Topology-Aware I/O Caching for Shared Storage Systems*

Seung Woo Son
Argonne National Laboratory
Argonne, IL 60439
Email: sson@mcs.anl.gov

Mahmut Kandemir, Yuanrui Zhang, Rajat Garg
The Pennsylvania State University
University Park, PA 16802
Email: {kandemir,yuazhang,rgarg}@cse.psu.edu

Abstract

The main contribution of this paper is a topology-aware storage caching scheme for parallel architectures. In a parallel system with multiple storage caches, these caches form a shared cache space, and effective management of this space is a critical issue. Of particular interest is data migration (i.e., moving data from one storage cache to another at runtime), which may help reduce the distance between a data block and its customers. As the data access and sharing patterns change during execution, we can migrate data in the shared cache space to reduce access latencies. The proposed storage caching approach, which is based on the two-dimensional post-office placement model, takes advantage of the variances across the access latencies of the different storage caches (from a given node's perspective), by selecting the most appropriate location (cache) to place a data block shared by multiple nodes. This paper also presents experimental results from our implementation of this data migration-based scheme. The results reveal that the improvements brought by our proposed scheme in average hit latency, average miss rate, and average data access latency are 29.1%, 7.0% and 32.7%, respectively, over an alternative storage caching scheme.

1 Introduction

I/O caching has been a popular research area to address the problems associated with high disk latencies and has been employed at different layers of the I/O stack (e.g., application level, file system level, and disk level). Most of the file-level I/O caching (also called *storage caching* in this paper) schemes in parallel architectures are, however, architecture agnostic; that is, the topology of the underlying network of the parallel machine does not play a major role in the caching decisions these schemes make. The main contribution of this paper is a *topology-aware I/O caching scheme.* The proposed storage caching scheme determines, for each data block, the most appropriate location (storage cache) in the architecture to store it at any given point during execution. This is accomplished by formulating the caching problem as the *two-dimensional post-office placement problem.* As a result, the chosen cache location for a data block may not necessarily be one of the caches of the customers of that block, though it is globally optimal when considering all customers of the block. The proposed scheme is dynamic; As a result, the location of a data block in the shared cache space can change when its access and sharing pattern changes, and this helps us adapt caching to dynamic variations in data access and sharing patterns at runtime. While we focus mainly on a two-dimensional mesh type of network in this paper, our approach can be extended to other network topologies, as long as the topology and the access latencies associated with it are exposed to our storage caching scheme. We also discuss in this paper how the proposed approach can be extended to allow data replication (i.e., having multiple copies of a given data block in different caches).

We implemented our storage caching scheme and tested its effectiveness using traces collected from actual disk-intensive applications. In our experiments, we also compared our approach to several other storage caching schemes. Our results with six disk-intensive applications show that (1) the proposed storage caching scheme is effective in practice and the performance difference between our scheme and the next-best scheme (among all the storage caching schemes we evaluated) is about 17.1% on average under the default values of our experimental parameters and (2) the savings achieved by our caching scheme are consistent across different network sizes, different block sizes, different placements of disks in the mesh, and other experimental parameters.

2 Target Architecture

In our target architecture, we assume a set of nodes are connected to each other using a network topology. All nodes are assumed to function as compute nodes (that is, they can execute application threads), but only a select set of nodes are attached storage (disks). As a result, these nodes can function as I/O nodes as well. While the nodes

*This work is supported in part by NSF grants #0927949, #0833126, #0821527, #0724599, #0720749, and # 0621402 and by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

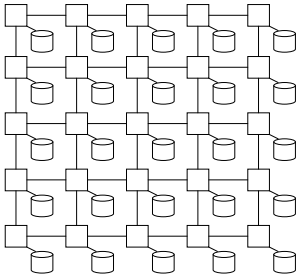

Figure 1: High-level view of the target architecture, where all nodes have disks. Each node allocates a portion of its available to storage cache.

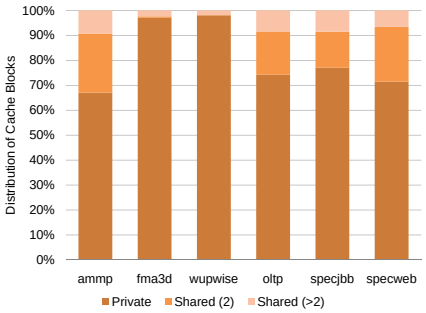

Figure 2: Distribution of the private and shared data blocks in the storage cache space.

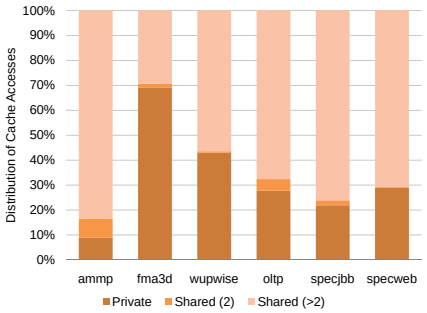

Figure 3: Distribution of accesses to the data stored in the cache space.

in many current parallel systems and I/O clusters are interconnected through cascaded switches arranged in a tree, we believe that more specialized network topologies, such as two-dimensional and three-dimensional mesh, hypercube, and torus, need to be utilized in order for a large number of nodes to work together with high bandwidth and very low latency. Each node of this parallel architecture is assumed to execute one thread (that belongs to an application) and uses a reserved portion of its main memory space as the *storage cache*, which is used to cache frequently used disk data. Note that the architecture has a shared view; that is, a *data block* (which is the granularity of caching and data migration/replication in this work) can be cached in any of the storage caches in the system; and, in fact, determining the ideal location (storage cache) to cache a given data block is our main goal. To keep our discussion simple, we assume that the storage caches in different nodes are of the same size, though our approach can also work with cases where different caches have different capacities. In the rest of the paper, the storage caches attached to nodes are said to form a *storage cache space*. Note that this is a *shared space* in the sense that a node can access data cached in any storage cache. However, the cost of an access depends on the *distance* (number of hops) between the requesting node and the node that contains the data in its storage cache. The disks can be attached to any of the nodes. Figure 1 depicts our default disk placements for the two-dimensional grid. Depending on the sharing patterns it exhibits, a data block can move (migrate) from one storage cache to another during the course of execution. If desired, a data block can also be replicated across multiple storage nodes.

3 Data Sharing in the Storage Cache Space

We start by presenting in Figure 2 the distribution of the private and shared data blocks, over the entire execution period of our applications, when each application is parallelized using 25 threads on a 5×5 mesh (the details of our experimental platform will be given in Section 6). If a data block is accessed by only one node during its lifetime (from its fetch from disk to its eviction from the storage cache space), it is tagged as "private"; otherwise, it is referred to as "shared." In this barchart, Shared (2) represents the data blocks shared by two nodes, and Shared (>2) represents the data blocks shared by more than two nodes. We see from these results that an overwhelming majority of data blocks are private (about 80.7% on average). In comparison, Figure 3 gives the distribution of *accesses* to the data stored in the cache space. One can observe that the fraction of accesses to Private, Shared (2), and Shared (>2) data are, on average, 33.3%, 2.8%, and 63.9%, respectively. Putting the results presented in Figures 2 and 3 together, we observe an interesting trend regarding the sharing behavior of the data resident in the storage cache space. While the majority of the storage cache resident data is private, most of the accesses to the cache are for data shared by more than two compute nodes. Consequently, careful placement of this (small amount of) shared data in the storage cache space can be critical for performance.

4 Our Approach

The target storage cache space we consider in this paper is similar to those considered by the prior efforts [5] [10] [21] [6] [20]. We assume that each node of our architecture is connected to each other through a network (a 2D mesh in our case). We also assume in our default configuration that each node has its own disk (see Figure 1). In our experimental evaluation, we take into account latencies due to accesses to disks, communication latencies (including latencies due to network contention), and latencies due to storage cache accesses. We further assume that the certain portion of memory space in each node is used as a local buffer cache space. These local cache spaces, called *storage caches* in this work, are maintained by each node using the LRU replacement policy. To utilize this aggregated cache space better, we also assume that each cache in this architecture can be used by other nodes as well. The

main difference between our design and the previous proposals is in the cache space management policy, which includes both a replacement policy and a migration policy, discussed in the next two subsections. Note that, while this storage cache space is shared, the latency of an access depends on the distance between the requesting node and the node that holds the requested data block. We use the term "local cache" to denote the storage cache attached to a node. However, all caches are accessible by all nodes in this system. From a node's perspective, any nonlocal cache is referred to as "remote cache." In our approach, there are two types of migrations in the shared storage cache space: *victim related migrations* and *sharing related migrations*. The first one is triggered by displacement of a data block from a cache, whereas the second one is triggered when there is a change in sharing patterns of a data block.

4.1 Initial Placement and Victim Migration

When a data block is brought from disk to the shared storage space, we place it into the cache of the requesting node just like most of existing techniques. We note that the initial placement of data from disk to the shared cache space may involve a replacement, which may in turn trigger *cascade migrations* in the storage cache space. Each node has limited cache space, depending on the amount of physical memory allocated for storage caching. Therefore, placing a data block into a particular node may require the eviction of a data block (*victim*) from that cache. At least two questions need to be answered to solve this replacement problem: How are we going to determine such a victim? and, Are we going to evict this victim from the entire shared storage cache space?

Our answer to the first question is to use LRU to select a victim from the storage cache to which we want to bring the new block. The next question to address is whether we directly evict such a victim to disk where the block is originally brought from. In a conventional system, the answer to this question is "yes" because, for each node, its data blocks can reside only in its local cache space, and victims are sent to the disk. However, this option clearly reduces the utilization of the aggregate storage cache space (e.g., a node with a large working set cannot make use of the idle cache capacity that belongs to a neighboring node). Therefore, we need to design a scheme that can handle victims more carefully. Our idea is to determine the location of the victim based on the *local cache miss* ratio, i.e., the fraction of cache misses that occur during a certain period of time in each node in our target architecture. Based on the storage cache capacity in each node and the characteristics of the workload, we determine a threshold value for the local cache miss ratio, denoted T_m in this paper. For a node N in our architecture, when the (local) cache miss ratio is equal to or higher than T_m, we set the flag (denoted as $N.flag$) attached to that node to 1. If the cache miss ratio is less than T_m, the flag is set to 0.

```
INPUT: Source node N_s that throws the victim
OUTPUT: Destination node N_d for the victim

 1: /* the following while loop is executed in background */
 2: while there is a cache miss do
 3:    update counters for cache misses
 4:    if miss ratio >= T_m then
 5:       set N.flag to 1
 6:    else
 7:       set N.flag to 0
 8:    end if
 9: end while
10:
11: /* the following routine is executed when a victim block has to
    be displaced from its current cache */
12: if N_s.flag = 0 then
13:    N_d = NULL /* evict from the cache */
14: else if N_s.flag = 1 then
15:    if ∃N_i that d(N_i − N_s) = 1 and N_i.flag = 0 then
16:       N_d = N_i
17:    else if ∃N_i that N_i.flag = 0 then
18:       N_d = N_i
19:    else
20:       N_d = NULL /* evict from the cache */
21:    end if
22: end if
23: if victim has been modified then
24:    write back the victim to disk
25: end if
```

Figure 4: Migration policy implementation for a victim data block (function d($N_i - N_j$) gives the Manhattan Distance between node N_i and node N_j). The victim is selected by the LRU policy in each node.

The combination of the local and remote node flag values determines the migration target for a victim data block. The pseudo-code for our migration policy is given in Figure 4. The idea is that, if the flag ($N.flag$) of the node evicting the data block is set to 1, it has higher ability of keeping its victim within the storage cache space (instead of sending it to disk) by checking more nodes' storage caches and asking one of them to accommodate this victim if possible. On the other hand, if the flag of the evicting node is 0, its victim will be evicted directly from the entire cache space (accompanied with a possible write back to the disk if it has been modified while residing in the storage cache). The rationale behind this strategy can be explained as follows. If $N.flag$ is 1, the local storage cache attached to node N is under pressure: it cannot hold the entire working set of the code running on that node and experiences frequent misses. In this case, it may be more beneficial to try to keep the discarded data block within the shared storage space. On the other hand, if $N.flag$ is 0, the local cache is not experiencing many misses. Hence, we may afford to send the displaced block to the disk. While even in this case it may look useful to keep the block in the shared cache space, we emphasize that the cache space is shared across multiple nodes and whenever we keep some block in this space, this will have a cost when other blocks are considered. While the victim migration policy given in

Figure 4 is our default policy, as will be discussed later, we also conducted with other policies as well.

Overheads. Several overheads are associated with our migration policy implementation. First, each node needs to keep track of its miss ratio and update flags. Maintaining a flag in each node is not expected to incur excessive overhead in practice. We need only to maintain two counters in each node: one for block accesses and the other for cache misses. The updates to these counters are triggered by the local cache miss monitor in each node, which can be implemented by using the performance counters available today in many modern processors [24]. All overheads incurred by our scheme are included in the experimental results presented later.

Locating a Data Block. How to locate a data block in the large shared storage cache (i.e., search policy) is an important issue. Since a data block can reside in any node, we employ a *multistep checking scheme* that first checks the local and neighboring nodes, and then sends requests (if necessary) to remote nodes until we determine whether we have a cache hit or miss (we found this multistep policy to be less costly than an alternative search scheme that checks all the nodes in parallel).

4.2 Migration of Shared Data Blocks within the Storage Cache Space

To implement a data migration scheme, we need to address two questions: How can we track the access patterns so that we can easily calculate migration targets for frequently accessed shared data blocks? and, How to trigger the migration of these shared data blocks? To address these two questions, we maintain, for each block, an array of $m \times n$ counters, where m and n are the number of rows (X direction) and columns (Y direction), respectively, in our two-dimensional mesh. As an example, in the default configuration, (i.e., 5×5 mesh topology), we have an array of 25 counters. Every time a node N_i requests a data block, the corresponding counter maintained by that block is increased by 1. The total number of accesses to a data block, which can be calculated by summing up all the entries in the array, is used for determining whether the block is frequently accessed. Once this value reaches a *threshold* (denoted as T_{tr}), the computation for determining the migration target is triggered. We first determine, as explained below in detail, the target node for the data block. If the calculated target node is not the current node (i.e., the one whose local storage cache holds that block), this data block is migrated to the determined target node. The widths for each counter are same (i.e., 6 bits) because the default value of T_{tr} in our experiments is 50 (see Table 1). All counters associated with a data block are reset to 0, whenever a block is migrated. The total space requirement for maintaining counters to track access patterns is 150 (25×6) bits per block. Since a typical size of a data block is 4 KB in our default setup, the extra spaces needed for maintaining the counters are negligible.

The average access latency of a shared data block can be estimated as:

$$T = \sum_{i \in \mathcal{N}} c * a[i] * d(i,j) / \sum_{i \in \mathcal{N}} a[i],$$

where j is the ID of the node where this data block currently resides, i is the ID of the requesting node (which belongs to a set, $\mathcal{N}$), and $a[i]$ is the number of accesses to this block by node i. Function $d(i,j)$ gives the Manhattan distance between node i and node j, and c represents the per hop access latency.

Although this equation does not consider the effect of network contention (since c is assumed to be constant)[1] and could not give an accurate average access latency value, it can be used as a first-degree optimization metric for the shared data blocks. Note that, if we can reduce the value of this metric, the real average access latencies will also be decreased most of the time.

Migrating a shared data block to an appropriate position (node) in the storage cache space has the potential of reducing access latencies for shared blocks. This means that if the owner of a data block is changed from j to j', the average access latency for this data block becomes

$$T' = \sum_{i \in \mathcal{N}} c * a[i] * d(i,j') / \sum_{i \in \mathcal{N}} a[i],$$

under the assumption of no contention in the network.

Note that T' might be higher or lower than the original T. Thus, our goal is to find a proper target node j' for this data block so that T' is minimized. Since the constant c and the total number of accesses $\sum_{i \in \mathcal{N}} a[i]$ do not change with different j's, we can remove these two terms and obtain

$$C = \sum_{i \in \mathcal{N}} a[i] * d(i,j').$$

Our optimization goal then becomes one of determining a node j' so that C is minimized. It turns out that this problem is a *post-office placement* problem: there are n input points p_1, p_2, ..., p_n with associated weights w_1, w_2, ..., w_n and we need to find a point p (not necessarily one of the input points) that minimizes the sum $\sum_{i=1}^{n} w_i d(p, p_i)$, where $d(p, p_i)$ is the distance between points p and p_i. According to [16], for the one-dimensional post-office placement problem, that is, the n distinct points are at x_1, x_2, ..., x_n, and their weights are w_1, w_2, ..., w_n, respectively, the optimal point p is the *weighted median* of these n numbers (weights). An important property of the weighted median, x_k, is that x_k satisfies both $\sum_{x_i < x_k} w_i < \sum_{i=1}^{n} w_i/2$ and $\sum_{x_i > x_k} w_i \leq \sum_{i=1}^{n} w_i/2$.

[1] Note, however, that in our experimental evaluation we account for the extra latency incurred as a result of conflicts in the network.

INPUT:
m: the number of nodes per row (X direction)
$a[0 \ldots m-1]$: access array for each node per row
$\mathcal{A}$: total access count
OUTPUT: X coordinate X_d of the migration target node

```
1: A' = A / 2
2: w = 0
3: for i = 0 to m − 1 do
4:    if w < A' and w + f[i] ≥ A' then
5:       X_d = i
6:       break
7:    else
8:       w = w + a[i]
9:    end if
10: end for
```

Figure 5: Algorithm for computing the X coordinate of the target node for a shared data block. The Y coordinate is computed similarly.

Since our architecture is a 2D mesh, our optimization problem can be described as the *two-dimensional post office placement problem.* Let us assume that there are n requesters, from p_1 to p_n, each of which can be represented by its X and Y coordinates, that is, node p_i is represented by (x_i, y_i). Each node p_i has an associated weight w_i, which is equal to the number of access requests issued by it, $a[i]$. Our goal is to find a node $p_{j'}$ that minimizes $C = \sum_{i=1}^{n} a[i] * d(i, j')$. The distance between two nodes i and j' is the Manhattan distance, $d(p_i, p_{j'}) = |x_i - x_{j'}| + |y_i - y_{j'}|$. Therefore, we can express our optimization target as

$$C = \sum_{i=1}^{n} a[i] * |x_i - x_{j'}| + \sum_{i=1}^{n} a[i] * |y_i - y_{j'}|.$$

Both parts on the right-hand side of this equation are non-negative. Thus, we can determine the location of migration target j' by searching the weighted medians at the X and Y directions separately and independently. The two weighted medians, x and y, constitute the coordinates of the final migration target (node). A simple linear-time algorithm given in Figure 5 is used to obtain the weighted median in a dimension, as we know the exact locations of all the nodes in a mesh. Note that, since the search for weighted medians is carried out separately for each dimension (X and Y), the obtained migration target of a data block may *not* necessarily be one of the requesters of that block. For example, if there are three requesters (3,4), (4,3), and (8,8) of a data block and their access frequencies are the same, the algorithm in Figure 5 determines the final target as (4,4), which is *not* one of the three requesters. Although not currently implemented, this approach can be enhanced to include thread priorities as well in deciding the best location.

5 Data Replication

The potential advantage of data replication is that it can further reduce the distance between the requesting node and the node that currently has the requested block. We focus on replicating only *read-only* data blocks. Similar to Section 4.2, for a shared read-only data block with n replicas, its average access latency can be estimated as

$$C = \sum_{k=0}^{n} \sum_{i \in \mathcal{N}_k} a[i] * d(i, j_k).$$

Since this data block has a total of $n + 1$ copies (1 original and n replicas), we say that there are $n+1$ *sharing ranges* (from 0 to n) for this data block. In each sharing range, the cache block has exactly one copy. Requesters in $R_k (0 \leq k \leq n)$ belong to sharing range k and use the copy in this sharing range. Here j_k is the ID of the node where the copy of cache block in sharing range k resides, and i is the ID of the requesting node, which belongs to a sharing range, k. As in Section 4.2, $a[i]$ is the number of requests issued by node i, and function $d(i, j_k)$ returns the Manhattan Distance between requester i and node j_k.

The main issue to be addressed by the replication-based scheme is to decide how to determine sharing ranges by balancing two aspects. On the one hand, the number of sharing ranges cannot be too large, since a large number of replications reduces the effective storage cache capacity and can cause increase in cache miss rates. On the other hand, the sharing range has to be small enough that the requesting processors can quickly access the replica in the sharing range. We might classify the nodes into several sharing ranges statically and force each sharing range maintain at most one replica of a given cache block. The main potential drawback of this static method is its inflexibility. For example, two neighboring requesters can belong to the different sharing ranges if they are boundary nodes between neighboring sharing ranges. In this case, each of them will have a separate replica of the shared block, which is undesirable from the viewpoint of effective cache capacity.

We propose a simple and flexible data replication scheme for the shared storage cache space. First, we define the concept of *replication distance*,[2] which is defined in terms of the number of hops. When a node requests a remote, shared, read-only data block, if the distance to the requested data block is larger than the replication distance, this data block is replicated into the requester's local cache space. This simple replication rule provides a flexible way of determining the sharing ranges by the support of our two-step sequential data block search policy (explained earlier in Section 4). Specifically, once the node knows that

[2]Replication distance is an important parameter whose value can influence the behavior of our approach significantly. In our experiments, we report results with different replication distances.

Table 1: Default simulation parameters.

Network Topology	2D Mesh
Network Size	5×5
Data Block Size	4 KB
Storage Cache Capacity	512 MB/node
Cache Access Latency	0.1 milliseconds (4 KB)
Per Hop Latency	0.1 milliseconds (4 KB)
Disks	IBM 10,000 RPM
Disk Access Latency	8.2 milliseconds (4 KB)
Max Number of Migrations	No limit
Disk Cache	16 MB (multisegmented)
Disk Capacity	40 GB
Counter Threshold (T_m)	400 misses per million cycles
Threshold for Triggering Migrations (T_{tr})	50 accesses

a nonlocal shared data block is close through the sequential search, it will not create a local replica. The sharing pattern and the replication distance together determine the sharing ranges for a data block. Note that under this scheme the different data blocks are likely to have different sharing ranges. Initially, we start with all the data blocks marked as read-only. When a write is issued for a block for the first time, we simply invalidate all the replicas (except the one for which the write is issued) and maintain a single copy of that shared block using a write-invalidate protocol [2]. The read-only status of that block changes to the read-write status, and thus no more replicas are created for it. Apart from the cache coherence protocol for handling write operations, the overhead incurred by replicating shared cache blocks is negligible.

6 Experiments

We discuss in this section the benchmark experiments we performed and the results we obtained.

6.1 Setup

To evaluate the effectiveness of our proposed topology aware storage caching scheme, we implemented it using the SIMICS toolset [13]. SIMICS is a full-system simulation platform, capable of simulating parallel systems with various system configurations. We also implemented and tested using SIMICS three other storage caching schemes against which our approach is compared. Table 1 lists the major simulation parameters and their default values. The latency values used are similar to those reported in [1].

We used six benchmark programs in this study: ammp [8], fma3d [8], wupwise [8], oltp [7], specjbb [17], and specweb [18]. The total size of the disk-resident data manipulated by these applications varies between 9.3 GB and 21.8 GB. Note that ammp, fma3d, and wupwise are the out-of-core versions of the corresponding SPEComp benchmarks [3]. These out-of-core versions have been coded carefully to optimize I/O as much as possible. For example, where applicable, we used collective I/O [19] to reduce the number of I/O requests issued at the application level.

For each application code in our experimental suite, we conducted experiments with four different versions: *First-and-Forever* (FAF): In this caching scheme, when a data block is brought from disk to a storage cache, it remains there as long as there is space. In other words, it is never migrated to another storage cache. When it is displaced from the cache, it is written back to disk if it has been modified while residing in the cache. Many conventional storage caching schemes work this way in practice. *Always-Migrate* (AMG): In this approach, when a data block is requested by a node, the block is always migrated to the storage cache of that node. Note that this is an aggressive data migration strategy and can lead to lots of data movements in the shared cache space. *One-Step-Migration* (OSM): In this scheme, each time a nonlocal data is requested, it moves one hop closer to the requester (either in the X or Y direction). Note that under this scheme in the stable state, we can expect a shared data block to find its ideal position, though it may take some time to do so. *Post-Office-Placement* (POP): This is our post-office placement-based storage caching solution presented in Section 4. It performs both victim-related migrations and sharing-related migrations.

We emphasize the common characteristics of these four storage caching schemes. First, as far as the space management of a local cache (attached to a node) is concerned, all these schemes use LRU to select the victim block. However, once the victim is selected, in all schemes except ours, the victim is migrated to the place (a storage cache or disk) from where the requested data block is coming. That is, the locations of the requested and victim data blocks are interchanged. In our scheme, however, the default policy is the counter-based one, which is explained in Section 4. Also, in all four schemes, the requested data from disk is first placed into the storage cache of the requester node.

6.2 Evaluation

Our first set of results is presented in Figure 6 and shows the average storage cache hit (access) latencies under different caching schemes. In this graph, all bars are *normalized* with respect to the FAF scheme. These results clearly show that the proposed POP scheme generates better results than all the other schemes tested. Specifically, the percentage cache access latency improvements brought by the AMG, OSM, and POP schemes (over the FAF scheme) are -10.4%, 11.7% and 29.1%, respectively. To understand these results better, we also collected distance-to-data statistics. The graph in Figure 7 gives, for each of the four caching schemes, the average number of hops traversed to reach the requested data block. In other words, these results capture the average distance between the node that tries to access the data and the node that holds the requested data in its storage cache. We see from these results

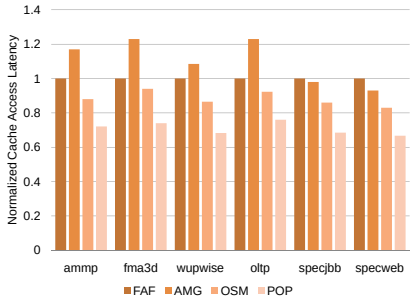

Figure 6: Average storage cache hit (access) latencies under the different caching schemes.

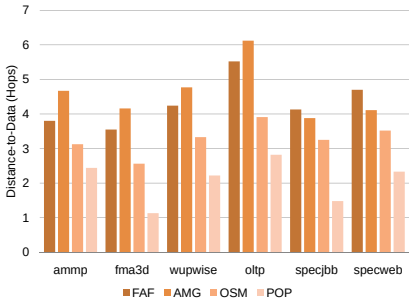

Figure 7: The average number of hops traversed to reach the requested data block.

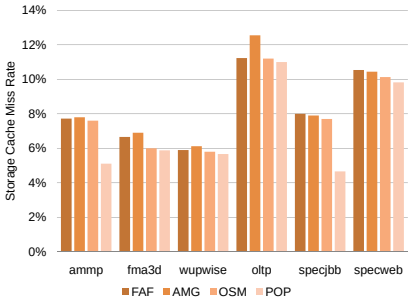

Figure 8: Storage cache miss rates for the different caching schemes.

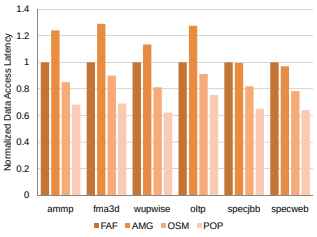

Figure 9: Average data access latencies, which include the impact of cache misses as well (all bars in this graph are normalized with respect to FAF).

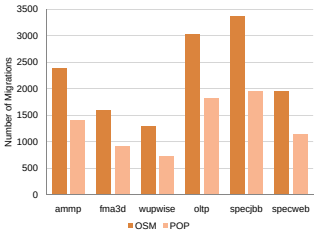

Figure 10: Number of times data blocks are migrated from one storage cache to another under our scheme and OSM.

that the average number of hops is 4.32, 4.61, 3.28, and 2.07 for FAF, AMG, OSM, and POP, respectively, which explains why our proposed caching scheme generates better hit latencies than the other schemes tested.

Note that hit latency is only one part of the picture. We are also interested in average data access latency, which includes cache misses as well. Figure 8 gives the storage cache miss rates[3] for the different schemes, and Figure 9 shows the average data access latencies, which include the impact of cache misses as well (all bars in this graph are *normalized* with respect to FAF). We can observe from Figure 8 that our approach generates better miss rates than the others, thanks to the careful selection of the node to which the victim block is sent. Specifically, our scheme tries to keep the victim block in the shared storage cache space as long as it is beneficial to do so, and this leads to an improvement in miss rates. As a result, the overall data access latency results in Figure 9 are better than the cache hit latency results presented in Figure 6. Overall, the average improvements brought by the OSM and POP schemes (over the FAF scheme) are 15.3% and 32.6%, respectively.

[3]Note that a cache miss occurs when the requested data block is not in any of the storage caches, and a disk access needs to be made.

Figure 10 presents, for our scheme and OSM, the number of times data is migrated from one storage cache to another during execution. Clearly, our storage caching scheme reduces the number of migrations experienced by the OSM scheme by about 19.6% on average. The reason is that in OSM, data is brought closer to its customers one hop at a time, resulting in a lot of reads and writes in the storage cache. In comparison, our approach tries to migrate the data to its ideal position quickly, and this helps to cut the number of migrations significantly. While not quantified in this paper, this reduction in migration counts can also help reduce the power consumption in the storage cache space.

7 Related Work

Shared cache management requires meeting the demands of competing services for space in the shared cache, while minimizing their interference with each other. Cache interference can lead to significant performance degradation and reduce overall system throughput. In order to alleviate this problem, the shared cache is partitioned such that, each application is allocated a portion of the cache buffers. Recently, many research groups have explored shared cache partitioning designs that attempt to avoid conflicts among multiple applications [25] [22] [23]. In Argon [22], the cache partitioning algorithm uses a simulator to predict the cache absorption rate with hypothetical cache sizes.

Managing shared cache among multiple concurrently executing applications requires minimizing the possibility of destructive interferences caused by their interaction with each other. Prior research has shown that lack of efficient shared cache management schemes can degrade cache performance significantly and lead to unpredictable system performance [4]. While single-server storage cache management has been studied well in the literature [14], there remains a need to understand and optimize caching when the storage caches are shared by multiple, simultaneously executing applications. Cooperative caching [5] seeks to

improve the network file system performance by coordinating the contents of client caches and allowing requests not satisfied by a client's local in-memory file cache to be satisfied by the cache of another client. Systems employing this technique include GMS [6] and PGMS [21]. Many techniques have been proposed for improving the management of second-tier server caches. Li et al. [12] use write hints to improve the performance of second-tier cache replacement policies. Moreover, storage cache management bears similarities to shared on-chip cache management in the chip multiprocessor domain. Several dynamic L2 cache partitioning algorithms have been proposed in hardware [9]. They focus their work on CMP/SMT systems and try to control the cache space and memory bandwidth. These papers strive to meet the QoS demands of applications by maintaining some hardware counters. Qureshi and Patt [15] partition the cache among multiple concurrently executing applications such that more cache is provided to an application if it helps reduce the number of misses without considering any QoS. Kandemir et al. [11] approximate a post-office placement based solution in hardware and apply it to nonuniform L2 architectures in chip multiprocessors. In comparison, our approach targets storage caches, is implemented in software (Linux), uses a different algorithm, and includes a data replication module.

8 Conclusion

In a parallel system with multiple storage caches, effective management of these caches is crucial, especially when they form a shared storage cache space. As the data access and sharing patterns change during execution, migrating data in the shared cache space can improve access latencies by reducing the distance between a data block and its customers. In this paper, we propose for such architecture, a topology-aware storage caching scheme that is based on the two-dimensional post-office placement problem. Our approach determines the most suitable location (storage cache) of data block shared by multiple nodes. Our experimental results with six applications codes show that our approach can improve the average hit latency, average miss rate, and average data access latency by 29.1%, 7.0%, and 32.7%, respectively. We also show that our approach can be extended to employ replication in the shared storage cache space.

References

[1] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L Torus Interconnection Network. *IBM Journal of Research and Development*, 49(2-3):265–276, 2005.

[2] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[3] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *WOMPAT*, pages 1–10, 2001.

[4] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *OSDI*, pages 165–177, 1994.

[5] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *OSDI*, pages 267–280, November 1994.

[6] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *SOSP*, pages 201–212, 1995.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, October 1992.

[8] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[9] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS*, pages 25–36, 2007.

[10] S. Jiang, F. Petrini, X. Ding, and X. Zhang. A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance. In *ICDCS*, page 42, 2006.

[11] M. T. Kandemir, F. Li, M. J. Irwin, and S. W. Son. A Novel Migration-Based NUCA Design for Chip Multiprocessors. In *SC*, page 28, 2008.

[12] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-Tier Cache Management Using Write Hints. In *FAST*, pages 115–128, 2005.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[14] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, pages 115–130, 2003.

[15] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO*, pages 423–432, 2006.

[16] R. L. Rivest and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., 1990.

[17] Standard Performance Evaluation Corporation. Specjbb 2000 Java business benchmark. `http://www.spec.org/osg/jbb2000/`, 1998.

[18] Standard Performance Evaluation Corporation. SPECweb2005. `http://www.spec.org/web2005/`, 2005.

[19] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. In *Scientific Programming*, pages 301–317, 1996.

[20] M. Vilayannur, A. Sivasubramaniam, M. Kandemir, R. Thakur, and R. Ross. Discretionary Caching for I/O on Clusters. *Cluster Computing*, 9(1):29–44, 2006.

[21] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin, and H. M. Levy. Implementing Cooperative Prefetching and Caching in a Globally-Managed Memory System. In *SIGMETRICS*, pages 33–43, 1998.

[22] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST*, pages 61–76, 2007.

[23] Y. Wang and A. Merchant. Proportional-Share Scheduling for Distributed Storage Systems. In *FAST*, 2007.

[24] M. G. Watson and J. K. Flanagan. Does Halting Make Trace Collection Inaccurate? A Case Study Using Pentium 4 Performance Counters and SPEC2000. In *IISWC*, 2004.

[25] G. Yadgar, M. Factor, K. Li, and A. Schuster. MC2: Multiple Clients on a Multilevel Cache. In *ICDCS*, pages 722–730, 2008.