):**
140. **if** \( current \) is accessed by \( Client_c \)
141. look up priority list for \( Client_c \)
142. **if** priority of \( current \) > priority of \( victim \)
143. return false
144. **return** true

---

Figure 6.2: Pseudocode for management of shared caches in \( MC^2 \)
6.2 Global allocation

A high level description of the global allocation scheme is outlined in Figure 6.2. The line numbers in this section refer to this figure. We partition the cache between clients according to a global allocation scheme similar to that used in LRU-SP [28] for multiple processes. Initially, each client is allocated an equal share of the cache, which is then dynamically adjusted to reflect this client’s use of the shared cache. Whenever a new block is fetched into the cache, a victim client is chosen – a client whose block will be evicted to make room for the new block. In global LRU allocation, this client is the one holding the LRU block in the cache. This approach penalizes a client that does not use LRU replacement in its share of the cache. Such a client will constantly be chosen as victim, as shown in the following example.

Consider two clients sharing a second level cache, as depicted in Figure 6.1(a). Client<sub>A</sub> keeps blocks 1, 2, ..., 100 in the cache and accesses them in a loop. Client<sub>B</sub> keeps only a few blocks of a large range and accesses them randomly. For every 5 requests of Client<sub>A</sub>, Client<sub>B</sub> requests one block. The optimal policy will keep all the loop blocks in the cache, because they will be accessed again soon. However, when Client<sub>B</sub> misses on block c, the LRU block is 16, from the partition owned by Client<sub>A</sub>. According to global LRU allocation, Client<sub>A</sub> is the victim client and should have one of its blocks removed, even though all of them will be accessed before the blocks saved by Client<sub>B</sub>.

To avoid this problem, we maintain an LRU stack of partitions. The victim client is the owner of the least recently used partition. The share of the victim client is decreased, and a block from its lowest priority partition is evicted. The allocation for the client whose block entered the cache is increased (lines 130-135). In the example in Figure 6.1, by choosing the victim client according to the LRU partition, we ensure that the portion of the cache allocated to the random blocks will not increase at the expense of the loop partition. Note that when its allocation increases, a client may
temporarily save a block with a low marginal gain. The space occupied by this block will later be used to increase the size of a partition with a higher marginal gain (lines 142-144).

The result of the global allocation scheme is that $MC^2$ favors clients that use the shared cache both frequently and efficiently. Each access might increase a client’s share of the cache. Thus, frequency is rewarded without being hinted or monitored. Furthermore, we need not worry about clients changing their access frequency, or joining or leaving the system. If a client no longer accesses the cache, its partitions become the LRU ones, and their space is gradually allocated to more active clients. When a new client joins the system, it gradually accumulates its fair share of the cache according to its access frequency. Partitions which are accessed frequently stay in the MRU side of the partition stack. Thus, clients that save valuable blocks are rarely chosen as victims and are rewarded for efficient use of their share of the cache.

Although workload priorities are not part of our model, they can be handled in $MC^2$ by adding restrictions in the global allocation scheme. For example, the number of times a client is allowed to “claim” blocks and increase its allocation can also depend on the priority of its workload.

We assume that clients do not “cheat” the lower levels. Thus, they do not issue I/O requests for blocks that are not demanded by the application. Furthermore, they do not move their most valuable blocks to lower levels in order to increase their use of the shared cache. This may not always result in improved performance: at any time an even faster client may join the system and cause these valuable blocks to be evicted.

**Allocation of shared partitions.** When a range is accessed by more than one client, $MC^2$ manages its blocks in a single shared partition, rather than distributing its allocation between the clients’ shares. All clients accessing a shared partition contribute to its allocation. The maximal space is allocated,
from the share of each client, given the marginal gain it computed for the shared range and the space available to this client. The amount of space allocated for the shared partition is limited by the sum of client allocations, the range size, and the cache size (lines 122-124).

Each client accesses the shared partition with a different frequency. Thus, a client with a low access frequency might exceed its fair allocation when its blocks are accessed frequently by another client. If that happens, the shared partition will not reach the LRU position in the partition stack, and the client will not be chosen as victim. This may prevent other clients from saving more valuable blocks in the cache. To address this problem, we refine the LRU allocation scheme described above, and use a stack of \((partition, client)\) pairs. When client \(c\) accesses partition \(p\), the pair \((p, c)\) moves to the MRU position in the stack. When a block has to be evicted from the cache, the victim client is the client in the LRU \((partition, client)\) pair (line 129). Thus, the allocation for a shared partition remains proportional to the accesses to its range by all clients.

To identify shared ranges, all clients must use the same range identifiers when passing their hints to the shared cache. This can be achieved by employing a common naming convention. For example, the range identifier can be a hash value of the range’s first and last LBAs, if its blocks are contiguous. A hash of the blocks’ LBAs can be used if the blocks are spread unevenly across a file. If hash values are used and sharing is detected, it should be verified with the clients to rule out hash collision.

**Local repartitioning.** When \(Client_A\) is supplied with a new set of hints which indicate that its access pattern is about to change, it may repartition its cache or resize some of its partitions, as described in Chapter 4.3. This does not affect the space allocated for other clients, unless they share partitions with \(Client_A\).

Consider partition \(P\), which is shared by \(Client_A\) and \(Client_B\). As a result of the new hints, the space allocated for \(P\) from \(Client_A\)’s share might
decrease. The allocation in the share of Client$_B$ is unchanged. However, if $P$ holds a range with a high marginal gain for Client$_B$, its lower priority partitions may be resized in order to reallocate the buffers removed by Client$_A$.

6.3 Global replacement

The space allocated for each client in a shared cache is managed according to the hints this client receives from the application and forwards to lower levels. Each I/O request from that client is tagged (by the upper level) with the client’s identifier, along with the original application hint. This identification links the block to the partition allocated for its range. The same hints are used by the shared cache to identify sharing scenarios and choose the best replacement policy for each shared partition.

The global replacement scheme first chooses between exclusive and inclusive management. Exclusive caching greatly improves the performance of multiple levels of cache used by a single client. However, in the presence of multiple clients, exclusive policies may result in destructive sharing scenarios, and may even degrade performance below that of an inclusive policy. MC$^2$ distinguishes between two types of partitions. In non-shared partitions, used by a single client, exclusivity is maintained by saving demoted and READ-SAVE-d blocks, and discarding READ blocks. In shared partitions, exclusivity is turned off and READ requests are treated as if they were READ-SAVE. A shared partition stores blocks that were READ, READ-SAVE-d or demoted. The replacement policy in a shared partition is chosen according to the combination of access patterns, using the following guidelines.

Random. When all the clients access the partition in a random pattern, either with uniform or skewed distribution, its blocks are managed by LRU, which is known to perform well for random access patterns. When some blocks have higher access frequency than others, a block requested by one
client is likely to be requested soon by another client. Similarly, a demoted block might be requested again by the demoting client. Therefore, both read and demoted blocks are treated equally, allowing LRU to keep in the partition those blocks that are indeed reused.

**Loop.** Several clients iterating over the same loop may advance at different speeds. A block read [demoted] by one client is likely to be accessed by another client before it will be accessed again by the client that just read [demoted] it. In order to allow for this benefit of sharing, shared loop partitions are managed by LRU rather than MRU. LRU management keeps all the blocks in the cache for some time, which depends on the partition size, rather than discarding them immediately as in MRU. This allows other clients to access them at a low cost.

**Random and scan.** A looping or a sequential access to a block serves as a counterindication that the block will be accessed again soon. Therefore, when a range is accessed randomly by some clients and scanned by others (either sequentially or in a loop), the scans are not allowed to alter the LRU stack of the partition. The scanning clients benefit from the presence of frequently accessed blocks in the shared cache. The random clients are not harmed because polluting scans do not affect the content of the shared partition.
Chapter 7

Cooperative Storage Model

7.1 Motivation

Back in 2000, Bolosky et al. [26] suggested a distributed file system with no central administration, where “the owner of each machine determines who can store files on it, and owners establish contracts with other machine owners to collaboratively share their resources.” A decade later, it is clear that even centrally owned storage clouds [86] or data centers [17] will have to address similar considerations to support multiple tenants and conflicting SLOs. In a sense, the seeds were planted in the 1990s when Wilkes advocated the “host-based perspective” for storage management [113].

In the network domain, content distribution networks (CDNs) are frequently used for large-scale content delivery. Many network service providers are deploying their own CDNs. Recently, several working groups have been discussing the standardization of interfaces and specifications to facilitate federated CDNs [82, 90]. The business aspects of such cooperation between selfish, otherwise competing entities, crucial to its success [84], are yet to be determined.

Cooperative caching is a critical element in this shift in resource management. Traditionally, cooperative caching was considered in systems with
central ownership and management. Accordingly, the goal of global memory management algorithms was to optimize the entire system’s response time [37, 40, 57, 65, 95]. But caches are a major resource to be shared between cooperating entities in the emerging models we described above. This is clearly the case in the storage domain, but also in the network domain, where NSP and proxy caches serve to reduce upstream bandwidth. In these models, caches belong to different owners and administrative domains, and should primarily be used for the purpose and benefit of their owners. Therefore, it is time to start addressing caches as selfish entities, which cooperate with one another only if the benefit of doing so exceeds the cost.

Previous research focused on how cooperation affects the hit rate of participating caches. A reduction in hit rate is the most obvious possible cost of cooperation. However, additional costs may apply, according to the clients’ objective functions. One such cost is the service delay – the delay in computation incurred by a client while its CPU is busy serving peer requests. Serving peer requests also consumes the client’s limited network bandwidth. Another cost 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. Energy productivity and similar measures [5, 50] are used to characterize individual applications’ performance, representing another example of selfish objectives.

A new storage model should be used for cooperative caching, and should enable informed decisions. In such a model, participating entities should be able to weigh the costs and the benefits from each cooperative interaction, and decide, selfishly, whether to cooperate. Such a model will not only avoid performance degradation in each such entity, but will also prevent cases in which cooperation is bad for the system as a whole.
Figure 7.1: Cooperative storage model and operations

### 7.2 Requirements

The first requirement for this new model is that it explicitly define the serve operation and the cost it incurs. The cost should be defined in terms of the clients’ objective function, similar to the way existing models define the costs of read and write operations. The cost of existing operations is usually defined in terms of the disk accesses, network transfers, and queuing delays they incur.

Accordingly, when a client serves a peer request, its CPU must initiate a copy of the requested data block, after which the network transfer takes place in the background, involving the NIC and possibly the DMA engine. The corresponding cost depends on the objective function. For example, when the objective is to minimize I/O response time or application run time, the cost of serve is the time the CPU spends sending a block to a peer. During this time, the client serving the request cannot perform computation. This client experiences a shorter delay if it is idle, waiting for I/O, when the request arrives.
The second requirement is that a client be able to measure the utility it derives from the content of its own cache, and the utility it can derive from service from remote caches. Utility is stated in terms of the client’s objective function, and represents the total cost of I/O operations that are avoided by accessing data from its own caches, or from remote caches. Selfish clients strive to cooperate iff

$$Utility(\text{cache content without cooperation}) <$$

$$[Utility(\text{cache content with cooperation})$$

$$+ Utility(\text{content accessed from remote caches})$$

$$- Cost(\text{total accesses to remote caches})$$

$$Cost(\text{total serves to peers})]$$

If the system’s incentive mechanism includes some form of credit transfer for each cooperative transaction between pairs of clients, then 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 [87] and workload attributes [113]. When accurate hints are unavailable, the hit rate can be estimated via methods such as statistics gathering in separate queues [46] or ghost caches [38], analytic models [124] and active sampling [52].

By capturing the individual objectives of separate entities in the system, this model is useful not only for inducing cooperation between selfish, autonomous entities, but also for managing resources in a centralized system while adhering to service level objectives and agreements within consolidated systems.
7.3 Model definitions

Our model, depicted in Figure 7.1, consists of several clients accessing a shared, centrally owned, storage server. Each client has its own first level cache. The server includes a storage array, called the disk, and a second level cache. In addition, clients may use blocks stored in caches of neighboring clients, called peers. For the purposes of data lookup and transfer, we assume that the clients and server are placed at uniform distances from one another. 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.

We base our model on four I/O operations defined in previous studies for the purpose of maintaining cache level exclusivity. Clients read blocks that they intend to store in their cache. The server removes such blocks from its cache, to avoid redundancy [114]. Clients then demote evicted blocks to the server for second level caching. To minimize network transfers, clients can use the read-save operation [115] for blocks they do not wish to store, thus keeping them in the server's cache. Promote [44] is an alternative mechanism, in which the server probabilistically evicts blocks from its cache and promotes them to the client caches.

Adding to the above, we define 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 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.

Note the difference between the client-client and client-server relationships. While a server always sends blocks read or read-saved to the requesting client, a client may reject a peer request according to its own
selfish interests. A client will, however, serve blocks quickly from its cache, while a server might perform a slow disk access.

Our model defines three basic costs:

- \( C_{\text{net}} \) is the cost of performing a network transfer. This cost is incurred on a client whenever it receives a block from \( \text{Cache}_2 \) or from a peer, or demotes a block to \( \text{Cache}_2 \).

- \( C_{\text{disk}} \) is the cost of performing a disk access. Since all disk blocks first arrive at \( \text{Cache}_2 \), even if they are not stored there, the cost incurred by a client for bringing a block from the disk is \( C_{\text{disk}} + C_{\text{net}} \).

- \( C_{\text{serve}} \) 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.

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.

We focus on clients whose objective is to minimize their average I/O delay. However, they also aim to minimize their energy consumption. In Chapter 7.4, we explain how the costs are expressed in terms of time and energy. In our model, the server is centrally owned, and is therefore not considered a selfish entity. We assume its only objective is to minimize the overall I/O delay in the system.

### 7.4 Objective functions

We use four objective functions to quantify the delay experienced by the client, the energy it consumes, the interaction between the two, and, for the CDN scenario, the Internet accesses incurred. We explain how the costs
defined above are computed for each of our objectives.

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

- $T_{\text{net}}$ 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.
- $T_{\text{disk}}$ is the average time spent waiting for a disk access, including queuing and seek times. A block request from the disk can take less than $T_{\text{disk}}$ if the block was already being fetched when it was queued.
- $T_{\text{serve}}$ 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 $< T_{\text{serve}}$ if it is idle, waiting for I/O, when the request arrives.

Throughout our evaluation we assume that $T_{\text{serve}} < T_{\text{net}} < T_{\text{Disk}}$. For our basic setup we assign $T_{\text{serve}} = 50\mu\text{secs}$, $T_{\text{net}} = 200\mu\text{secs}$ and $T_{\text{Disk}} = 5\text{msecs}$, 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 Chapter 10.

**Energy.** To preserve energy, CPUs operate at several $C$ states, each at a different energy consumption rate, measured in watts. The CPU must be in $C_0$ to perform computation or busy waiting. $P_{\text{busy}}$ is the maximal energy consumption rate of the CPU at $C_0$. The CPU can switch to $C_2$ whenever it does not perform actual computation, but the NIC or the DMA engine are operating. The corresponding $P_{\text{net}}$ is usually smaller than $P_{\text{busy}}$. When entering a long idle period, such as while waiting for the disk, the CPU can switch to $C_4$, with a respectively low $P_{\text{idle}}$.

Switching back from a low power state to the next active state incurs additional power. Thus, the CPU may choose to stay in its current state if a long wait is not guaranteed. The decisions of when to transition to a low power state are different between CPUs, and are difficult to model due to lack of available information. Therefore, we chose to simplify our model and assume the wake-up overhead is negligible, and that the CPU always switches to the lowest state possible. We explain how this is done in the simulations
in Chapter 9.1. Below, we define the corresponding costs in terms of energy, measured in joules.

- $E_{\text{net}} = T_{\text{net}} \times P_{\text{net}}$ is the energy consumed by the CPU when performing a network transfer.

- $E_{\text{disk}} = T_{\text{disk}} \times P_{\text{idle}} + E_{\text{net}}$ is the energy spent while reading a block from the disk, including the additional transfer from $Cache_2$.

- $E_{\text{serve}} = T_{\text{serve}} \times P_{\text{busy}} + E_{\text{net}}$ is the energy consumed by a client serving a peer request. The CPU must first initiate the copy in $C_0$, and then remain in $C_2$ for the duration of the network transfer. Note that the additional network cost is incurred only if the client was originally in $C_4$. Otherwise, it can continue its computation after $T_{\text{serve}}$, transferring the block in the background.

**Energy Delay Product.** In our model, cooperation consumes energy, but may reduce I/O delay. This conflict is well known in the context of chip design, where increasing clock frequency reduces computation time but increases energy consumption. There, the *Energy Delay Product* [49] is used to express the goal of minimizing both measures simultaneously. We borrow this measure and compute it by multiplying the entire run time of a client’s workload by the energy it consumes.

**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 traces described below we measured the portion of all requested videos served from the Internet, as well as the upload bandwidth used by the clients. For the utility calculations in C-Hints, we used the operational costs of the ISP, where bandwidth within the ISP network (between all clients and between the clients to 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 deci-
sions on this metric. Thus, $C_{net} = 1$, $C_{disk} = 100$, and $C_{serve} = 0$. The cost of 
sERVE is zero because uploading a video to a peer does not affect the client’s 
viewing performance, determined by its available download bandwidth.
Chapter 8

Cooperative Caching Approaches

We suggest four approaches for coordination and management of the cooperating clients, with an increasing degree of selfishness. The algorithms are 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 8.1 summarizes the characteristics of our cooperative policies, and places them on the scale between selfish and altruistic.

8.1 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 an MD5 hash key for each block, and distributes the key space evenly between all participating clients, so that each client is re-
Most altruistic

Clients store only blocks assigned to them, and always serve peer requests

Most selfish

Clients store LRU blocks without replication, and serve peers selfishly

C-DHT

Clients store and replicate blocks according to ARC, and serve peers selfishly

ARC

Clients store blocks and serve requests according to utility computations, if beneficial

C-Hints

Figure 8.1: Cooperative policies on a scale of selfishness. Selfish approaches keep data close to the application, in the cache of the client accessing it, while altruistic approaches achieve exclusive caching. C-Hints uses additional information to achieve both

responsible for an agreed upon portion of the keys, as in Chord [104]. 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 blocks from the server.

Cache management is based on Demote, 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 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 when evicted. 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 and fixed its size at 3 blocks for the TPCH workloads, and at 20% the size of the client cache for the TPCC and video workloads. In a practical implementation, the size of the private partition can be adjusted dynamically at run time, by comparing the hit rate in the private partition and in the rest of the cache [85].
8.2 Peer-to-peer cooperative caching

C-P2P is constructed for moderately selfish clients. It adapts peer-to-peer techniques, and specifically BitTorrent [36], to storage caching. In BitTorrent, peers query a dedicated server called a tracker to learn about other peers currently 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 storage server counts accesses and demotes to each block range. For each block range, all clients that currently cache blocks from that range, compose the swarm of this range. Clients periodically query the server to track 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 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. The 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 that it may 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 evicted from client caches are demoted to the server, and blocks received from peers are not stored in the main cache. Rather, they are stored in a private partition similar to that in C-DHT. In that sense, clients participate in a global optimization when they cooperate. In addition, since the tracker provides only approximate information about peer cache content, 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 [44].

8.3 Cooperative ARC

C-ARC is constructed for strictly selfish clients and minimal server involvement. It is based on ARC for cache management. ARC [44] distinguishes between new blocks that are stored for a trial period in the $L_1$ list, and useful blocks that have been seen more than once and are stored in the main, $L_2$ list. Clients run ARC with PROMOTE, explained in Chapter 7, which is also suited for multiple clients. Clients in C-ARC store all their blocks in $L_1$ or $L_2$ according to ARC, without distinguishing between blocks received from peers or from the server. When a peer request for a block hits in $L_2$, it does not alter the block’s LRU position. However, when a block from $L_1$ 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, as close as possible to the original ARC.

Clients use the ghost cache maintained by ARC to store information about the peer which previously supplied recently evicted blocks. As in C-P2P, on a cache miss, a client attempts to request the block again from the peer that previously served it. 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 (different) random peers. If all attempts fail, 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 according to this balance.
8.4 Cooperative caching with hints

C-Hints is constructed for selfish clients that can supply hints about their future accesses, and are willing to share them with a trusted central server. The cooperation component at the server uses these hints 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 as close to zero as possible. The configuration is updated periodically according to measured behavior or updated hints.

**Initialization.** Similar to the marginal gains in $MC^2$, C-hints assumes the application supplies, for each block range $R$ accessed by client $C$, $F_C(R)$, the frequency of accesses of $C$ to blocks in $R$ in a given time frame\(^1\). 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 $C_{disk} \times F_C(R)$, the cost of disk 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 $MC^2$. 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.** As in C-P2P, where clients maintain a credit balance, 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. The balance is maintained without distinguishing which peers the client serves. This allows clients to exploit more opportunities for cooperation. A

\(^1\)The hints supplied for $MC^2$ 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.
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 in return. Therefore, the cooperation component keeps client balances within limits, and attempts to keep them as close to zero as possible.

**Maximal Utility.** In a cooperation transaction, 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 $C_{disk} \times F_P(R) - C_{net} \times F_P(R) - C_{send} \times F_P(R)$, corresponding to the amount of disk 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 when balances are guaranteed to stay within limits, both the sending client and the receiving peer benefit from cooperation and their utilities increase. The cooperation component attempts to maximize the global utility of the system, which is the sum of all client utilities.

The problem of finding an optimal configuration under both guidelines is closely related to densest subgraph [25] and small set expansion [89], which are conjectured to be hard to approximate. We therefore have reason to believe it will be hard to find an optimal cooperative configuration in our system. Instead, we construct a cooperative configuration greedily, as described by the pseudocode in Figures 8.2 and 8.3.

The cooperation component chooses clients in order of their current credit balance (line 201), and checks whether they can serve others with the ranges already stored in their cache (lines 181–191). Client $C$ can serve range $R$ to
peer $P$ if $P$ does not store $R$ in the base configuration (line 184). If $P$ already stores $R$, it can remove it from its cache and store another range $R'$ instead, under the restriction that the benefit from $R'$ entering the cache ($C_{disk} \times F_P(R')$) is larger than the cost of receiving $R$ from $C$ ($C_{net} \times F_P(R)$) plus the investment in accumulating the credit needed for receiving $R$ ($C_{send} \times F_P(R)$) (lines 186–188).

The final configuration is communicated to the clients in the form of the four arrays defined in Figure 8.2. Clients populate their caches and access blocks according to the configuration, as described in Figure 8.4. Clients always serve peer requests according to the configuration, even if the requested block is not currently in the cache, in which case the client requests the block from the peer that was responsible for this block’s range in the previous configuration (lines 231–232). If the block is not found, it is requested from the server (line 234). This lazy population of the caches ensures that the block will be at the responsible client’s cache, ready to serve subsequent requests, while a long startup delay is avoided.

The cooperation component updates the configuration when a client provides new hints which alter its utility or when a client exceeds its credit limit (line 248). 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 hints, causing clients to serve more requests than intended by the cooperation component. The search for a new configuration begins with a greedy construction according to the current credit balances (line 162). The clients update their cache content in a lazy manner, similar to the initial population.
**Input:**
- \( R \) ranges
- \( C \) clients accessing \( Cache_2 \) // also called peers
- \(|Cache_2| = m\)
- \( T \) // the number of time frames a configuration is supposed to stay valid for
- \( \forall c, r F_c(r) // \) (hinted) number of accesses by \( Client_c \) to blocks in \( Range_r \)
- \( MaxBalance // \maximal \) credit a client may accumulate
- \( MinBalance = -MaxBalance // \minimal \) (negative) credit a client may accumulate
- \( CanServe // \) list of clients in increasing order of their account balance
- \( CanReceive // \) list of clients in decreasing order of their account balance
- \( ClientSavesRange(C)[R] \) // 0: No, n: Saves and serves n-1 peers.
- \( ClientReceivesRange(C)[R] = \{NoClient\} // [i][j] == k \iff\) Client \( i \) receives \( Range_j \) from Client \( k \)
- \( ClientServesRange(C)[R][C] = \{0\} // [i][j][k] == TRUE \iff\) Client \( i \) stores \( Range_j \) and serves Client \( k \)
- \( ConfigBalance(C) = AccountBalance(C) // \) initialized to \( \{0\} \) only on the first call

**NextRangeToSave(c):**
- return \( Range_r \) with highest \( F_c(r) \) for which \( (F_c(r) \neq 0) \)
  & \( (ClientReceivesRange(c)[r] == NoClient) \) & \( (ClientSavesRange(c)[r] == 0) \)

**NextRangeToServe(c):**
- return \( Range_r \) with lowest \( F_c(r) \) for which \( (F_c(r) \neq 0) \)
  & \( (ClientSavesRange(c)[r] == 1) \) and was still not returned

**InsertToLists(c):**
- if \( ((Client_c \text{ accesses ranges that it does not store or receive}) \)
  & \( (ConfigBalance[c] > MinBalance) \)
  insert \( Client_c \) to \( CanReceive \) according to \( ConfigBalance[c] \)
- if \( ((Client_c \text{ stores ranges it did not try to serve others}) \)
  & \( (ConfigBalance[c] < MaxBalance) \)
  insert \( Client_c \) to \( CanServe \) according to \( ConfigBalance[c] \)

**RemoveFromLists(c):**
- if \( (Client_c \in CanServe) \)
  Remove \( Client_c \) from \( CanServe \)
- if \( (Client_c \in CanReceive) \)
  Remove \( Client_c \) from \( CanReceive \)

**PeerCanReceive(p, r, r'):** // will \( Peer_p \) benefit from receiving \( Range_r \)?
- if \( (F_p(r') > 0) // \) needs range
  & \( (ClientReceivesRange[p][r] == NoClient)) // \) does not receive from anyone
  if \( (ClientSavesRange[p][r] == 0) // \) does not save range
    return TRUE
  else if \( ((ClientSavesRange[p][r] == 1) // \) saves range without serving
    // will benefit from saving another range
    if \( (C_{disk} \times F_p(r') > C_{net} \times F_p(r) + C_{serve} \times F_p(r)) \)
      return TRUE
  else
    return FALSE

Figure 8.2: Pseudocode for the construction of a cooperative configuration in C-Hints (Part I).
192. AllocateServerCache():
193.     foreach Range\_r
194.     \( F'(r) = 0 \)
195.     foreach Client\_c
196.         if (ClientReceivesRange[c][r] == NoClient) & ClientSavesRange[c][r] == 0)
197.             \( F'(r) += F_c(r) \)
198.         Allocate space in server cache for the \( m \) ranges with highest \( F'(r) \)

199. GreedyConstruct():
200.     while (CanServe \( \neq \varnothing \)) & (CanReceive \( \neq \varnothing \))
201.         Client\_c = first client in CanServe
202.         RemoveFromLists(c)
203.         Range\_r = NextRangeToServe(c)
204.         foreach Peer\_p \in CanReceive
205.             RemoveFromLists(p)
206.             Range\_r' = NextRangeToSave(p)
207.             if PeerCanReceive(p,r,r')
209.                 ClientSavesRange[c][r][p]++ // update configuration
210.                 ClientServesRange[c][r][p] = TRUE
211.                 ClientReceivesRange[p][r] = c
212.             if (ClientSavesRange[p][r] == 1) // space is saved, peer stores next range
213.                 ClientSavesRange[p][r][r'] = 1
214.         InsertToLists(p)
215.         if (ConfigBalance[c] > MaxBalance) // can client serve more peers?
216.             break // moving on to next client
217.         InsertToLists(c)
218. AllocateServerCache()

Figure 8.3: Pseudocode for the construction of a cooperative configuration in C-Hints (Part II).
219. **Evict(c):**
220. \( Range_r = \text{range with lowest } F_c(r) \text{ for which ClientSavesRange}[c][r] == 0 \)
221. remove block \( Y \) from \( Range_r \) according to replacement policy of \( Range_r \)
222. if \((\text{ClientReceivesRange}[c][r] == p)\)
223. send \( Y \) to \( Peer_p \) // lazy population
224. else
225. discard \( Y \)

226. **Client\(_c\) accesses block \( X \) of \( Range_r \):**
227. Cache hit:
228. update stacks
229. Cache miss:
230. if \((\text{ClientSavesRange}[c][r] \geq 1) // \text{client is range owner}\)
231. if \((\text{OldClientReceivesRange}[c][r] == p) // \text{maybe previous owner still has it}\)
232. request \( X \) from \( Peer_p \)
233. else // or if request rejected
234. read \( X \) from \( Cache_2 \)
235. **Evict(c)**
236. store \( X \) in cache
237. else
238. if \((\text{ClientReceivesRange}[c][r] == p) // \text{a peer is responsible for serving}\)
239. request \( X \) from \( Peer_p \)
240. else
241. read-save \( X \) from \( Cache_2 \)
242. discard LRU block \( Y \) from private partition and store \( X \) there

243. **Client\(_c\) serves block \( X \) to \( Peer_p \):**
244. send \( X \) to \( Peer_p \)
245. // credit accumulation and balance checks can be performed periodically
246. AccountBalance\([c]\)++ // update account balances
247. AccountBalance\([p]\)--
248. if \((\text{AccountBalance}[c] > T \times \text{MaxBalance}) || (\text{AccountBalance}[p] < T \times \text{MinBalance})\)
249. GreedyConstruct()

250. **Peer\(_p\) requests block \( X \) of \( Range_r \) from \( Client\(_c\):**
251. if \((\text{ClientServesRange}[c][r][p] == \text{TRUE}) // \text{client is responsible}\)
252. \( \text{Client\(_c\) accesses block } X \text{ of } Range_r \) // page if necessary
253. serve \( X \) to \( Peer_p \)
254. else if \((\text{OldClientServesRange}[c][r][p] == \text{TRUE}) // \text{lazy population}\)
255. Cache hit:
256. serve \( X \) to \( Peer_p \)
257. Cache miss:
258. reject // no paging
259. else
260. reject

---

Figure 8.4: Pseudocode for cooperation protocol and lazy population in C-Hints. The arrays with the prefix “Old” represent the previous configuration, which was stored before constructing the current one (only one previous version is stored).
Chapter 9
Experimental Testbed

9.1 Simulation environment

We use a simulation environment to experiment with a variety of storage configurations, described in detail in Chapter 10. Our simulator is a single threaded C program composed of approximately 20,000 lines of code. Caches are represented as separate modules, each with an adjustable buffer size and cache management algorithm. I/O requests are represented as timed events. Each cache has three event queues. The request queue holds I/O requests received from an upper level cache, from a peer cache or from the application. The wait queue holds requests that are waiting for response from a lower level cache or from a peer cache. The completion queue holds requests that have been handled by a lower level cache or by a peer cache, and whose completion time has been determined accordingly.

Clients are represented by hosts, each with its own input trace and CPU module. Each host is attached to a single, first level, cache. Trace lines represent I/O requests, specifying a logical block number and a delta – the time that passed in the original execution between this request and the previous one. Hosts create one event at a time, setting its request time by adding the delta of the new request to the completion time of the previous request. The
event is then queued at the request queue of the respective first level cache.

The simulator is an iterative program, handling one event in each iteration. At the beginning of each iteration, the simulator chooses the event with the closest request or completion time, and advances the simulation clock to that time. The event is handled by the cache it is queued in, according to the content of the cache and the management algorithm it implements. Events in the request queue may trigger a cache hit or a cache miss. On a cache hit, the event is returned to the completion queue either of the cache in the upper level, of the requesting host, or of the cache of the requesting peer. On a cache miss, the event is queued in the wait queue while the request is propagated, in the form of a new event, to the lower level cache or to a peer. Events in the completion queue are handled in two phases. In the first phase, the cache content and state is updated in the current cache. In the second phase, the waiting event – the event that triggered this event – is handled: its completion time is set, and it is moved from the wait queue to the completion queue of the cache it was waiting at.

For energy computations, the CPU modules in the hosts hold three accumulators to measure the time spent in each of the CPU states described in Chapter 10.3.6. In commercial systems, CPUs transition to a lower power state when they predict a long idle period. The exact algorithms are proprietary and vary between systems. Therefore, we do not attempt to simulate the online decisions of such algorithms. Instead, the simulator updates the CPU state in retrospect, after the idle period has ended, and only if it was long enough to justify a transition into $C_2$ or $C_4$. We assume that the CPU is performing its application’s tasks, and is therefore in state $C_0$, for the entire delta between requests. When the simulation completes, the energy consumed by each host is computed by multiplying the time spent in each CPU state by the respective energy consumption rate. The energy consumption rates used in our simulations are summarized in Table 10.5.

The simulator outputs, for each trace, the hit rate in each cache, the
bandwidth usage by each host, the total and average I/O response time, service time, energy and energy delay product. It also computes the global average and standard deviation in each of the measures for all traces in each execution.

While our solution is designed to work in \( n \) levels of cache, we compare it to existing policies on a testbed consisting of two cache levels \((n = 2)\). We simulate a client cache attached to each client in the first level and a shared second level cache.

We simulated the existing policies described later in this section, Karma, \( MC^2 \) and our cooperative approaches on a series of increasing cache sizes. For all experiments in the non cooperative model (excluding the one depicted in Figure 10.7), we set the access costs \( C_1 = 1\mu s, C_2 = D_2 = 200\mu s \) and \( C_{Disk} = 10000\mu s \). Since every I/O request incurs a first level access, we chose the minimal cost for \( C_1 \). In our experience, the value of \( C_2 \) best represents the cost of a cache hit to a modern, storage controller. \( C_3 \) represents the cost of a disk access requiring a seek to a SATA drive. For the experiments with the cooperative model we set \( C_{Disk} = 5000\mu s \), as described in Chapter 7.4.

For all experiments, excluding the one depicted in Figure 10.8, we assume, as in previous studies \([30, 114]\), that the caches are of equal size \((S_1 = S_2)\). The cache size in the graphs for the non cooperative model refers to the aggregate cache size as a fraction of the size of the data set. Thus, a cache size of 1 means that each cache level is big enough to hold one-half of the data set. A cache size of 2 means that the entire data set fits in the top level. The results in Chapter 10.3 refer to the size the cache of a single client, also as a fraction of the size of the data set. 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.

Some of our figures depict the results normalized to their average I/O response time when using LRU. This gives a better representation of the
improvement over the default LRU management, making it easier to compare
the policies over several related traces.

A useful tool for evaluating a cache replacement policy is comparison to
the performance of an optimal offline policy. The optimal policy for a single
cache is Belady’s MIN [23], which evicts the block which will be accessed
again furthest in the future. The optimal replacement policy for a multilevel
cache in a model with DEMOTE is unknown. Since we are interested in the
I/O response time experienced by such a policy, we use Gill’s lower bound
on the optimal I/O response time for a single client [44] in our experiments
with the non cooperative model. The computation of this lower bound is
described in detail in Chapter 12.

9.2 PostgreSQL database

We chose PostgreSQL [76] as a source of application hints. Each of its data
files (table or index) is divided into disk blocks. Accesses to these blocks
were traced by adding trace lines to the existing debugging mechanism.

Like most database implementations, PostgreSQL includes an explain
mechanism, revealing to the user the plan being executed for an SQL query.
This execution plan determines the pattern and frequency with which each
table or index file will be accessed during query execution. We used the
output of explain to supply Karma and $MC^2$ with the characterization of
the access pattern for each range, along with a division of all the blocks into
ranges. Blocks are first divided by their table, and in some cases, a table
can be sub-divided into several ranges. For example, in B-tree indices each
level of the tree is characterized by different parameters (as in Figure 4.1).
The hints were manually generated in our simulation environment. In a
production system, a similar process can be performed automatically.

In the experiments reported in Chapter 10 (Sections 10.1 and 10.2), the
data set consisted of approximately 30 ranges. The resulting metadata fit
Table 9.1: Summary of workload characteristics

in one cache block, deducted from the cache size available to each client. In the experiments with the cooperative approaches we divided the data into 512 ranges, to facilitate fine grained cooperation transactions. The resulting metadata, described in Figure 8.2 fit in 200 KB (approximately 24 cache blocks), deducted from the cache size available to each client.

9.3 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. Table 9.1 summarizes the characteristics of our traces.

9.3.1 TPC benchmark traces

We performed our simulation with several sets of input traces that we generated beforehand. For the traces, we use two benchmarks defined by the Transaction Processing Council. The first is TPCH, a decision support
benchmark which exemplifies systems that examine large volumes of data and execute queries with a high degree of complexity [2]. In our experiments we used the default implementation provided in the benchmark specification to generate both the raw data and the queries.

We simulated our cache replacement policies on two types of traces:

- Repeated queries: each query is repeated several times, requesting different values in each run (execution). This trace type models applications that access the database repeatedly for the same purpose, requesting different data in different iterations.

- Query sets: each query set is a permutation of the 22 queries of the benchmark, executed serially. The permutations are defined in the benchmark specification. The query sets represent applications that collect various types of data from the database.

Queries 17, 19, 20, 21 and 22 have especially long execution traces (each over 25,000,000 I/Os). As these traces consist of dozens of loop repetitions, we used for our simulation only the first 2,000,000 I/Os of each trace.

In the multiple-client setting we use TPCH traces in two different experiments. The first examines several clients sharing a second level cache, each executing a stream of one repeated query. We experimented with 7 combinations of 2 to 4 clients and queries, with varying degrees of data correlation between the queries. In the second experiment, each client executes a different query set.

The second benchmark is TPCC, an on-line transaction processing (OLTP) benchmark, involving a mix of concurrent transactions of different types and complexity [1]. This workload consists mainly of index accesses, with a different access skew to each table. Therefore, recency of access is the best indication that a block will be accessed again. We generate the TPCC traces using TPCC-UVa [73], an open-source implementation of the benchmark. Since this implementation is essentially serial, for the multiple client setting
we generate ten client traces separately and then run them concurrently on our simulator.

9.3.2 Video playback.

The video 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. The simulation parameters correspond to the original setup.

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. 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.

Hints for $MC^2$ and C-Hints were generated as follows. We assumed the ISP cache collects access statistics to the most popular videos. At the beginning of each week, hints were generated from statistics collected on the last day of the previous week, for 3% of the most accessed videos on this day. The hints represent the access distribution at the ISP, and are not necessarily accurate for each individual client. For utility calculations in C-Hints we used the costs described above. Although these costs represent the objective of the ISP and not of the individual clients, they can be incorporated into the clients' SLAs. Therefore, we believe it is reasonable to base the clients' selfish decisions on this metric.

9.3.3 Synthetic Zipf workload

In traces with Zipf distribution, the frequency of access to block $i$ is proportional to $1/i^\alpha$, for $\alpha$ close to 1. Such distribution approximates common
access patterns, such as file references in Web servers. Following previous studies [83, 114], we chose Zipf as a non-trivial random workload, where each block is accessed at a different, yet predictable frequency. We use this trace in the single-client experiments, with settings similar to those used in previous work [114] and set $\alpha = 1$, for 25,000 different blocks.

Karma does not require information about the access frequency of each block. We supplied it with a partitioning of the blocks into several ranges and the access frequency of each range. This frequency can be easily computed when the distribution parameters are available.

The blocks in each range have different access frequencies: nonetheless, for any two blocks $i$ and $j$, if $i < j$, then the frequency of access to block $i$ is greater than that of block $j$. The blocks are assigned to ranges in increasing order of block number, and so for any two ranges $I$ and $J$, if $I < J$, then all the blocks in range $I$ have greater access frequencies than the blocks in range $J$.

### 9.4 Comparative framework

We compared $MC^2$ to four existing replacement algorithms: LRU, ARC, MultiQ, and LRU-SP, each of which we examined using three different approaches: Basic, Double, and Global. In the Basic approach, each algorithm is used in conjunction with LRU, where the existing algorithm is used on one level and LRU on the other, as was defined in the respective literature. In the Double approach, each is used on both cache levels. The third approach is a global management policy, where each algorithm must be explicitly adapted to use DEMOTE.

Although Global-ARC was not actually implemented in our experiments, we describe, in what follows, how it would work in a multilevel cache. It is also important to note that Global-MultiQ is not an existing policy: we defined this algorithm for the purpose of extending the discussion, and it is

69
<table>
<thead>
<tr>
<th>Alg</th>
<th>Basic-Alg</th>
<th>Double-Alg</th>
<th>Global-Alg</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>Basic-LRU (LRU+LRU)</td>
<td>Double-LRU</td>
<td>Demote</td>
</tr>
<tr>
<td>ARC</td>
<td>Basic-ARC (LRU+ARC)</td>
<td>Double-ARC</td>
<td>Global-ARC</td>
</tr>
<tr>
<td>MultiQ</td>
<td>Basic-MultiQ (LRU+MultiQ)</td>
<td>Double-MultiQ</td>
<td>Global-MultiQ</td>
</tr>
<tr>
<td>LRU-SP</td>
<td>Basic-LRU-SP (LRU-SP+LRU)</td>
<td>N/A</td>
<td>(SP-Karma)</td>
</tr>
</tbody>
</table>

Table 9.2: The policies in our comparative framework. In addition to the policies known from the literature, we defined the Double and Global extensions indicated in bold. We compared Karma to all applicable Basic and Double policies, as well as to Demote and to Global-MultiQ.

implemented here for the first time. We refer to the special case of LRU-SP in Chapter 9.5. The algorithms and approaches are summarized in Table 9.2. The actual experimental results are described in Chapter 10.

**Least Recently Used (LRU).** LRU is the most commonly used cache management policy. **Basic-LRU** and **Double-LRU** are equivalent, using LRU on both cache levels. **Global-LRU** is the Demote policy [114], where the upper level cache demotes all blocks it discards. The lower level cache puts blocks it has sent to the upper level at the head (closest to being discarded) of its LRU queue, and puts demoted blocks at the tail.

**ARC.** ARC [75] divides blocks between two LRU queues, \( L_1 \) and \( L_2 \). \( L_1 \) holds blocks requested exactly once. \( L_2 \) holds blocks requested more than once. The bottom (LRU) part of each queue is a ghost cache. The percentage of cache space allocated to each queue is dynamically adjusted, and history is saved for as many blocks as would fit in twice the cache size. As ARC was originally designed for a second level cache, **Basic-ARC** uses LRU at the first cache level and ARC at the second. **Double-ARC** uses ARC at both cache levels. **Global-ARC** keeps the ghost caches in the second level cache, as well as the LRU part of what is left of \( L_1 \) and \( L_2 \). An equivalent policy,
DEMOTE-ARC, is used for evaluation purposes in [44]. ARC is implemented for each cache level with dynamic adjustment of the queue sizes [75]. For the cooperative experiments we implemented the more recently suggested ARC with Promote [44].

**MultiQ.** MultiQ [121] was originally designed for a second level cache. It uses multiple LRU queues, each having a longer lifetime than the previous one. When a block in a queue is accessed frequently, it is promoted to the next higher queue. On a cache miss, the head of the lowest non-empty queue is evicted. Basic-MultiQ uses LRU at the first cache level and MultiQ at the second. Double-MultiQ uses MultiQ at both cache levels. We implemented MultiQ for each cache level with 8 queues and a ghost cache. The Lifetime parameter is set according to the observed temporal distance. We extended MultiQ to Global-MultiQ in a straightforward way. The ghost cache is allocated in the second level cache, and the queues are divided dynamically between the cache levels, with at most one queue split between the levels. Whenever a block is brought into the first level cache, the block at the bottom of the lowest queue in this level is demoted to the second level cache.

In the non-cooperative multiple-client experiments, the Double-extensions of ARC and MultiQ performed better than the Basic-ones. Therefore, only the results for the Double-extensions are presented in the figures in Section 10.2. The Global-extensions are not applicable in a multiple-client setting [44].

### 9.5 Application controlled file caching

In LRU-SP [28], applications may use specific interface functions to assign priorities to files (or ranges in files). They may also specify cache replacement policies for each priority level. Blocks with the lowest priority are first candidates for eviction. In the original paper, applications were modified to include calls to these interface functions.
As a policy assuming hints from the application, LRU-SP is designed for a first level cache. Therefore, we implement *Basic-LRU-SP* with LRU at the second level. *Double-LRU-SP* would let the application manage each cache level directly yet independently. This seems unreasonable and thus we did not implement this extension. Without hints available at the second level cache, the simple addition of *demote* will not result in global management, since informed decisions cannot be made in the second level. Thus, extending LRU-SP to *Global-LRU-SP* is not applicable.

**A new multilevel extension.** To evaluate the contribution of the different mechanisms of Karma, we defined a new policy, *SP-Karma*, for managing multilevel caches, which added to LRU-SP most of the features we defined in Karma. This extension resulted in a new cache management algorithm which is similar to Karma and allows us to better understand the value of specific mechanisms in Karma. In particular, we added *demote* for cooperation between cache levels, we derived priorities using Karma’s calculation of normalized marginal gains (this mechanism was also used to supply hints to Basic-LRU-SP above), and we supplied these priorities to both cache levels. Since SP-Karma now bases its decisions on Karma’s priorities, the significant difference between our two policies is the use of *read-save*.

The results of SP-Karma resembled those of Karma, but Karma achieved better performance on all traces. The use of *read-save* resulted in Karma executing fewer *demote* operations, thus outperforming SP-Karma by up to 1% in the large caches and 5% in the small ones. Since we defined our new policy, SP-Karma, to be very similar to Karma, and since it shows similar results, in Chapter 10 we compare only Karma to the existing policies.
Chapter 10

Results

In our experimental evaluation we wish to determine the effect of sharing application information between components and levels in the cache hierarchy. In the first part of our evaluation we determine whether $MC^2$ is better than existing policies for managing a cache hierarchy with multiple clients. Our results will provide answers to the following questions:

- Does $MC^2$ achieve the shortest I/O response times?
- Does $MC^2$ benefit from data sharing more than existing policies?
- Does $MC^2$ avoid destructive sharing better than existing policies?
- Does $MC^2$ maintain a high degree of fairness between clients?

In order to answer the first question, we first provide a detailed analysis of Karma, our algorithm for a single client, on a multilevel hierarchy corresponding to the model in Figure 2.1(b).

In the second part of our evaluation we evaluate, based on our new model, the effect that client selfishness has on performance and answer the following questions: is cooperative caching really helpful? In other words, we wish to schematically fill the blank graph in Figure 10.1.
Selfishness

Figure 10.1: What effect does client selfishness have on performance?

Performance

Selfishness

Figure 10.2: Karma’s improvement over LRU on Query 3 run multiple times.

10.1 Karma

How does Karma compare to Basic-LRU? We ran all policies on traces of each query, using several cache sizes. Karma generally yielded similar curves for all queries. In Figure 10.2 we take a close look at Query 3 as a representative example, in order to understand the behavior of Karma with different cache sizes. Query 3 is sequential, scanning 3 tables of different sizes, each with a different frequency. PostgreSQL creates a hash table for each database table and joins them in memory. Subsequent runs of Query 3 result in a looping reference. Karma is supplied with the size and access
Figure 10.3: Average I/O response time for all policies on Query 3 run twice. Policies with identical results are plotted using a single line. Other than Karma, only the Global policies are able to fit the entire data set in the cache when the aggregate cache size equals the size of the data set. Unlike the non-informed policies, Karma and Basic-LRU-SP reduce their I/O response time gradually as the cache size increases.

frequency of each table. The largest table, whose size is more than half the data set size, has the highest marginal gain. It is therefore allocated all the available cache space when the aggregate cache size is half the data set or smaller.

We ran Query 3 eight times, to show how the average I/O response time incurred by Karma decreases as the cache size increases. The cost of the compulsory misses in the first run of the query is similar for Karma and for LRU. However, the average I/O response time of subsequent accesses is significantly lower with Karma. LRU suffers from the three problems of multilevel caching. The second level cache experiences no locality of reference, as all repeated accesses are hidden by the first level. Even when the aggregate cache is as large as the data set, LRU does not exploit the entire space avail-
Table 10.1: The base case: average I/O response time for each query with LRU (µsec). N/A - we ran the long, concatenated queries (see Chapter 9.3.1) only once.

<table>
<thead>
<tr>
<th>Query</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 run</td>
<td>10136</td>
<td>740</td>
<td>10157</td>
<td>2897</td>
<td>2427</td>
<td>10142</td>
<td>1425</td>
<td>1737</td>
<td>925</td>
<td>10135</td>
<td>10033</td>
</tr>
<tr>
<td>2 runs</td>
<td>10136</td>
<td>726</td>
<td>10157</td>
<td>2897</td>
<td>2435</td>
<td>10142</td>
<td>1423</td>
<td>1736</td>
<td>1060</td>
<td>10135</td>
<td>10033</td>
</tr>
<tr>
<td>4 runs</td>
<td>10136</td>
<td>700</td>
<td>10157</td>
<td>2907</td>
<td>2431</td>
<td>10142</td>
<td>1434</td>
<td>1706</td>
<td>969</td>
<td>10135</td>
<td>10033</td>
</tr>
</tbody>
</table>

able due to redundancy between levels, and all the blocks have to be fetched from the disk. Finally, when the cache is smaller than the data set, LRU is unable to maintain used blocks until the next time they are accessed, and so it does not benefit from increasing the cache size. Karma does not suffer any of those drawbacks, and its I/O response time decreases significantly as the cache size increases. Although the portion of the loop which fits in the cache is equal for all runs (for each cache size), the hit rate of LRU remains zero as long as the first level cache is smaller than the data set, while Karma’s hit rate increases in both cache levels.

How does Karma compare to single level non-informed policies?

Figure 10.3 depicts the results for all policies on Query 3. The Basic implementations of MultiQ and ARC behave like LRU. Their division into separate LRU queues does not help in identifying the looping reference if it is larger than the cache size.

The Double extensions of MultiQ and ARC allocate a ghost cache (whose size in our implementation is 0.16% and 0.43% of the cache size, respectively) in both cache levels. This small portion is sufficient to prevent the data set from fitting in one level when the cache size is 2, leading to performance similar to that of LRU on half the cache size. Note, however, that allocating the ghost cache in a separate buffer, or, alternatively, increasing the cache size accordingly, would eliminate this effect.
Figure 10.4: The average I/O response time of Karma, Basic-ARC, Demote and Basic-LRU-SP is compared to LRU, on repeated runs of all queries. The columns show the improvement of these policies for each query type, when the aggregate cache size equals the size of the data set. Each column is tagged with the standard deviation (%) for this query type. A zero standard deviation means the improvement was equal for all queries of this type. The I/O response time of Karma was lower than that of LRU on repeated runs of the queries by an average of 59%, 64%, 70% and 96% on query types a, b, c, and d, respectively.
Karma is informed of the looping reference and manages it with MRU replacement. Like the optimal policy, Karma benefits from every increase in cache size. The average I/O response times it incurs are very close to the theoretical lower bound. When the entire loop fits in the aggregate cache, Karma benefits from its exclusive caching and shows the largest improvement over the average I/O response time incurred by LRU. We refer to the other global and informed policies later in this section.

The results for the other queries are similar to those for Query 3. Figure 10.4 summarizes the results for multiple runs of each query, comparing the average I/O response time of Karma to three representative policies. Each query was executed four times, with different (randomly generated) values. Different values result in dissimilar traces, but similar execution plans. The traces are a concatenation of one to four of those runs, and the results are for an aggregate cache size which can contain the entire data set. Most of the queries were primarily sequential, causing all policies to perform like LRU for a single run. Subsequent runs, however, convert the sequential access into a looping access, which is handled poorly by LRU when the loop is larger than the cache. These experiments demonstrate the advantage of policies which are not based purely on recency.

The queries were of four basic types, and the results are averaged for each query type. While the average I/O response time for each query was different, the improvement of the policies over LRU was similar for each query type. Therefore, the results are represented as the average I/O response time compared to that of LRU. The I/O response times of LRU for each independent query are listed in Table 10.1. The first type includes queries which consist only of sequential table scans. Karma’s I/O response time was shorter than that of LRU and LRU-based policies on these queries by an average of 59% on four runs (Figure 10.4(a)). In the second type, an index is used for a table scan. Indices in PostgreSQL are constructed as B-trees. An index scan results in some locality of reference, improving LRU’s performance
on query traces of this type, compared to its performance on queries with no index scans. Karma’s I/O response time on these queries was shorter than that of LRU by an average of 64% on four runs (Figure 10.4(b)). In the third query type, one or more tables are scanned several times, resulting in a looping pattern within a single execution of the query. In queries of this type, I/O response time lower than that of LRU is obtained by some policies even for one run of the query. Four runs of each of these queries consisted of eight runs of the loop, resulting in significant reduction in I/O response time as compared to the other query types. Karma’s I/O response time on these queries was lower than that of LRU by an average of 70% on four runs (Figure 10.4(c)). In the last type, each query instance included dozens of iterations over a single table. These traces were trimmed to the first 2,000,000 I/Os, which still contained more than ten loop iterations. On a single run of these queries Karma performed better than LRU by an average of 96% (Figure 10.4(d)). In all query types, Karma’s I/O response times were very close to the theoretical lower bound.

The results in Figure 10.4 for Basic-ARC correspond to those in Figure 10.3. Like the rest of the LRU-based policies constructed for a single cache level, it is not exclusive, and thus it is unable to exploit the increase in cache size and fit the data set in the aggregate cache. We refer to the performance of Demote and LRU-SP in this experiment later in this section.

To show how the different policies perform on more heterogeneous traces, we use the query sets described in Chapter 9.3.1, which are permutations of all queries in the benchmark. Our results for the first 20 sets are shown in Figure 10.5 for the largest cache sizes, where the policies showed the most improvement. The left- and right-hand columns show results for an aggregate cache size that can hold half of the data set or all of it, respectively. The results in this figure are represented as the average I/O response time compared to that of LRU, for the same reasons as in Figure 10.4.

The average I/O response time of Basic-ARC and Double-ARC is not
much shorter than that of LRU. ARC is designed to handle traces where most of the accesses are random or exhibit locality of reference. Its dynamic adjustment is aimed at preventing looping and sequential references from polluting the cache. It is not designed to handle traces where larger segments of the trace are looping or sequential. When the cache size increases from 1/2 to 1, the difference in the I/O response times of Double-ARC and LRU decreases: LRU response time is shorter because it is able to store the entire data set in the cache, while Double-ARC allocates some of its space for a ghost cache.

The average I/O response time of Basic-MultiQ is shorter than that of LRU for the larger aggregate cache. It benefits from dividing blocks into multiple queues according to access frequency. Double-MultiQ, however, suffers when the ghost cache is increased in two cache levels and does not show average improvement over Basic-MultiQ. The high standard deviation of the Basic and Double extensions of MultiQ demonstrate its sensitivity to the order of the queries in a query set, and consequently, its poor handling of transitional stages.

Karma outperforms all these policies. Its reduction in I/O response time is significantly better than that of the non-informed single level policies. This reduction is evident not only when the entire data set fits in the cache, but in smaller cache sizes as well, and is close to that of the theoretical lower bound. The low standard deviation shows that it copes well with changing access patterns resulting in transitional stages. In Karma, blocks are immediately assigned to new partitions according to their new priorities. In the other policies, blocks remain in their present queues until they are actually accessed (if the new priority is higher), or until enough time passes (if the new priority is lower).

Figure 10.6 shows how Karma compares to existing policies on a Zipf workload. We chose to present the results for Double-ARC because it showed better performance than all non-informed policies that were designed for a
Figure 10.5: Average I/O response time of Karma and the existing policies on 20 query sets, as compared to LRU. The columns show the average I/O response time for each policy averaged over 20 query sets for each of two cache sizes. The left bar for each policy is for an aggregate cache size of 1/2 and the right one is for 1. Each column is tagged with the standard deviation (%) for this policy. The policies are sorted in descending order of their I/O response time for the large cache size. Karma shows improvement over all cache sizes by combining knowledge of the access pattern with exclusive caching. It improves the I/O response time of LRU by 51% and 87% for the small and big cache sizes, respectively.
Figure 10.6: Average I/O response time for selected policies on a Zipf workload. Even when the access pattern exhibits strong temporal locality, Karma outperforms all LRU-based policies and the basic hint-based policy. Karma saves between 26% and 41% of the I/O response time incurred by LRU.

single cache level. This is because ARC avoids caching of blocks that are accessed only once in a short period of time. In a Zipf workload, these are also the blocks which are least likely to be accessed again in the near future. Note that ARC’s improvement is most evident when the cache is small. When the cache is larger, such blocks occupy only a small portion of it in LRU, and so the benefit of ARC is less notable.

Karma saves as much as 42% of the I/O response time incurred by LRU, and as much as 29% of the I/O response time incurred by Double-ARC. Since the access frequency of the blocks is available to Karma, it does not bring into the cache infrequently accessed blocks. Exclusiveness further increases its effective cache size, resulting in improved performance, closer to the theoretical lower bound than all the online policies.

How does Karma compare to global non-informed policies? When run on Query 3, Demote (Global-LRU) (Figure 10.3) exhibits different behavior than the single level policies. It manages to reduce the I/O response
time significantly when the entire data set fits in the aggregate cache. This is the result of the exclusive caching achieved by using demote. Still, for smaller cache sizes, it is not able to “catch” the looping reference and does not benefit from an increase in cache size. Instead, its overhead from futile demote operations means that its performance is worse than LRU’s for all cache sizes in which the aggregate cache cannot contain all the blocks. We expected Global-MultiQ to perform at least as well as Demote. However, even when the aggregate cache size equals the size of the data set, its large ghost cache prevents it from keeping the entire data set in the cache. Instead, it only suffers the overhead of futile demote operations. Being a global policy, Karma is able to exploit the entire aggregate cache. Since it manages the looping accesses with MRU replacement, it improves gradually as the cache size increases.

Figure 10.4 shows only the results for the largest cache size, where Demote shows its best improvement. In fact, we expect any global policy to achieve this improvement when the entire data set fits in the aggregate cache. Even there, Karma achieves lower I/O response times than Demote. This is especially evident in Figure 10.4(c). Thanks to its use of read-save and MRU management for loops, the first level cache in Karma has a hit rate of 50%. In Demote, all I/O requests are served from the second level cache. Unlike Demote, Karma achieves this advantage in smaller cache sizes as well.

The performance of the global policies on the query sets is represented well in Figure 10.5. It is clear that when the cache size is smaller than the data set, these policies are not able to improve the I/O response time of LRU significantly. Improvement is achieved only when the entire data set fits in the aggregate cache (represented by the right-hand column). Karma, however, improves gradually as the cache size increases. Combining exclusive management with application hints enables it to maximize the cache performance in all cache sizes.

When the access pattern does not include looping references (Figure 10.6),
the global policies improve gradually as the cache size increases. Although a Zipf workload exhibits significant locality of reference, adding exclusiveness to LRU does not achieve good enough results. In the smallest cache size, Demote and Global-MultiQ improve over the I/O response time of LRU by 13%. Karma, with its knowledge of access frequencies, achieves additional improvement of 23%.

**How does Karma compare to hint-based policies?** Basic-LRU-SP using Karma’s hints and priorities performs better than any non-informed single level policy, for all our traces. Using hints enables it to optimize its use of the upper level cache. When the access pattern is looping, the combination of MRU management in the upper cache and default LRU in the lower results in exclusive caching without the use of demote. Note the surprising effect on the queries with many loops (Figure 10.4(d)), where Basic-LRU-SP outperforms Karma when the entire data set fits in the aggregate cache. Karma pays for the use of demote, while Basic-LRU-SP enjoys “free” exclusive caching, along with the accurate priorities generated by Karma. The average I/O response time of Basic-LRU-SP is 93.7% lower than that of LRU. Karma’s average I/O response time is 92.8% lower than that of LRU. When the aggregate cache is smaller than the data set, or when the pattern is not a pure loop (figures 10.3, 10.4(a-c), 10.5, and 10.6), Karma makes optimal use of both caches and outperforms Basic-LRU-SP significantly.

**How is Karma affected by the model parameters?** We wanted to estimate Karma’s sensitivity to varying access costs in different storage levels. Figure 10.7 shows how Karma behaves on query set 1 (the behavior is similar for all traces) when the disk access delay ranges from 10 to 100 times the delay of a read from the second level cache (or a demote to that cache). The results in this figure are represented as the average I/O response time compared to that of LRU, for the same reasons as in figures 10.4 and 10.5. For a larger delay in disk access, the “penalty” for demote is less detrimental than the decrease in the number of disk accesses. When demote is only ten
Figure 10.7: Karma’s average I/O response time compared to that of LRU for different disk access delays, for query set 1 (the standard deviation of all traces was always less than 2%). When delays for a disk access are longer, using DEMOTE is of greater benefit, despite its additional cost.

times faster than a disk access, its added cost outweighs its benefits in very small caches.

**How is Karma affected by the cache size at each level?** Figure 10.8 compares the behavior of Karma to other policies on cache hierarchies with varying lower-level (Cache$_2$) sizes. The details are numerous and so we present here only results for the best LRU-based policies, Double-ARC and Global-MultiQ. Global-MultiQ (Figure 10.8(b)) performs futile DEMOTES both when the cache sizes are small and when the lower cache is larger than the upper cache. In the first case, demoted blocks are discarded before being used, and in the second they are already present in the lower cache. As a result of data redundancy between caches, Double-ARC (Figure 10.8(a)) is highly sensitive to the portion of the aggregate cache that is in the upper level. Karma (Figure 10.8(c)) does not suffer from those problems and outperforms both policies in all cache settings.

We expect Karma to maintain its advantage when the difference in cache
sizes increases. In the Basic and Double policies, the benefit of the smaller cache will become negligible due to data redundancy, whereas in the Global policies the number of futile demote operations will increase. Karma, on the other hand, benefits from any additional cache space, in any level, and maintains exclusive caching by using only the necessary number of demotes.

10.2 $MC^2$

We showed above that Karma achieves the shortest I/O response times for almost all traces and cache sizes. We now show that $MC^2$ maintains this superiority in the presence of multiple clients.

Does $MC^2$ achieve the shortest I/O response times? Our experimental results show that the answer to this question is yes. Figure 10.9 shows the results for a pair of TPCH queries. Following a previous study [112], we chose queries 3 and 7. Query 3 is sequential, scanning 3 tables. PostgreSQL hashes each table and joins them in memory. Subsequent runs of Query 3 result in a looping reference. Query 7 accesses some of the tables sequentially, but most of its accesses use an index tree. The data set of Query 3 is a subset of the data set of Query 7. However, the ranges accessed by Query 3 have the lowest marginal gain for Query 7. Consequently, the first level cache of the client running Query 7 is devoted to non-shared data, and both clients use all the data in the shared cache.

The performance of $MC^2$ (Figure 10.9(c)) is very close to the optimal lower bound, and is the best of all online policies, both for the single client setting and for both clients executing concurrently. Based on the hints, MRU replacement is chosen to manage loop accesses, and LRU is used for index accesses. By combining the right replacement policy with exclusivity, I/O response times decrease linearly with the increase in available cache space.

Double-ARC is the best of the inclusive policies for this experiment. Still, the client running Query 3 suffers from managing its loop blocks in an LRU
Figure 10.8: Average I/O response time of Karma and the best LRU-based policies on query set 1 (the behavior is similar for all traces) when cache levels are of different sizes. *Same:* $|\text{Cache}_2| = |\text{Cache}_1|$. *Big:* $|\text{Cache}_2| = |\text{Cache}_1| \times 2$. *Small:* $|\text{Cache}_2| = |\text{Cache}_1|/2$. Double-ARC is highly sensitive to the difference in size and Global-MultiQ suffers overhead from DEMOTE operations, while Karma outperforms all policies for all cache settings.
Figure 10.9: TPCH queries 3 and 7. Symbiotic sharing occurs for inclusive policies, or for exclusive policies that demote useless blocks. When the performance of a single client (“baseline”) is close to the optimal lower bound, sharing is destructive.
Figure 10.10: TPCH query sets on 5 clients. “ARC” and “MultiQ” both refer to the Double extensions of these policies. Error bars represent one standard deviation. The standard deviation in (a) is plotted only when larger than 2%. Although $MC^2$ exhibits destructive sharing, its I/O response times are significantly shorter than all other online policies, both with a single client (“baseline”) and with multiple ones (“shared”).
based policy, and the client running Query 7 does not enjoy all available cache space due to data redundancy. Demote keeps the caches exclusive for the single client workload, but still manages the loop blocks with LRU. The situation is somewhat improved by sharing for both policies, but their performance is far from that of $MC^2$.

Figure 10.10 shows the results for the TPCH query sets. The large standard deviation in MultiQ and ARC is a result of their high sensitivity to the order of the queries in a set, as seen in Figure 10.5 (see discussion there). $MC^2$ achieves the shortest I/O response times in both single- and multi-client settings. For a single client, this is the result of combining exclusivity with the best replacement policy for each hinted access pattern. For multiple clients, this is the result of exploiting data sharing whenever it is useful, and partitioning the cache to minimize the interference between clients.

Figure 10.11 shows the performance of the different policies on ten TPCC clients. $MC^2$ performs as well as the exclusive policy in the single client setting. Only in the smallest cache sizes (1/16 and 1/8) are demoted blocks not accessed frequently enough to justify the cost of demotion. $MC^2$ is better than all inclusive policies for multiple clients, with I/O response times shorter than the lower bound for a single client, thanks to the prefetching effects of sharing.

The hints used by $MC^2$ in this experiment for the non-uniform accesses were fairly accurate. Random accesses were grouped into a single range, and the skew was computed according to their distribution in the traces. We ran the same experiment using a default (80,20) skew, with similar results. The difference in I/O response time was 0.6% on average for all cache sizes, and was always below 2.1%. This demonstrates that $MC^2$ is able to save the most valuable blocks in the cache even when supplied with approximate hints.

Does $MC^2$ benefit from data sharing more than existing policies? Ironically, the answer is no. Since $MC^2$ performs so well for a single client,
Figure 10.11: Ten clients running TPCC. “ARC” and “MultiQ” both refer to the Double extensions of these policies. Error bars in (b) represent one standard deviation. The standard deviation in (a) was always smaller than 3%. $MC^2$ maintains exclusive caching for a single client, utilizing the entire aggregate cache. When sharing is identified, exclusivity is disabled in the shared cache. Valuable blocks fetched by one clients act as “prefetches” for another, reducing I/O response times below those of the optimal lower bound for demand paging.
it does not leave much room for improvement by sharing. Our results show that the worse performance a policy achieves with a single client, the more it benefits from sharing. This is not surprising; long I/O response times often indicate a poor utilization of the second level cache. This is usually caused by data replication. When several clients share the second level cache, the interleaving of their accesses increases the exclusivity of the cache levels. Such a symbiotic sharing scenario is similar to the behavior of LRU in Figure 5.2. Double-ARC, Double-MultiQ and LRU demonstrate this behavior in the following experiments.

When the pair of TPCH queries in Figure 10.9 execute concurrently, loop blocks are accessed out of order. This greatly improves the performance of the inclusive policies, as shown by the results of Double-ARC in Figure 10.9(a). Figure 10.11(b) shows results for ten TPCC clients executing concurrently. Since the access distribution is similar for all clients, the inclusive policies (LRU, Double-MultiQ and Double-ARC) achieve shorter I/O response times than for a single client. Blocks fetched by one client are also used by other clients, so it is useful to keep them in the shared cache.

In some cases, poor utilization of the second level cache is caused by the wrong choice of replacement policy, even when exclusivity is maintained. Specifically, a bad policy demotes worthless blocks that do not contribute to a reduction in I/O response time. In such cases, when several clients “interfere” with one another’s exclusivity, the blocks saved in the shared cache by one client may be worthless for this client, but useful for another. This symbiotic scenario is demonstrated by Demote in the small cache sizes on TPCH workloads (Figures 10.9(b) and 10.10(c)).

$MC^2$ does benefit most from sharing on the TPCC workload, demonstrating that a constructive sharing scenario can also occur when the cache is utilized well for a single client. When the workload performed by multiple clients is non-uniform yet similar for all clients, every synchronous fetch by one client is a potential “prefetch” for another. A replacement policy which
<table>
<thead>
<tr>
<th>Policy</th>
<th>LRU</th>
<th>Demote</th>
<th>Basic-ARC</th>
<th>Double-ARC</th>
<th>Basic-MultiQ</th>
<th>Double-MultiQ</th>
<th>MC²</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 TPCH query sets</td>
<td>16.1%</td>
<td>7.6%</td>
<td>22.5%</td>
<td>31.5%</td>
<td>15.5%</td>
<td>19.6%</td>
<td></td>
</tr>
<tr>
<td>TPCH Query 3</td>
<td>29.8%</td>
<td>30.7%</td>
<td>31.7%</td>
<td>31.5%</td>
<td>31.5%</td>
<td>25.1%</td>
<td></td>
</tr>
<tr>
<td>10 TPCC clients</td>
<td>37.8%</td>
<td>30.7%</td>
<td>33.9%</td>
<td>32.1%</td>
<td>38%</td>
<td>38.1%</td>
<td>34%</td>
</tr>
</tbody>
</table>

Table 10.2: Maximal reduction in average I/O response time due to sharing. A blank slot means that performance is always better without sharing.

...saves the most “popular” blocks in the cache may achieve shorter wait times than the lower optimal bound for a single client. This symbiotic sharing scenario is demonstrated in Figure 10.11(c) for $\text{MC}^2$.

$\text{MC}^2$ disables exclusive caching for this workload. Valuable random blocks are kept in the shared cache, to allow for the benefit of data sharing. The looping patterns are identified using hints, and shared loop partitions are managed with LRU in the second level cache. This keeps loop blocks in the cache long enough for other clients to use, creating a prefetching effect. As a result, the average I/O response time for multiple clients is shorter than those of the rest of the policies (Figure 10.11(b)). In some cache sizes it is also shorter than the optimal lower bound, similar to the scenario in Figure 5.1.

Clearly, we would like to avoid policies whose benefit from sharing is the result of poor performance for each single client. This appears to be a problem common to all existing policies. In contrast, $\text{MC}^2$ achieves the best performance for a single client, along with symbiotic sharing scenarios, making it the best policy for both single- and multi-client workloads. Table 10.2 summarizes the maximal reduction in I/O response time due to sharing, for all policies.

**Does $\text{MC}^2$ avoid destructive sharing better than existing policies?** In the majority of our experiments, the answer is yes. $\text{MC}^2$ suffers a destructive sharing scenario only when it cannot be avoided, as opposed to the rest of the policies, which simply make the wrong management decisions.
When the cache hierarchy is fully and efficiently utilized for a single client by a strictly *exclusive* policy, sharing is usually destructive. This behavior is observed in two cases. The first is when the aggregate cache is big enough to hold the entire data set or most of it. The I/O response times for an exclusive policy approach those of the optimal lower bound, regardless of its replacement decisions. Sharing in this case is always destructive, since it interferes with exclusivity. Demote exhibits such behavior in the large cache sizes for the TPCH traces, in figures 10.9(b) and 10.10(c).

Sharing and exclusivity also lead to a destructive scenario when exclusivity is combined with the best replacement policy for the current access pattern. In such cases, clients “steal” valuable blocks from one another, as READ blocks are automatically discarded. Demote demonstrates this scenario on the TPCC workload in Figure 10.11(c), as it does in the example in Figure 5.2.

Exclusivity aside, destructive sharing can also result from competition between clients. When all the clients are able to fully utilize the shared cache, they harm one another’s performance because each one takes up space required by the others. This is the case of $MC^2$ in the TPCH query sets in Figure 10.10(c). The clients execute the queries in different order, and their working sets do not completely overlap. Sharing is destructive due to increased load on the shared cache.

Clearly, we would like to avoid exclusive caching whenever it degrades performance. The resulting destructive sharing scenario would indicate poor management, as it does in the case of Demote. In contrast, a destructive scenario which results purely from increased load on the shared resources cannot be avoided, and does not indicate a weakness of the management policy. This is the case of $MC^2$, which achieves the best performance both for a single and for multiple clients. Table 10.3 summarizes the maximal increase in I/O response time due to sharing experienced by all policies.

**Does $MC^2$ maintain a high degree of fairness between clients?**
Our experimental results show that the answer to this question is yes. $MC^2$ allocates more cache space to clients which appear to use the shared cache more frequently and efficiently than others. Apart from achieving short I/O response times, this allocation also maintains a high degree of fairness.

Recall that perfect fairness is achieved when $Fairness = 1$. Table 10.4 shows the average fairness for each policy and workload. Fairness values are averaged for all cache sizes. We also show the worst fairness achieved by each policy. LRU usually maintains the highest degree of fairness in the shared cache. Demote, ARC, and MultiQ are based on LRU, and therefore achieve similar results. Our experiments show that $MC^2$ is able to maintain fairness close to that of LRU, even though it does not always use LRU (or LRU-based policies) to manage the shared cache, and even though it is primarily designed to reduce I/O response times in the system.

We conclude that $MC^2$ is the best policy for managing a hierarchy of mult-
multiple cache levels used by a single client as well as by multiple ones. It achieves the shortest I/O response times in all our experiments. By combining exclusive caching and application hints, it guarantees the best performance for a single client workload. The same principles are used for each client in a multi-client setting, by the local allocation and replacement scheme. The application hints are further leveraged to choose between inclusive and exclusive sharing, optimizing the performance of the system in the presence of multiple clients. Thanks to the global allocation scheme, the most valuable blocks in the system are saved in the shared cache, while a high degree of fairness between competing clients is maintained.

10.3 Cooperative approaches

Clients that serve peer requests invest work, and possibly valuable cache space, hoping to avoid expensive disk or Internet accesses. We examine four representative test cases to answer our question: *is cooperative caching really helpful?*

We use three state-of-the-art caching approaches that have been optimized for multiple clients to define the baseline performance achievable without cooperation. The first is LRU, the most commonly used cache management scheme. The second is ARC with Promote, that represents a non-informed replacement scheme optimized for multiple cache levels and multiple clients. Finally, $MC^2$ represents a management scheme optimized for multiple cache levels and multiple clients that uses hints from the application.

10.3.1 TPCH query sets workload

Figure 10.12 shows the average I/O delay incurred by the different policies on the TPCH sets workload. All clients run the same set of TPCH queries in different order, resulting in a dynamic mix of workloads. Clients provide new hints whenever their workload and corresponding priorities change. Since the
Figure 10.12: Performance of cooperative approaches on the TPCH query sets

changes in workload are small and there is considerable data sharing between
the clients, cooperation always improves performance. However, the different
cooperative 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-Hints, the best ap-
proach in this case, reduces the average I/O delay by as much as 32% for 2
clients and a cache size of $\frac{1}{8}$. When hints are not available, 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 for sev-
eral reasons. 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 indirect incentive mechanism in C-hints successfully addresses
this issue. C-ARC is also limited by its lookup mechanism – clients are
Figure 10.13: Schematic representation of the effect of client selfishness on performance with different workload characteristics (a) uniform distribution, medium correlation (b) heavy tail distribution, medium correlation (c) non-negligible SERVE and high access skew (d) stable, uniform distribution.

unable to locate blocks in peer caches. Additionally, since peers are not obligated to store served blocks for a long period of time, clients duplicate their frequently accessed blocks and don’t rely on their peers to supply them. The configurations constructed by the server in C-Hints address this issue.

In contrast, clients in C-Hints 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 $\frac{1}{10}$ and 20 clients, the benefit of C-Hints, 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 incentive schemes are addressed by Algorithmic Mechanism Design [81].

Figure 10.13(a) provides a schematic representation of the negative effect client selfishness has on cooperation with large caches and uniform, strongly correlated workloads.
Figure 10.14: Performance of cooperative approaches on 50 clients viewing videos. The error bars (b) depict the standard deviation.

### 10.3.2 Video workload

Figure 10.14(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-Hints 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%, 14% and 44% over the hit rates of C-P2P, C-DHT, and LRU, respectively\(^1\).

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 GBs, the popular videos are already stored in the cumulative client cache, but

---

\(^1\)ARC, and subsequently, C-ARC, are specifically optimized for storage I/O workloads, explaining their inferior performance on these traces. We present their results for completeness, but omit them from the discussion.
unpopular videos fail to “enter” it: an unpopular video is accessed by few clients, and if none of them is responsible for it, it will not be stored at any cache, and the clients will all request this video from the server.

Figure 10.14(b) shows the average upload bandwidth consumed by the individual peers. The error bars, depicting the standard deviation, show the load distribution between the clients. The per-peer credit balances in C-P2P ensure even distribution, but also limit cooperation opportunities. C-Hints is more susceptible to imbalance, but, surprisingly, achieves better load balancing than the hash keys in C-DHT. Clients using C-Hints selfishly replicate extremely popular videos, while in C-DHT those videos turn arbitrary peers into “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-hints. Although the hints successfully identify the most popular movies, the workload changes faster than the hints are updated. Videos which are most popular when the hints are generated 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 10.13(b) schematically shows that cooperation is always helpful when its cost is negligible, even with long tail distributions and medium data correlation, and that selfish considerations improve performance in this case.

### 10.3.3 TPCC workload

Figure 10.15(a) shows the reduction in I/O response time achieved by cooperation between 20 clients running the TPCC 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
Figure 10.15: Performance of cooperative approaches on 20 clients running the TPCC workload

...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 set without accessing the disk. However, this measure does not take into account the work clients invest in serving peer requests.

The cost of serving peers, the service delay, is depicted in Figure 10.15(b). Comparing Figure 10.15(b) to Figure 10.15(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 infrequently accessed blocks, whose utility does not mask the cost of serving peers.

The average I/O delay takes this cost into account. Figure 10.15(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 10.13(c) schematically 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 are expected to refuse to cooperate under such circumstances.

Specifically, the hints used by C-Hints were supposed to help it detect that cooperation is not beneficial with TPCC (Figure 10.15(c)). However, they were not accurate enough: 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 MC², clients selectively store only the more frequently accessed blocks within such ranges. However, with C-Hints 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.

Without hints, selfishness may help minimize the negative effect of cooperation in this case. Clients in C-ARC maintain control of their cache content, thus suffering the smallest penalty from cooperation. This is especially evident when C-ARC is compared to ARC, which employs the same cache management without cooperation.

### 10.3.4 TPCH query combinations workload

In our last workload, each client runs one query repeatedly, with varying parameters, as explained in Chapter 9.3. In such a scenario, the hints provided by the clients are highly accurate, and the stability of the working sets pro-
Figure 10.16: Performance of cooperative approaches on TPCH queries 3, 10, and 18

provides good opportunity for cooperation. Figure 10.16 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-Hints achieve the best performance improvements, thanks to their ability to achieve exclusive caching. In C-DHT exclusivity is the result of dividing the hash key space statically between clients. In C-Hints 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 the lack of central management, causes excessive data replication.

Figure 10.13(d) depicts the trend we observed in this workload; the most selfish and the most altruistic approaches achieve the best results. 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 was the same. The results for the rest of our query combinations are summarized in Figure 10.17.

10.3.5 Energy considerations

The tradeoff between power consumption and performance is a key factor in processor design, resulting in a wide range of processors optimized for different uses. Server class processors operate at high clock frequencies and consume large amounts of energy. Mobile processors operate at lower frequencies and lower energy consumption rates. We explored the effects of different power specifications on different cooperation policies. For an example server class processor, we use Intel’s high end Xeon® E7 [8], and for a high performance mobile processor we chose Intel® Core™ i7-900 [7]. Intel’s Centrino® [4] represents a mobile processor optimized for energy savings. The specifications of these cores appear in Table 10.5.
Table 10.5: Power specifications: Core maximal clock frequency (GHz), power consumption (watts) and corresponding processor C state, and energy required (µJ)

<table>
<thead>
<tr>
<th></th>
<th>Centrino</th>
<th>i7-900</th>
<th>Xeon E7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2.33</td>
<td>3.20</td>
<td>3.20</td>
</tr>
<tr>
<td>( P_{busy} )</td>
<td>34 (C_0)</td>
<td>55 (C_0)</td>
<td>130 (C_0)</td>
</tr>
<tr>
<td>( P_{net} )</td>
<td>13 (C_2)</td>
<td>20 (C_{1E})</td>
<td>56 (C_{1E})</td>
</tr>
<tr>
<td>( P_{idle} )</td>
<td>1.2 (C_4)</td>
<td>14.5 (C_3)</td>
<td>36 (C_3)</td>
</tr>
<tr>
<td>( E_{serve} + E_{net} )</td>
<td>0.007</td>
<td>0.011</td>
<td>0.029</td>
</tr>
<tr>
<td>( E_{disk}(SATA) )</td>
<td><strong>0.015</strong></td>
<td><strong>0.149</strong></td>
<td><strong>0.371</strong></td>
</tr>
<tr>
<td>( E_{disk}(SAS) )</td>
<td>0.009</td>
<td><strong>0.077</strong></td>
<td><strong>0.191</strong></td>
</tr>
<tr>
<td>( E_{disk}(SSD) )</td>
<td>0.003</td>
<td>0.008</td>
<td>0.021</td>
</tr>
<tr>
<td>( E_{disk}(DRAM) )</td>
<td>0.004</td>
<td>0.006</td>
<td>0.016</td>
</tr>
</tbody>
</table>

We expected a reduction in the energy consumed by cooperating clients whenever the energy invested in cooperation (serving a peer request, and later fetching a block from a peer) is lower than that consumed by a disk access. We observed this result with the setups highlighted in Table 10.5. In those setups, the reduction in energy savings of each policy was proportional to the reduction in average I/O delay.

In the Centrino processor and SAS disk setup, cooperation almost always increased energy consumption, and the potential for reduction was not realized. Figure 10.18(a) shows the energy consumption of the different policies with 20 clients running the TPCH query sets. With most cooperative policies, the amount of peer requests served exceeded the amount of disk accesses avoided – the space freed in the cache by requesting blocks from peers was populated with less popular blocks. Thus, the energy invested in service exceeded the energy saved by avoiding disk accesses. The gap was larger in the altruistic approaches, especially C-DHT and C-P2P.

In addition, the extra energy consumed by cooperation can be higher than the estimate, due to queuing and interleaving effects. If an arriving peer request forces a client to “wake up,” there is not always enough time to return to \( P_{idle} \) after the request is served. Thus, the utility calculations are not accurate enough, even though the hints are accurate, causing clients...
in C-Hints to spend more energy than those in $MC^2$ (but less than all other non-cooperative policies. In general, the results for this setup resembled the dashed line in Figure 10.13(c), where selfish clients do a better job at avoiding cooperation when the benefits are small.

**Combining Energy and Performance.** To evaluate the two conflicting objectives together, we use the energy delay product ($ET$), described in Chapter 7.4. Figure 10.18(b) shows $ET$ for 20 clients running the TPCH query sets. The reduction in $ET$ is similar to the reduction in I/O delay, for all policies. For the setups and workloads where cooperation reduced the I/O delay, it reduced $ET$ proportionally. This demonstrates that the increase in energy was small compared to the decrease in I/O response time.

### 10.3.6 Storage access costs

The benefit from cooperation between clients, like that of other techniques that incur space [44] and transfer [114] 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 ($T_{\text{disk}} = 5\,\text{ms}$), we added here a SATA disk drive, representing long access delays ($T_{\text{disk}} = 10\,\text{ms}$), an SSD drive, representing short delays ($T_{\text{disk}} = 260\,\mu\text{s}$), and a controller with DRAM storage, representing high-end fast storage access ($T_{\text{disk}} = 120\,\mu\text{s}$) [9]. Figure 10.19 shows the performance of the best two cooperative approaches on these setups.

We expected the benefit from cooperation to increase with $T_{\text{disk}}$. Indeed, with the SATA drive all policies benefit more from cooperation than they do with SAS. Cooperation also proves beneficial with small cache sizes on the TPCC workload, where it enables clients to cache larger portions of the most frequently accessed blocks.

We expected some benefit from cooperation with faster storage as well, as long as $T_{\text{serve}} < T_{\text{disk}}$. In practice, all cooperative policies except C-Hints increased the average I/O delay, for all our workloads, both with SSD and with DRAM storage. Cooperation improves exclusivity and frees up cache buffers, but these buffers are not necessarily populated with frequently accessed blocks. Thus, disk accesses saved are fewer than the number of peer
REQUEST and serve operations, canceling any benefit from cooperation. C-Hints suffers less as a result of decreases in $T_{disk}$ because clients know the access frequencies of their blocks and selfishly avoid cooperation in those setups.

10.3.7 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-Hints, 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. While upload bandwidth was a bottleneck for C-Hints, the performance of C-DHT and C-P2P was determined by the hit rate in the client caches.

We reached a similar conclusion when we examined the amount of client requests that were rejected due to unavailable upload bandwidth by their peers. With 1 Mbps upload bandwidth, the rejected requests consisted less than 0.3% of all requests with all policies and cache sizes, except for C-hints. There, with caches of 4 GB or more, clients using C-Hints rejected up to 1.7% of peer requests. With 2 Mbps of available bandwidth the number of rejects with all policies was zero.

10.3.8 Lookup mechanisms

Lookup mechanisms have been suggested in previous studies [37, 39, 40, 95], and were not the focus of our evaluation. However, we identified cases in which insufficient lookup capabilities limit cooperation. While C-DHT and C-Hints have built-in mechanisms that do not incur additional overhead, C-ARC and C-P2P are limited by $max_p$, the number of peers a client is allowed
To query for each block before requesting it from the server.

To determine the effect $max_p$ has on the performance of C-ARC and C-P2P, we vary it from 2 to the number of clients. In the video traces, we measured the reduction in Internet accesses. The improvement was small in C-P2P (up to 3%) because the server served as a tracker, and clients stored peer information in their ghost cache. C-ARC improved slightly more, by up to 7%. The improvement was more substantial with the TPCH query sets. There, when $max_p$ increased the performance improved continuously, decreasing average I/O delay of C-ARC and C-P2P by as much as 63% and 76%, respectively, when $max_p$ was equal to the number of peers, eliminating all block replications.

Querying all the peers in the system is clearly infeasible in a practical implementation. Instead, policies that lack a built-in mechanism for block lookup should be augmented by some external mechanism to enable full utilization of the cumulative client caches.

10.3.9 Account limits

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-hints, each client maintains an aggregate credit balance for all peers, that restricts the cooperation transactions added to the configuration.

In order to evaluate the effect account limits have on performance, we varied the aggregate limit from 100K to 800K in the TPCH and TPCC workloads, and from 25K to 200K in the video workload. We examined the change in performance, as well as the change in the load balance between clients, which we measured as the standard deviation in service delay. 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.
Table 10.6: Average and maximal changes in load balance (standard deviation in service delay) and performance (average I/O delay and web accesses) when increasing account limits.

The results, summarized in Table 10.6, were somewhat surprising. Increasing account limits had no effect on the performance of C-ARC. This demonstrates that that cache hit ratio and lookup limitations were the bottlenecks of cooperation in this approach. In C-P2P, the effect was minor and matched our expectations. C-Hints was more affected by the increase in balance limits, but much less than we expected.

The performance of C-Hints on the TPCH query set workload exposed a surprising phenomenon. With small caches, increasing the balance limits decreased the average I/O delay by as much as 35%, as it provided for more cooperative configurations. With large caches, however, the average I/O delay increased by as much as 64%, because clients agreed to cooperate on less valuable blocks, whose utility did not mask the additional cost of serving peers. Load balance improved in all cache sizes, because it was easier for the server to find configurations that “fixed” the differences between clients.

In conclusion, although account limits seem like a critical design parameter, our analysis shows that their precise value has little effect on the performance of most cooperative approaches. However, account balances have a fundamental role as incentives for selfish clients to cooperate, and their
non-zero limits allow clients to initiate their cooperative transactions. At the same time, note that high account limits increase the vulnerability of the system to free riders[36] – peers that maliciously receive service without contributing their resources.

10.3.10 Hint generation

Our results showed that hints and their quality can have a significant effect on utility calculations, and hence on the performance of hint based approaches. We discussed the limitations of hint generation from the database explain mechanism in Section 10.3.3.

For the video traces, we performed an offline analysis on the prediction accuracy of hints based on statistics gathering in the server with three different time frames. In each test, we gathered statistics on a training data set, and ordered the videos in this set in decreasing order of popularity. We used the same ordering for the videos in the test data set. Let a prefix of size $x$ be the group of $x\%$ most popular videos in a data set. An intersection of size $y$ means that $y\%$ of the videos that appear in the test set prefix, appeared in the training set prefix.

The results of this analysis are presented in Table 10.7. They show that long training sets and long prefixes tend to result in overfitting, since they include too many moderately popular videos. While the most popular videos stay popular for longer periods, the moderately popular videos may not be as popular in the following time frame. We expect prediction accuracy to improve with statistics collected in larger, more active ISPs.

10.3.11 Configuration search

The utility of cooperating clients in C-Hints depends on the quality of the cooperative configuration constructed by the server. As noted above, finding an optimal configuration is infeasible in an online system. A commonly used
technique for searching near-optimal solutions in such situations is by use of stochastic search [54]. We applied this technique to the configuration constructed greedily by the server to measure the benefit that may be gained from configurations closer to the optimal one.

The *stochastic search* is composed of a series of steps in which random changes are made in the configuration. In each step, a random client is picked and one of the block ranges it saves is replaced with another, randomly picked, range. Several peers are picked randomly to receive this range from the client. The resulting configuration must be *valid*, i.e., the utility of all the clients must be greater than their utility in the base configuration. Non-valid configurations are discarded and the search continues from the previous configuration. Otherwise, if the global utility of the new configuration is greater than the global utility of the old one, the step is an *improvement*, the step is applied, and the search continues from the new configuration.

Valid steps which are not improvements are also applied with a low probability (*noise*), to enable escape from local minima. We experimented with two implementations of the search. In the first, the noise is constant throughout the search. In the second, the noise is adjusted with Simulated Annealing [64] to approach zero by the end of the search. Our measurements show that several thousands of steps can be performed in a few milliseconds, the time required for one disk access.

We experimented with seven values of noise, from 0 to 10%. In all workloads, increasing the noise resulted in degraded performance. Moreover, even though the utility of the configurations found by the search phase was larger

<table>
<thead>
<tr>
<th>Training set</th>
<th>Test set</th>
<th>5%</th>
<th>10%</th>
<th>15%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Day</td>
<td>Next day</td>
<td>29</td>
<td>26</td>
<td>21</td>
</tr>
<tr>
<td>Week</td>
<td>Next week</td>
<td>36</td>
<td>36</td>
<td>31</td>
</tr>
<tr>
<td>Day</td>
<td>Next week</td>
<td>49</td>
<td>44</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 10.7: Prediction accuracy presented as the average size of the intersection (%) for three training and test set size combinations and three prefix sizes.
than the utility of the greedy configuration, the performance of C-Hints did not improve. In the video workload using the stochastic search harmed the performance of C-Hints, increasing the portion of videos accessed from the Internet almost by 1%. The reason is insufficient accuracy of the hints. The stochastic search results in configurations with more cooperation transactions than the greedy construction. However, the marginal gain from these transactions is smaller, and is more susceptible to even minor inaccuracies in the hints.

10.3.12 Is cooperative caching really helpful?

For the majority of our workloads and setups, the answer is yes, cooperative caching is helpful. The effects that client selfishness has on the performance of a cooperative cache are depicted in Figure 10.20. With large caches or strong data correlation, altruistic behavior is better than selfishness. With small caches and weak data correlation, selfish clients perform better than altruistic ones. With stable workloads and medium correlation managed protocols where the responsibility of each peer is well defined is better than the non managed ones. Finally, with a non-negligible cost of serve and high access skew cooperation may actually degrade performance, and independent caching is preferable.

Our analysis of the system and design parameters revealed that performance depends mostly on the system and workload parameters. The behavior with the different approaches is consistent when examining different cost functions and costs. When varying the storage access times or when considering energy consumption or the energy delay product as the objective functions, the benefit from cooperation always increases with the increase in the cost of a disk access compared to the costs of send and receive. The effects of selfishness depend on the workload characteristics regardless of the objective function.

When considering the algorithm design parameters, our analysis reveals
Figure 10.20: Schematic representation of the effect of client selfishness on performance with different workload characteristics (a) Large caches, strong data correlation (b) Small caches, weak data correlation (c) Non-negligible cost of serve and high access skew (d) Stable workload and medium correlation.

that account balance limits have little effect on the performance of the various algorithms. However, an efficient lookup mechanism, either built in or external to the cooperative protocol, is imperative.

When explicit application hints are used for utility calculations, their quality and level of detail are a dominant factor in several workloads and setups, as they affect the quality of the configurations constructed by the central component. In a production system, we recommend that a cooperative caching scheme will be augmented with some means of sampling the true utility of the caches, and whether the supplied hints are accurate enough for utility estimations.
Chapter 11

Discussion

Three aspects of our storage model and algorithm require further discussion.

11.1 Write traffic

Our model assumes that write traffic is managed in a separate write cache. In the upper level cache, the client must determine what portion of its cache to devote to dirty pages. The AWOL framework [21] provides an effective way to do this dynamically, working with existing algorithms for managing the read portion of the cache. It could be combined with $MC^2$ in the following manner. When the workload changes, AWOL will adjust the memory available for read caching, while $MC^2$ will repartition this memory according to the new hints. When a dirty page is destaged, it logically moves from the write cache to the read cache. There, it should either be stored in the appropriate partition, or become an immediate candidate for eviction, according to its marginal gain.

In the low level cache, non-volatile memory is typically used as a write cache to enable “fast writes.” This write cache can be managed efficiently using WOW [47]. To maintain the advantages of $MC^2$ in shared write caches, an allocation scheme similar to our LRU-(partition,client) may be used.
When a non-volatile write cache is used in the low level, dirty blocks can remain cached for a considerable amount of time. During that time, the upper level may still cache these blocks for future access. To maintain exclusivity and ensure that a block is cached only once, $MC^2$ must be adjusted to explicitly handle write operations. One way to do this is by treating writes as demotes, discarding a block from the upper level cache when it is written. When the write completes in the disk and the block is discarded from its cache, it may be sent again to the upper level cache. Special care must be taken to avoid excessive network transfer in such scenarios.

A block in the non-volatile write cache can also serve read requests that miss in the upper level. A further enhancement would be to consider the presence of blocks in the low level write cache in the management decisions of the read cache in the same level. However, such coordination between separate read and write caches is outside the scope of this work.

In addition to the replacement decisions, 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. On the contrary, exclusive client caches may suffer less from frequent invalidations. The precise effect of write traffic on our cooperative approaches is out of this paper’s scope. However, some of them are more suited to handle write traffic than others.

A built-in lookup mechanism which ensure exclusive caching can easily be augmented with a cache coherency protocol. Thus, in C-DHT and C-Hints, a client responsible for a range or a hash key can also be in charge of master copies of the corresponding blocks. Replication in C-P2P occurs only through accesses to the shared cache, and it does not introduce coherency issues beyond those of a shared server cache. C-ARC, however, presents an additional challenge, as clients are allowed to replicate served blocks according to their needs.
11.2 The quality of hints

While $MC^2$ is designed for applications which supply prior information about their accesses, it can cope well with several common cases of applications that do not supply hints.

Consider a system with a single client running an application that does not supply hints. Its blocks will be managed as a single range, without partitioning the cache. The cache levels will remain exclusive, with the performance of $MC^2$ similar to that of Demote. To improve performance, such “unhinted partitions” can be managed by any other non-informed policy, such as ARC and its variants.

A shared cache with multiple clients that do not supply hints will manage the blocks of each client in a separate partition. On a cache miss, the global allocation scheme will evict a block from the LRU partition, and each client can manage its blocks with the non-informed policy of its choice. Without hints that classify blocks into ranges, sharing cannot be detected in advance. However, a client can still access a block stored in a partition of another client. The allocation scheme of $MC^2$ can be adjusted to keep this block in the partition of the client which is likely to store it for a longer period of time. This can be determined according to the allocation scheme of each client, and the respective position of the block in its queue.

In certain scenarios, the performance of a client using $MC^2$ can be worse than if it were using LRU. The first such scenario occurs when a client supplies the wrong hints. Hinting the wrong access pattern can cause $MC^2$ to choose the wrong replacement policy for a partition. A substantial miscalculation of access frequencies can cause it to allocate space for partitions with low marginal gain. Both “mistakes” will result in a cache storing blocks with a small probability of being reaccessed. This was the case, for example, in the video traces, where hints generated at the server were used by individual clients (Figure 10.14). This problem can be solved by simple monitoring. A client with low confidence in the quality of its hints may use a ghost cache.
to determine whether its performance is worse than what it would be using LRU.

The second scenario occurs in a cache shared by multiple clients, when some provide hints and some do not. A client that does not provide hints manages all its blocks in a single partition. All its accesses will be to this one partition, which will most likely remain at the head of the LRU (partition,client) stack. Thus, even if a large portion of this partition holds blocks of low marginal gain, it will not be chosen as victim and this client will enjoy more than its fair share of the cache. Simple monitoring of the hit rate of each partition can solve this problem as well. MC\(^2\) can be adjusted to prevent unhinted partitions from claiming blocks that belong to partitions with a higher hit rate.

When wrong or inaccurate hints are used for utility calculation in cooperative caching, we observed two possible outcomes. In the video traces (Chapter 10.3.2), the inaccurate hints caused MC\(^2\) to perform worse than LRU, but with C-Hints performance improved dramatically, for two reasons. First, by cooperating clients had access to a larger cumulative cache. Thus, the space allocated to less valuable blocks was of less importance. Second, blocks that are less valuable for some clients are more valuable to others, so storing them results in improved overall utility.

With the TPCC traces (Chapter 10.3.3), the roles were reversed. Non-cooperative clients using MC\(^2\) selectively store only the more frequently accessed blocks within such ranges. However, cooperating clients using C-Hints agree to be responsible for a range, and are obligated to store its low priority blocks as well, at the expense of their own performance.

The performance of clients in the system can also be harmed by malicious clients sharing their cache. MC\(^2\), in its current form, is vulnerable to attacks by malicious clients, but not more than the rest of the policies discussed here. A client that wishes to claim more than its fair share of the cache can issue extra I/O requests. On a cache miss, it will claim additional blocks, and on a
cache hit, it will keep its partitions at the head of the LRU (partition,client) stack. A denial of service attack can be launched applying the same method more aggressively. The attacker can make sure it accesses enough blocks with a high enough frequency, until no other client can keep its blocks in the cache. Note that this simple attack does not rely on either of the special mechanisms of $MC^2$: hints or explicit partitioning. Thus, any policy that partitions the cache dynamically, including LRU, is vulnerable to the same attack. To avoid it, explicit security mechanisms should be employed.

In the cooperative model, the vulnerability of clients to attacks by malicious peers increases. Such peer may refuse to share the blocks they are assigned, or even serve incorrect data. Security mechanisms such as those employed in Shark [11], should be added to the cooperative protocol to prevent such attacks.

\section{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 [29].

While the replacement decisions alone are orthogonal to the cooperation protocols, all cooperative approaches discussed in this study will require some kind of adjustment to accessing multiple servers.

In C-P2P, each server should track the clients accessing its data blocks, performing the same functionality as the single server in our model. In C-DHT, the data blocks from each storage server should be distributed among the clients accessing this server. Consequently, clients should have some means of identifying peers that access the same servers they do. Clients in C-ARC will need similar means of detection in order to send REQUESTS only
to relevant peers.

In C-Hints, clients are expected to adhere to the configuration constructed by the server. Therefore, they cannot adjust their partition sizes independently. Instead, we suggest that clients will adjust their partitions only at the end of predetermined time frames. At the beginning of each time frame, the clients will inform the relevant servers of the partition sizes available for them, and each server will construct its configuration accordingly. The configurations of the different servers do not interfere with one another – they guarantee utility in independent cache partitions. Clients may also use these utilities to determine their partitioning in subsequent time frames.

### 11.4 Security

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, in Shark [11], encryption and opaque tokens are used to enable cooperation between mutually distrustful clients. CloudViews [43] suggests unforgeable tokens to ensure privacy in data sharing scenarios, replacing the traditional access control lists. Finally, the credit balances used in some of our approaches can also prevent free riding [36].
Chapter 12

Related Work

We discussed several existing cache replacement policies in Chapter 9. Here we elaborate on additional aspects of related work.

12.1 Multiple cache levels

Policies oblivious to sharing. Many replacement policies are oblivious to the cache level they manage and to the number of clients accessing the cache. Their performance may be implicitly affected by the interleaving of access patterns, and some are specifically designed for a certain cache level, but no effort is made to distinguish between different sources of requests. Examples of such policies are LRU, MRU, ARC [75], CAR [18], MultiQ [121] and many others [58, 60, 83]. Some policies which attempt to optimize certain aspects of disk scheduling, such as WOW [47] and STEP [71], also fall into this category.

Exclusive policies. Exclusive caching is difficult to achieve and is not always efficient in a system with multiple clients. Several policies designed for global management of the cache hierarchy do not address the issues discussed in Chapter 5. In one example [69], the application attaches a “write hint” to each WRITE operation, choosing one of a few reasons for perform-
ing the write. The storage cache uses these hints to “guess” which blocks were evicted from the upper level cache and which are cached for further use. In X-Ray [100], the information on the content of the upper level cache is obtained using gray-box techniques, and derived from operations such as file-node and write log updates. The extensive survey in [30] examines different approaches to multi-level cache management, but focuses on the case of a single client. Similarly, the global algorithm proposed in [121] assumes a single client. There, blocks evicted by the client are reloaded (prefetched) into the second level cache from the disk. Multiple clients are mentioned and there are several ideas for extending the algorithm, but it is constructed only for one client. In heterogeneous caching [12], some degree of exclusivity is achieved by managing each cache level with a different policy, chosen dynamically in accordance with changing workloads. The experiments were conducted for a single cache in the first level, but this implicit method might also be useful for multiple clients.

**Multiple clients.** A different group of multi-level cache policies specifically addresses multiple clients. The first such policy was Demote [114], when the notion of exclusive caching was first introduced. The single client Demote functions well for two corner cases of data sharing, disjoint and conjoint. For real-life sharing scenarios, which usually fall somewhere in between these two cases, ghost caches are used to estimate the value of demoted blocks compared to that of read blocks. This estimate is used to choose the depth in the LRU stack in which blocks of each type are inserted.

In ULC [59], the client cache is responsible for the content of all cache levels underneath. The level of cache in which blocks should be kept is decided by their expected locality of reference. A “level tag” is attached to each I/O operation, stating in which cache level the block should be stored. Global LRU allocation is employed for multiple clients. This method works well when the entire cache is managed by LRU, as opposed to the example in Figure 6.1. The issue of shared data is not addressed.
MC² uses read-save to avoid unnecessary demotes. As with the average read access time in ULC [59], our storage model enables specific calculations of the demote operations that actually occurred, instead of estimating them in the cost of each read, as in the original evaluation [114]. The use of read-save reflects a non-centralized operation, as opposed to the level tags. Our results in Chapter 10 (figures 10.3, 10.5, and 10.6) show that exclusive caching is not enough to guarantee short I/O response times. MC² is able to make informed replacement decisions to achieve better results. MC² does not assume, or detect, the degree of sharing. Instead, shared blocks are identified by client hints, which also disclose the combination of access patterns. Exclusive caching is maintained only when it is known to be useful.

Special attention must be given to promote [44], which is an alternative technique for achieving exclusive caching. It lets a lower level cache take a (probabilistic) decision whether to own a block and store it locally, or to promote it to the requesting upper level and evict it. This is, in a sense, similar to our use of the read-save operation, in which the upper level decides not to store a block. Both approaches reduce the cost incurred by numerous demotes.

Promote could be used in MC² as an alternative mechanism for achieving exclusivity in the management of skewed ranges, when sub-ranges are placed in different levels. With demote, blocks belonging to such ranges are always placed at the highest level and are gradually demoted. This type of management emulates Global-LRU for this range. A promote variant of Global-LRU could achieve better performance, as we describe below. This additional improvement of MC² does not consist of a structural change in the algorithm. Rather, it is another possible replacement policy to be used for a specific access pattern.

MC² uses the demote operation in two other scenarios as a mechanism for achieving exclusivity, where it cannot be replaced by promote. The first such scenario is when repartitioning occurs, and demote is used for
moving blocks from the first level cache to a new partition in a lower level. This is done when the blocks’ new marginal gain is lower than the original one. Using PROTOTYPE instead of DEMOTE would cause these blocks to be discarded and fetched again from the disk, which is clearly less efficient.

In the second scenario, a range with a uniform access pattern is split between cache levels, and DEMOTE ensures that all the range’s blocks stay cached in one level or the other. PROMOTE, on the other hand, would cause blocks to be discarded from the first level cache. These blocks would eventually have to be fetched again from the disk, since they would also be discarded from the lower level cache (at the time of promotion).

PROMOTE is evaluated by implementing Global-LRU and Global-ARC for a single client using PROMOTE instead of DEMOTE. The analysis includes three aspects of the performance of the PROMOTE and DEMOTE variants: inter-cache traffic, hit rate in the first level cache, and response time. PROMOTE is 2x more efficient than DEMOTE in inter-cache bandwidth usage. The PROMOTE variants also achieve up to 38% more hits in the first level cache. Response time is measured in two settings. One is with unlimited bandwidth, where the response times of the PROMOTE variants are up to 5% better, 1.5% better on average. The other setting consists of limited bandwidth and possible network congestion. The PROMOTE variants outperform the DEMOTE ones by up to 47% in this setting.

As described in Chapter 9.1, the cost of DEMOTE in our experiments is 1/50 the cost of a disk access, while it is only 1/5 in the PROMOTE study. As a result, although we model the maximal delay caused by a DEMOTE to the READ request that triggered it, our results should be compared to the unlimited bandwidth setting in the PROMOTE study. Accordingly, we expect that the PROMOTE variants will achieve only a minor improvement over the DEMOTE variants in our experimental setting. Examining our results, it is clear that subtracting 5% of the response time of Global-LRU or Global-MultiQ will not bring them much closer to the improvements of $MC^2$. This
is because the greatest advantage of $MC^2$ for a single client comes from choosing the replacement policy most suited for each access pattern.

**PROMOTE** can be used in systems with multiple clients by maintaining and adjusting the $probPromote$ variable (the probability of promoting a block to the upper level cache) for each client separately. The experimental evaluation in [44] did not include workloads with data sharing. We expect that when exclusivity causes destructive sharing, the dynamic adjustment of $probPromote$ will cause it to approach zero, with few blocks being promoted to upper levels, effectively disabling exclusivity. While such behavior in PROMOTE depends on monitoring and on the stability of the workload, $MC^2$ only needs a simple indication that a range of blocks is shared by several clients.

### 12.2 Application based caching

**Detection.** Detection based policies use history information for each block in the cache (and sometimes for blocks already evicted from the cache) to try to “guess” the access pattern of the application. This may help identify the best candidate block for eviction. DEAR [32], AFC [33], and UBM [63], which are designed for a file system cache, all collect data about the file the block belongs to or the application requesting it, and derive access patterns (sequential, looping, etc.).

In PCC [48], the I/O access patterns are correlated with the program counter of the call instruction that triggers the I/O requests, enabling multiple patterns in the same file to be differentiated if they are invoked by different instructions. Each pattern is allocated a partition in the cache, whose size is adjusted dynamically until the characterization of patterns stabilizes.

MRC-MM [120] monitors accesses to virtual memory pages in order to track the page-miss ratio curve (MRC) of applications. The MRC in each “epoch” is then used to calculate the marginal gain of all processes. Memory utilization is maximized by allocating larger memory portions to processes...
with higher marginal gain. BORG [24] analyzes an I/O trace collected in the file system in order to copy working-set data blocks into a disk-based cache, in their relative access sequence.

In DB2 [106], several memory consumers, such as compiled statement cache and buffer pool, are balanced. There is no sharing between consumers, but each one represents different processes accessing them. Ghost caches are used to compute the cost-benefit for each consumer, and balance their memory allocation. Argon [112] uses a similar approach for a storage server. It combines partitioning with scheduling to provide insulation, where a service allocated 1/nth of a server gets nearly 1/nth of the throughput it would get alone. The services are assumed to be completely independent, so data sharing is not considered. A recent scheme for multi-resource allocation [102] creates a performance model for each application, and performs experimental sampling, in order to divide the storage bandwidth and cache between them.

Since $MC^2$ is provided with the characterization of block ranges, it does not incur the additional overhead of gathering and processing access statistics. The access pattern and access frequency for each range are known and on-the-fly adjustment of partition sizes is avoided. $MC^2$ adjusts its partitions only in transitional phases, when the application changes its access pattern.

**Informed caching.** A different approach to determining access patterns is to rely on application hints that are passed to the cache management mechanism. This eliminates the need for detection, thus reducing the complexity of the policy. However, relying on hints admittedly limits applicability. Existing hint based policies require the applications to be explicitly altered to manage the caching of their own blocks. One such policy is LRU-SP [28], described in detail in Chapter 9.1.

Another example is TIP2 [87], in which applications disclose information about their future requests via an explicit access string submitted when opening a file. The cache management scheme balances caching and prefetching by computing the value of each block to its requesting application. The
benefit (decrease in I/O service time) of bringing a block into the cache is compared to the cost (increase in I/O service time) of discarding a block already in the cache.

Unlike TIP2, MC² does not require an explicit access string, but a general characterization of the observed access patterns (i.e., looping, random or sequential). In this way, it is similar to LRU-SP. This makes MC² useful for applications such as databases, where accesses can be characterized in advance into patterns.

CLIC [72] combines detection and hints by allowing clients to attach a set of hints to each I/O request. The server assigns priorities to each hint set by monitoring the history of accesses tagged by that set. This generic approach provides a “built-in” mechanism for prioritizing blocks of different clients, by comparing the priorities assigned to their hint sets.

While MC² relies on a standard set of hints, they are not application specific. The information they provide in addition to simple priorities enables MC² to choose a suitable replacement policy for each access pattern combination. Its LRU-(partition,client) allocation scheme takes into account the relative importance of blocks accessed by different clients without having to monitor them explicitly.

The ability of databases to disclose information about future accesses has made them ideal candidates for hint generation. Database query optimizers [99] choose the optimal execution path for a query. They aim to minimize the execution cost, which is a weighted measure of I/O (pages fetched) and CPU utilization (instructions executed). Once the optimal path is chosen, the pattern of access to the relevant blocks is implicitly determined. It can then be easily disclosed to the cache manager. A fundamental observation [105] was that in order for an operating system to provide buffer management for database systems, some means must be found to allow it to accept “advice” from an application program concerning the replacement strategy. The following studies base their replacement policy on this method.
A *hot set* is a set of pages over which there is a looping behavior [94]. Its size can be derived from the query plan generated by the optimizer. In the derived replacement policy [94], a separate LRU queue is maintained for each process, with a maximal size equal to its hot set size. Data sharing occurs when a query accesses a block fetched by another query. This type of access does not update the block’s position in the stack of the owner. DBMIN [34] enhances the hot set model in two ways. A hot set is defined for a file, not for an entire query. Each hot set is separately managed by a policy selected according to the intended use of the file.

By adding marginal gains to this model, MG-x-y [80] is able to compare how much each reference string will “benefit” from extra cache blocks. The marginal gain of random ranges is always positive, and so MG-x-y avoids allocating the entire cache to only a small number of them by imposing a maximum allocation of \( y \) blocks to each.

In Priority Hints [55], disk pages are managed according to the priority of the transaction which requested them. In the case of data sharing, the owner of a page is the transaction which assigns it the highest priority.

\( MC^2 \) builds upon the above policies, making a fine-grained distinction between ranges. A file may contain several ranges, each accessed with a different pattern and frequency. As in DBMIN and MG-x-y, each range is managed with a policy suited for its access pattern. Like MG-x-y, \( MC^2 \) uses marginal gains for allocation decisions, but instead of limiting the space allocated to each range, it brings every accessed block into cache to capture fine-grain locality. Most importantly, unlike the above policies, \( MC^2 \) maintains all its benefits over multiple cache levels, and chooses the replacement policy on the basis of all the users accessing a group of blocks.

A recent study [30] evaluates the benefit of *aggressive collaboration*, i.e., use of DEMOTE, hints, or level tags, over *hierarchy-aware* caching, which does not require modifications of current storage interfaces. Their results show that the combination of hints with global management yields only a
slight advantage over the hierarchy aware policies. Such conclusions may be the reason why storage vendors have not yet incorporated support for the \textsc{demote} operation. However, this study experiments with very basic hints, combined with LRU or ARC management, while it is clear from our results that simply managing loops with MRU replacement is enough to achieve much better results. Since $MC^2$ distinguishes between access patterns and manages each partition with the policy best suited for it, its improvement is significant enough to justify the use of hints and new I/O operations. We hope this will provide an incentive for incorporating these mechanisms into new system design.

12.3 Prefetching as a form of data sharing

Although prefetching is not part of our storage model, several studies of prefetching deal with sharing scenarios similar to the ones we identified in Chapter 5. Prefetched blocks are accessed in two interleaving patterns: the one in which they are read by the application, and the sequential access of prefetching. Even when the application accesses the blocks sequentially, its accesses are not necessarily synchronized with the stream of prefetches.

SARC [46] and AMP [45] combine caching and prefetching in a storage server. They are designed to prevent eviction of prefetched blocks before they are used. SARC allocates a limited amount of space for prefetched blocks, adjusting their position in the stack to avoid eviction. In AMP the degree of prefetching is adapted separately for each stream, according to its speed. In the specific context of a RAID controller, ASP [16] adjusts the allocation for accessed blocks and prefetched RAID strips using differential feedback—a method similar to marginal gains. A different approach was suggested in a recent study [118], where prefetching for multiple concurrent streams is mapped to the problem of setting inventory levels in the field of supply chain management.
When prefetching is done by the application itself, its management can be combined with additional information disclosed by the application. In TIP2 [87], the application submits prefetch requests as part of an access string, enabling the cache manager to estimate when the requested blocks will be needed. In LRU-SP [28], applications are allowed to prefetch only if their previous prefetch directives proved to be similar enough to the final access string. The management of prefetched and paged blocks is based both on the disclosed access string and on the policy chosen by the application.

Although $MC^2$ does not address prefetching explicitly, it enables clients to enjoy effects similar to prefetching, when clients use each other’s blocks. Furthermore, by addressing different sharing scenarios, $MC^2$ lays the foundations for combining caching and prefetching in a constructive manner.

### 12.4 Offline replacement

Aggarwal et al. [10] introduced the *relaxed list update problem* as a model for the management of hierarchical memory. Chrobak and Noga [35] discuss this model at length. They propose an offline algorithm for solving the problem and prove its optimality. Due to the complexity of the subject, we describe their model and algorithm only briefly enough to explain the difficulties in applying it to our storage model in Chapter 2.

In the relaxed list update problem (RLUP), a linear list is maintained in response to access requests. If the current list $x$ is $x_1, x_2, \ldots, x_k$, the cost of accessing item $x_p$ is $c_p$, where $0 \leq c_1 \leq \ldots \leq c_k$. After being accessed, item $x_p$ can be moved at no cost to any location earlier in the list, by swapping positions with arbitrary items earlier in the list. The cost of a *service* for a request sequence is the cost of accessing the requested elements, and the goal is to find an optimal service with minimal cost.

Consider a multilevel cache system consisting of $m$ fully associative caches, where $cache_i$ has size $S_i$ and access time $f_i$. This system can be modelled as
Figure 12.1: A multilevel cache system with 3 caches, modelled as an instance of RLUP, assuming $f_1 \leq f_2 \leq f_3$.

an instance of RLUP where the size of the list, $k$, is the aggregate size of the caches, and the cost of accessing each item is the access time of the cache it is in (see Figure 12.1). The optimal algorithm for this model is closely related to Belady’s MIN [23] for a single cache. Instead of evicting the block seen again furthest in the future, it simply moves the corresponding item to a higher cost position in the list, in a series of swap operations.

In a real system, these swaps can be viewed as DEMOTE operations, moving a block from one level in the hierarchy into a lower level. While in the theoretical model these operations carry no cost, in reality a DEMOTE operation can incur additional I/O delay. This can happen when a READ request is delayed until a block is demoted to make space for the requested block. When the cost of the swap operation is non-zero, the optimality of the algorithm does not hold.

To address this problem, Gill [44] uses a lower bound on the optimal I/O response time in the analysis of new cache replacement policies. For cache$_1$ of size $S_1$ and cache$_2$ of size $S_2$, miss$_1$ is the number of cache misses of MIN on a cache of size $S_1$, and miss$_{1+2}$ is the number of cache misses of MIN on a cache of size $S_1 + S_2$. For a trace with $R$ I/O requests, the lower bound on average I/O response time is $[(R \times C_1) + (miss_1 \times C_2) + (miss_{1+2} \times C_{Disk})]/R$. The extension of this lower bound for multiple clients is valid only under the assumption that the relative order of requests from various clients is fixed. Since our simulations reflect more complicated scenarios, we use the lower bound on the optimal I/O response time for a single client.
12.5 Storage system design

Modern storage systems are designed as standalone platforms, separated from their users and applications by strict protocols. This modularity allows for complex system layouts, combining hardware from a range of manufacturers. However, the separation between the storage and the application layers precludes interlayer information sharing that is crucial for cooperation between the system’s components—cooperation we believe will lead to substantial performance gains.

Many recent studies attempt to bypass this inherent separation: the levels gather knowledge about each other by tracking the implicit information exposed by current protocols. For example, in several recent studies [74, 97], the operating system extracts information about the underlying disk queues and physical layout from disk access times. In the gray-box approach [13, 101], the storage system is aware of some operating system structures (such as inodes), and extracts information about the current state of the file system from operations performed on them. In C-Miner [70], no knowledge is assumed at the storage level. The storage system uses data mining techniques in order to identify correlations between blocks. The correlations are then used for improving prefetching and data layout decisions. PFC [119] is a hierarchy-aware prefetching coordinator situated at the storage controller. It monitors upper level access patterns to dynamically adjust the aggressiveness of lower level prefetching.

In contrast, other studies explore the possibility of modifying traditional storage design. AWOL [21] and DiskSeen [38] propose to integrate disk scheduling with prefetching and write optimization in the file system cache. Others suggest a new level of abstraction, such as object storage [15]. Other work has focused on introducing new storage management architectures aimed at optimizing database performance [53, 96].

MC$^2$ belongs to a group of schemes which require minor modifications to existing storage interfaces. The DEMOTE operation [114] was discussed earlier.
in this section. QuickMine [103] requires the application to attach a context identifier to each I/O request. ULC, described earlier in this section, attaches a level tag to each READ request forwarded to a lower cache level. For $MC^2$, the required minor modifications to the storage interface enable the storage system to exploit the information available in other levels, without incurring the overhead of extracting and deriving this information. We believe that the additional complexity is worthwhile, given the large performance benefits presented in Chapter 10.

Table 12.1 summarizes the non cooperative policies discussed in this study. The policies are compared according to their ability to perform well in more than one cache level, to achieve exclusiveness in a multilevel cache system, to use application hints, and to handle data sharing by different users.

### 12.6 Cooperative P2P and storage systems

Traditional storage caching approaches assumed altruistic clients in centrally managed systems. Most of them consider the global system’s optimization as the only objective [40, 65, 95]. Some exploit locality of reference within a client’s workload by allowing the clients to manage a private LRU partition [37, 85]. None of the above approaches consider the delay incurred by a client serving peer requests. In Chapter 10 we show that this delay may mask, or even exceed, the delay saved by cooperation.

Recent cooperative caching studies address load balancing between the clients. Experiments with the Shark network file system [11] show significant differences in the bandwidth served by different proxies. In NFS-CD [20], the load of serving popular files is split between “delegates” when it is detected. LAC [57] equalizes cache utilization between all clients. Similarly, distributed storage systems [66, 104] and NoSQL databases [67] use hash keys to evenly
Table 12.1: Summary of non cooperative cache management policies, the extra information they require, whether they are suitable for use in first and second level caches, and whether they specifically address data sharing.

<table>
<thead>
<tr>
<th>Policy</th>
<th>1st level</th>
<th>2nd level</th>
<th>Extra Information</th>
<th>Exc.</th>
<th>Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU,MRU</td>
<td>√</td>
<td>√</td>
<td>(none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEAR,UBM,AFC,PCC,MRC</td>
<td>√</td>
<td></td>
<td>file,application,PC (none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWOL, DiskSeen, ASP</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU-SP</td>
<td>√</td>
<td></td>
<td>pattern,priority explicit trace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HotSet,DBMIN, MG-x-y</td>
<td>√</td>
<td></td>
<td>query plan</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Priority-Hints</td>
<td>√</td>
<td></td>
<td>transaction priority indirect hints</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>X-Ray, Write hints</td>
<td>√</td>
<td></td>
<td>(none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiIQ,ARC, CAR, Argon</td>
<td>√</td>
<td></td>
<td>(prefetch stream)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SARC, AMP</td>
<td>√</td>
<td></td>
<td>(data mining) client hint sets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C-Miner</td>
<td>√</td>
<td></td>
<td>context,process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIC</td>
<td>√</td>
<td></td>
<td>(none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QuickMine,BORG</td>
<td>√</td>
<td></td>
<td>client instructions experimental sampling</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Demote</td>
<td>√</td>
<td></td>
<td>(machine learning)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global-L2,PROMOTE</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ULC</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACME</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-resource allocation</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uCache,LAC</td>
<td>√</td>
<td></td>
<td>cooperative caching ranges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$MC^2$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Technion - Computer Science Department - Ph.D. Thesis PHD-2012-12 - 2012
distribute replicas and load between dedicated storage servers.

The distribution of load between cooperative caches should not only be addressed explicitly as a major objective, but should also reflect the heterogeneous workloads and objectives of the caches themselves. Autonomous clients may not comply with protocols that degrade their performance. Similarly, resource owners can be expected to share them only within systems that guarantee a specified degree of fairness, and application owners are likely to prefer platforms that guarantee their objectives are met.

Peer-to-peer (P2P) systems constitute a large number of independent, autonomous peers, and an incentive mechanism which motivates peers to invest their resources in serving other peers. System efficiency is evaluated by measuring its global throughput. In BitTorrent [36], peers upload data to other peers in a tit for tat manner, favoring peers that upload to them. The reputation system in [88] enhances this mechanism. Alternatively, currency and balance based mechanisms are used for packet forwarding [79] and multicast [62] in ad hoc networks.

In such systems, peers cooperate according to their location or group assignment. The incentives help induce cooperation between peers that can choose not to cooperate. However, these are short-term cooperative transactions that usually involve a single operation. Such schemes are not suitable for stateful systems such as caches. There, cooperation is most beneficial if a client can evict a block from its cache, relying on its peers to store the block and provide it on request, for an agreed upon period of time.

Auction based mechanisms were suggested for managing resources in distributed systems [22], according to workload priorities and the load on the system’s components. Similarly, jobs in heterogenous grid systems can be scheduled using various economic models for setting the price of services on a supply-and-demand basis [27]. These models accurately capture the interaction between conflicting objectives of the different entities, and allow for long-term agreements. However, the resulting mechanisms usually carry
significant computation and network overheads, which make it difficult for them to adapt quickly to dynamic workload changes, as required from cache management algorithms.

12.7 Theoretical models for cooperation

When clients are altruistic, the objective in the cache placement problem is to minimize a global cost, e.g., average I/O access time. This can be formulated as a min-cost max-flow problem, which can be solved in polynomial time. Thus, the global optimum can be found relatively efficiently as long as the cost of cooperation, sending blocks to peers or fetching them from peers, is small enough compared to the cost of a disk access – choosing the blocks to store in the aggregate client caches is much more important than choosing in which cache to place them. With selfish clients, a solution must also satisfy the constraint that every cooperating client has an access cost lower than its cost without cooperation, likely making the problem computationally hard.

Noncooperative game theory addresses selfish agents, but lacks a mechanism that enforces agents’ obligations to one another. Thus, conventional solution concepts are problematic and do not accurately reflect a feasible, cooperative, cache placement. For example, a Nash equilibrium must satisfy the condition that no single player has anything to gain by changing only its own strategy unilaterally. In the context of cooperative caching, a client can change its strategy and refuse to serve peer requests, while still having its own requests served. Therefore, the only Nash equilibrium is no cooperation at all.

Another widely used solution concept is Pareto optimality. A solution is Pareto optimal if it is impossible to reduce the cost of one player without increasing the cost of any other player. Although it captures the tradeoffs between different players’ objectives, the cost of a solution can be arbitrarily higher than the system’s global minimum. This is a general drawback of
Pareto optimality [14].

Other models, such as cooperative game theory or market models, can successfully enforce cooperative commitment among players. However, these models are generally computationally hard [110]. They are not applicable for large scale systems with numerous clients and cache buffers, and most certainly not for dynamic systems with changing workloads.

12.8 Energy management in storage systems

Significant effort has been invested in reducing the energy consumed by the disks. Energy aware cache management algorithms [78, 116] reduce energy consumption by allowing disks to spin down completely for long periods of time, or to spin at lower rotational speed [123]. These algorithms do not deal with energy consumed by the clients. Cache management can also reduce the energy consumed by the memory itself [117].

Another approach is to consider the entire storage server. Sierra [109] powers down entire groups of servers and their disks during periods of low I/O loads, while SRCMap [111] accommodates migration of virtual machines to the most power efficient servers. On a smaller scale, the Cinder [92] operating system for mobile devices controls the energy consumed by applications to extend battery life. In this context, our study is the first to consider the energy consumed by the CPU while performing different I/O operations.

A recent study [98] provides an extensive analysis of file system performance and energy efficiency. Their results present strong correlation between improving file system performance and improving its power efficiency. This is also the case with the noncooperative caching policies we experimented with. However, our analysis extends to additional I/O operations that may improve performance at the cost of increased power consumption.

The Green Grid defines data center energy productivity[5] as tasks complete / energy consumed. Similarly, computation per provisioned watt, is used
in [50] for distributing provisioned power among hosted workloads. These
measures are closely related to the energy delay product. However, they
intentionally isolate the work being done by the application from the back-
ground work being done by the data center, effectively neglecting the cost of
cooperation between clients.
Chapter 13

Conclusions

We defined a model for multilevel caching and proposed a policy which solves the five problems that can occur in a multilevel cache. Blurring of locality of reference in lower level caches, data redundancy, and lack of informed caching at the lower cache levels all limit the performance of the cache with a single client. Negative sharing scenarios and competition for resources are introduced in the presence of multiple clients. None of the existing policies (summarized in Table 12.1) address all these problems.

Our proposed solution combines management in two levels. In the local, per-client level, Karma approximates the behavior of the optimal offline algorithm, MIN, reducing I/O response times close to the theoretical lower bound. Like MIN, it aims to optimize the cache content by relying on knowledge about the future, instead of on information gathered in the past. Karma uses application hints to partition the cache and to manage each range of blocks with the policy best suited for its access pattern. It saves in the cache the blocks with the highest marginal gain and achieves exclusive caching by partitioning the cache using DEMOTE and READ-SAVE.

At the global, system-wide level, $MC^2$ leverages the same hints for choosing the best replacement and allocation schemes for the combination of access patterns. More importantly, it uses these hints to derive the degree and type
of data sharing between clients, and chooses between inclusive or exclusive caching accordingly.

We have demonstrated that the minor changes in the storage interface used by $MC^2$ yield substantial performance gains. $MC^2$ relies on passing simple application hints to the storage system, and on two non-standard I/O operations. When combined with a smart choice of allocation and replacement, the effects of these changes are similar to those of cooperation between clients, since a client can rely on the presence of valuable blocks in the cache, without fetching them from the disk itself.

Explicit sharing of resources is becoming increasingly popular, entailing significant benefits for selfish entities in distributed systems. We demonstrated the limitations of existing practical models in addressing cooperation between selfish caches. The theoretical models addressing selfish entities do not provide an applicable framework for managing real systems. As resource consolidation grows both in scale and popularity, a new model for managing cooperating caches is necessary. We showed that such a model should explicitly address the cost of cooperation, as well as allow participants to calculate the utility they derive from their cache and from cooperation. The objective of increasing the utility of all participants must be considered together with the objective of optimizing global system performance.
Bibliography


הבעקבות המחלשות המתרחשות עליל של המודלים הקימיים, נדרש מודל איכון וحوا שיענה על שאלה זו: 

ועקרון: יעיל לכלל בפרפורמאגי המחלשה של תחנת הפגיעת, רגיל לאפזר את היציבות התנודה מישורית הממסמנות, כןочный, ניתן להתאדים במכס של חלוקה האולטトラ על התנודה התנודה של שיתוף פעילותكوלכול התנודה ומשלת האמה לשתי פעילות ביניהם.

ההרחב של פלドラ מ العدو מCompound שיתוף


 nhậtים של פל德拉 מ العدو M Compound שיתוף פעילות ביניהם, שענה על שאלה זו: 

ברקמת והרשעה של גונייה מדליית מדליית מדליית מדליית מדליית מדליית מדליית את הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת האcompanית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת(audio:he)

 nhậtים של פל德拉 M Compound שיתוף פעילות ביניהם, שענה על שאלה זו: 

ברקמת והרשעה של גונייה מדליית מדליית מדליית מדליית מדליית מדליית מדליית את הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת האcompanית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת האcompanית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת(audio:he)

 nhậtים של פל德拉 M Compound שיתוף פעילות ביניהם, שענה על שאלה זו: 

ברקמת והרשעה של גונייה מדליית מדליית מדליית מדליית מדליית מדליית מדליית את הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת(audio:he)

 nhậtים של פל德拉 M Compound שיתוף פעילות ביניהם, שענה על שאלה זו: 

ברקמת והרשעה של גונייה מדליית מדליית מדליית מדליית מדליית מדליית מדליית את הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת(audio:he)

 nhậtים של פל德拉 M Compound שיתוף פעילות ביניהם, שענה על שאלה זו: 

ברקמת והרשעה של גונייה מדליית מדליית מדליית מדליית מדליית מדליית מדליית את הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת הא companית ואת/audio:he)
האנטרופיה הבינוליונית נמוכה יותר

ראודס מסمجتمع איה שש רוזח וקצץ בלתייון אובֵּרוֹ, התמקה. זהול וכתם. מסרוריי, משביכ אסם
ותשובה אוחזת מבחר ארון בדף. בהתח帳, התaciente גוט惛ב שיתוף הווה בעברו של שמותינו
נבעלו משותפת, נשיאת המיתר איה מנורות במgıונום המיתרים של העמותה..
לאחרונה מתבצע אעות ומ[Byte]כバル מיד דוגל הברוח, בארץ ליפות המיסטיבים קשורים לישון שנותנה.
ונהגאת ופועלת ולヂולות רשתות לשון, עונח וחושב אלפסיומס, ורגל עזב ורצוי גולוים. גד המידה
המתרחב של של גוד ידידו השמות ואפליקציות, שלח אינטרסים פתרון. ח进城 מכים ובית
המכיתות של גוד המhettoות של, אכך בכל ידידים המיתרים הנותרים, אלא גם בכל הזרעים
לאטן ביניהם י다는 על השמות, כתיב עד מגיונום רמת דרום.

גוטל טמונות שיתופים מחוז אוכז בזוי לגדול מתוכנות כאל. המינים או יש מאבק על פרש. שלבר מרגז
הגרכ על יד ב耴ל. ברקוט מידי יכה להתרחש עם ביני משותפים חי, למלש נגני המותו פּאֵליריהוות.
הכניס הפרל, תומך ארנסט, או קאשר מחלקה שנות חנס והראות ואצלות דיתנים. ספגיה כה
אלא, גישה לא ההולכי מתואם של עמיד איה שלד מצביוניות לינומר גישה לשלום האסף הקטן המשיווק.
אודאילטו, מחולים משתפים יושכפל בפגועי המיתרים המיתרים רק בפּאֵליריה הלהזגו בין יאו אבר בקוב
בנישא איליהם, ואליוס לאלקוליאוות.

שתוות פועל בתוככייס כאליא יכל לעצם שיכונים על מיות ולחסונות המשותפים והרצועי המערית לא, או
עלול מגבר התרחק נטפשת סכילה לפוגעת בברניין המיתרים. פכימן שדואר במעשיים.
אומואים ולעלא באלように, המש יזימה לחרב ללשון פּאֵליריה התא.

נוסחת ניוו קירוט מתייחסות לבלווי השפעה של המשותפים המיתרים על משא בשותה, או במקדה
שני המיתרים הוא ממאב פרש של כל אדם במשותפים, לזכות והחיי ברזים פּאֵלירית
לפּאֵליריה חנוכיים של המשותפים. לע מנח עלודים אוחיות לשתות פּאֵליריה. מחקריהם בחנית Attribution alcan מאסם אל ימודים שיתוף פּאֵליריה
הכימיה ב יעל המשותפים אאוסים, או多媒体ים יגוררים איונים מִמורפקים יכיף ילודד שיתוף פּאֵליריה
במעריך זהות ומא סכינים. מִמורפקים Attribution עלしましたRandir וקシー והשונים אלו מתאימים
לשימש במעריך Attribution דרגות זהות והShortly תגבה מורה בום אמת.
ההנמכה של מדיניות שניתן המטמון בתוך שיתוף בור לכל מספר, נכונה שבה המוקצה וגם שבאין כל תבנית לניהול ידי ה南沙נו נינו. המשותפים השליחים בתקופה, כפי שהבוודא, הם הפעילות של מספר, מדיניות משמשת על כל תחומים מחולקים למספר פלטCHOICE2 של חברת המשותפים שבかり הפילו בתקופה. הטבע של כל האやはりים מצויאים לערוך של מחיצה של שגיאה. מחוון של מחיצות של תחתון המשותפים של שלושה גלובליים וכללים משלים מחיצה של שגיאה. מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה גלובליים וכללים משלים מחיצה של שגיאה. מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותפים של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיאה מחוון של מחיצות של תחתון המשותños של שלושה Глобализация של שגיא
מדיות תקשורת מהנחתים את הלוקס א"י להגלה בין פעילות במערכת ובצירוף לפרס废旧ות בתלולו.
ולכן, נဓלו את הפרטיםFileStream הנמצא על העוגה נוגעת בحمايةstral למקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגה נוגעת בحمايةstral למקסם.
ולכן, נ命周期 البلادFileStream הנמצא על העוגה נוגעת בحمايةstral למקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגa נוגעת בحمايةstral למקסם.
ולכן, נ命周期 البلادFileStream הנמצא על העוגa נוגעת בحمايةstral למקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגa נוגעת בحمايةstral למקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةstral למקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةstral למקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةstral למקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةstral למקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةstral למקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةtraî על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةtraî על מקסם.

לנ_paid את הפרטיםFileStream הנמצא על העוגא נוגעת בحمايةtraî על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةtraî על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةtraî על מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةtraî על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةtraî על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةtraî על מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحماية trì על מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטים FileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.

לנ_paid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנpaid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנPaid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנPaid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנPaid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.

לנPaid את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.

לנPaid את הפרטיםFileStream הנמצא שם العוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
ולכן, נ维奇איצן את הפרטיםFileStream הנמצא שם העוגא נוגעת בحمايةetri מסך של מקסם.
המטמון

משמש את חותם מפורש בכתובת אחות כדי להصحة את מדריך הלחימה של שכרב גניש אלה לשלאחרונה. מחקר

הcrets לשגרת כנסים לשפר את ביצועי המטרות בameleon הלשון של צד צבאי המטרות כולה. בקופסדורטיות

רבות לשגרת הכאזה, מסומניםbeiten של_preference opc תגי וירוס הראיריאציה דר ישבחת (א. תלם). בץ

שכתביה, המגזרות הפריטים והקשרים הם הפריטים של מטרות של מטרות והסיבות של אטרואים. האטרואים

כלילולים ההלшение באור רם לשמור עד היום, פרץ שלמרש על תצלימים איבר הורמאיל או המידיע תמרות נפגש

בכל הרמות פיתר לעילונונ. ההנבורה של אטרואים alma כלל שלמרש באיפור ומ텐וגות של צד צבאי המטרות כולה.

המטמון

משלת הסופיות מעשה בכתובת אחות כדי לה FontStyle את מדריך הלחימה של שכרב גניש אלה לשלאחרונה. מחקר

הcrets לשגרת כנסים לשפר את ביצועי המטרות בameleon הלשון של צד צבאי המטרות כולה. בקופסדורטיות

רבות לשגרת הכאזה, מסומניםبيدvenirשל.preference opc תגי וירוס הראיריאציה דר ישבחת (א. תלם). בץ

שכתביה, המגזרות הפריטים והקשרים הם הפריטים של מטרות של מטרות והסיבות של אטרואים. האטרואים

כלילולים ההלшение באור רם לשמור עד היום, פרץ שלמרש על תצלימים איבר הורמאיל או המידיע תמרות נפגש

בכל הרמות פיתר לעילונונ. ההנבורה של אטרואים alma כלל שלמרש באיפור ומ텐וגות של צד צבאי המטרות כולה.

המטמון

משלת הסופיות מעשה בכתובת אחות כדיה להصحة את מדריך הלחימה של שכרב גניש אלה לשלאחרונה. מחקר

הcrets לשגרת כנסים לשפר את ביצועי המטרות בameleon הלשון של צד צבאי המטרות כולה. בקופסדורטיות

רבות לשגרת הכאזה, מסומניםبيدvenirשל.preference opc תגי וירוס הראיריאציה דר ישבחת (א. תלם). בץ

שכתביה, המגזרות הפריטים והקשרים הם הפריטים של מטרות של מטרות והסיבות של אטרואים. האטרואים

כלילולים ההלшение באור רם לשמור עד היום, פרץ שלמרש על תצלימים איבר הורמאיל או המידיע תמרות נפגש

בכל הרמות פיתר לעילונונ. ההנבורה של אטרואים alma כלל שלמרש באיפור ומ텐וגות של צד צבאי המטרות כולה.

המטמון

משלת הסופיות מעשה בכתובת אחות כדי להصحة את מדריך הלחימה של שכרב גניש אלה לשלאחרונה. מחקר

הcrets לשגרת כנסים לשפר את ביצועי המטרות בameleon הלשון של צד צבאי המטרות כולה. בקופסדורטיות

רבות לשגרת הכאזה, מסומניםبيدشيرשל.preference opc תגי וירוס הראיריאציה דר ישבחת (א. תלם). בץ

שכתביה, המגזרות הפריטים והקשרים הם הפריטים של מטרות של מטרות והסיבות של אטרואים. האטרואים
אני מודה לטכניון, למכון האס פלאטנר ולמשרד המדע והטכנולוגיה על התמיכה הכספית בחרשי והרשים במחקרי התמחותי.

בראש ובראשונה ברצוני להודות למנהליי, פרופ' אסף שוסטר ודר' מיכאל פקטור. הם הביאו לי עונשים ומצאי אמנים במחקרי התמחותי, עזרו לי במחקרי התמחותי, וǦיוו לי כל האפשרים但我ים ל發展 конце הקריירה конце הקריירה,냅ו וتقدمו לי את הת訓ול על בתת-ממסים, נרחב ותבנאות. היה לי הזכות לעבוד איתם יחדיו.

ברצוני להודות לפרופ' קאי לי, שנתרביה עיון-added על מחקרי מערכות. תובנותיו והעצמות לייוו אותי לכל אורך לימודי.

ברצוני להודות לורי גולן והדרו בן-יוסף, שלימדו אותי לשאול, לחקור, ולה ايضا למידה. הם תמכו בי בכל ההיבטים של לימודי, מבית הספר היסודי ועד בית הספר לממנים. אני מודה להם על התלהבותם, ממן ור当たりי, יוזמה ו estratégia של לי לממש את המטרה.

לברנור, אני מודה להם, אבי י(strict, על האב ואביהם על צורים של้า מערכי בעיון, מה=mysqliים בעיון, abbreviated, abbreviated, abbreviated, abbreviated, abbreviated, abbreviated.

מנועי התסיטים והערכות על מדיניות הובלה. ואני обеспותי באיתן עבד.
ניהול מטמון רב שכבתי בעזרת רמזים מהאפליקציה

חיבור על מחקר

לשימ מילוי חלקי של דרישות לקבלת תואר דוקטור לפילוסופיה

גלית ייגר

הנהלת לפקט טכנולוגיה - מכון טכנולוגי לישראל

איליר תשע"ב

מאי 2012
ניהול מטמון רב שכבתי בעזרת רמזים מדאדים ומואפים

גילה י وغيرها