

Computation Mapping for Multi-Level Storage Cache Hierarchies

Mahmut Kandemir¹ Sai Prashanth Muralidhara¹ Mustafa Karakoy² Seung Woo Son³
¹Pennsylvania State University ²Imperial College ³Argonne National Laboratory
{kandemir,smuralid}@cse.psu.edu mtk2@psu.edu sson@mcs.anl.gov

ABSTRACT

Improving I/O performance is an important issue for many data intensive, large scale parallel applications. While storage caches has been one of the ways of improving I/O latencies of parallel applications, most of the prior work on storage caches focus on the management and partitioning of cache space. The compiler's role in taking advantage of, in particular, multi-level storage caches, has been largely unexplored. The main contribution of this paper is a shared storage cache aware loop iteration distribution (iteration-to-processor mapping) scheme for I/O intensive applications that manipulate disk-resident data sets. The proposed scheme is compiler directed and can be tuned to target any multi-level storage cache hierarchy. At the core of our scheme lies an *iterative strategy* that clusters loop iterations based on the underlying storage cache hierarchy and how these different storage caches in the hierarchy are shared by different processors. We tested this mapping scheme using a set of eight I/O intensive application programs and collected experimental data. The results collected so far are very promising and show that our proposed scheme 1) is able to improve the I/O performance of original applications by 26.3% on average, and this leads to an average of 18.9% reduction in overall execution latencies of these applications, and 2) performs significantly better than a state-of-the-art (but storage cache hierarchy agnostic) data locality optimization scheme. We also present an enhancement to our baseline implementation that performs local scheduling once the loop iteration distribution is performed.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, L^AT_EX, text tagging

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HPDC '10 Chicago, Illinois USA

Copyright 2010 ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

1. INTRODUCTION

As system sizes and capabilities approach petascale range [31, 40], opportunities to solve or make significant strides in scales of scientific and engineering problems that were unimaginable a few years ago now exist. It is widely accepted that petascale systems (and beyond) will have architectures where nodes are multi-core processors (few cores to hundreds of cores) connected via very high-speed interconnects. Just because potential for raw performance exists however, does not necessarily mean these systems can be easily exploited, due to the complexities facing a typical application programmer or user. One of the aspects that is often not as well developed or ignored in the context of utilizing large scale parallel systems that target parallel scientific and engineering applications is *parallel I/O and storage management*.

Parallel scientific applications, such as INCITE projects [24] including the community climate system model (CCSM at NCAR [12]), analyzing cosmic microwave background radiation (CMBR at LBNL [13]), and studying astrophysical thermonuclear flashes (FLASH at ANL [16]), have significant and growing I/O needs, demanding high performance I/O. Caching I/O blocks in memory (called *storage caching*) is one effective way of alleviating disk latencies in such applications, and there can be multiple levels of caching on a parallel system. Previous studies [4, 14, 39, 49] have shown the benefits of storage caching – whether it be local to a particular node, or a shared global cache across multiple nodes.

Most current storage systems used by high-end computing platforms are typically organized as a cluster of client (compute) nodes connected to a cluster of I/O (server) nodes over a network. Such a system can have different types of storage caches. For example, each client can have a private cache or multiple clients can share the same client-level cache implemented either in one of the clients entirely or in a distributed fashion over multiple clients (e.g., as in the case of cooperative caching [14]). Similarly, each server can accommodate a cache, which may be accessed by two or more clients. More recent high-end systems accommodate even deeper storage hierarchies. For example, IBM Blue Gene/P [35] is organized as a cluster of *compute nodes* that execute application threads, *storage nodes* that are connected to disks, and *I/O nodes* which transfer data between compute nodes and storage nodes. In such an architecture, each layer (compute, I/O, storage nodes) can accommodate its own cache, resulting in a three-layer storage cache hierarchy. Clearly, efficient management of such multi-level storage cache hierarchies is a challenging problem, especially in the context of I/O intensive scientific applications that perform frequent disk reads (e.g., for visualization) and writes (e.g., for checkpointing).

Many I/O intensive scientific applications are structured as a series of nested loops operating on disk resident data sets. Execution of a loop nest in a parallel computing platform requires two steps:

loop parallelization and *iteration-to-processor mapping*. The former deals with deciding the set of loops (in the nest) to execute in parallel and applies (if necessary) several code transformations to enable more parallelism. The latter on the other hand decides, for each loop iteration, the processor on which to execute that iteration. For data intensive applications that manipulate large, multi-dimensional arrays, prior compiler research studied numerous loop parallelization strategies (see [41, 2] and the references therein). The mapping problem on the other hand is very interesting in the context of I/O intensive applications that execute on parallel architectures with shared storage caches. This is because the way in which loop iterations are assigned to processors (client nodes) can have tremendous impact on storage cache behavior, which in turn can influence overall program behavior dramatically.

The main contribution of this paper is a shared storage cache aware loop iteration distribution (iteration-to-processor mapping/assignment) scheme for I/O intensive applications that manipulate disk-resident data sets. The proposed scheme is compiler directed and can be tuned to target any multi-level storage cache hierarchy. At the core of our scheme lies an *iterative strategy*, that clusters loop iterations based on the underlying storage cache hierarchy and how the client nodes share the different storage caches present in the hierarchy. We tested this mapping scheme using a set of eight I/O intensive application programs and collected experimental data. The results collected so far are very promising and show that our proposed scheme 1) is able to improve the I/O performance of original applications by 26.3% on average, and this leads to an average of 18.9% reduction in overall execution latencies of these applications, and 2) performs significantly better than a state-of-the-art (but storage cache hierarchy agnostic) data locality optimization scheme. To our knowledge, this is the first work that performs fully-automated, storage cache hierarchy aware iteration-to-processor mapping.

The remainder of this paper is organized as follows. The next section presents the background on data representation and loop parallelization. Section 3 introduces the problem of iteration-to-processor mapping in a multi-level storage cache hierarchy. Section 4 discusses the technical details of our proposed approach, and Section 5 presents the results collected from our experimental evaluation. Section 6 discusses the related work, and finally, Section 7 concludes the paper by summarizing our main contributions and discussing the possible future extensions briefly.

2. BACKGROUND ON DATA REPRESENTATION AND LOOP PARALLELIZATION

Let i_1, i_2, \dots, i_n denote the iterators of the n loops in a loop nest. We define an *iteration* formally as $\vec{i} = (i'_1 i'_2 \dots i'_n)^T$, where $L_k \leq i'_k \leq U_k$ holds. In this last expression, i'_k is a particular value that loop iterator i_k takes, and L_k and U_k are the lower and upper bounds, respectively, for i_k . An array reference within such a loop nest can be represented in a linear algebraic form using $\mathcal{R}(\vec{i}) = Q\vec{i} + \vec{q}$, where Q is termed as the access matrix and \vec{q} is referred to as the offset vector. For example, for array reference $A[i_1 + 3, i_2 - 1]$, Q is two-by-two identity matrix and \vec{q} is $(3 \quad -1)^T$.

In the context of this paper, parallelizing a loop means running its iterations in parallel. While it is possible to parallelize a loop whose iterations depend on each other, commercial optimizing compilers normally parallelize a loop only if there is no data dependence across its iterations. This is because parallelizing a loop that contains dependences typically requires explicit synchronizations across its iterations (to guarantee correctness), which may be costly at runtime.

In a loop nest with n loops, any of these n loops can be parallelized (the ones with dependences require synchronization as stated above). We note that, when one or more loops of a nest is parallelized, the iterations of that nest will be distributed across available processors for parallel execution. This distribution is called the *iteration-to-processor mapping* or *iteration-to-processor assignment*. As will be demonstrated in this paper, as far as the underlying storage cache hierarchy is concerned, different mappings/assignment can lead to dramatically different I/O performances. Our goal in this work is to derive a mapping that maximizes the performance of a multi-level storage cache hierarchy.

3. MULTI-LEVEL STORAGE CACHE HIERARCHY AND MAPPING PROBLEM

In mapping a loop-based data-intensive application to a parallel computing platform, there are two complementary steps: *loop parallelization* and *iteration-to-processor assignment*. In the parallelization step, the user/compiler decides the set of loop iterations that will be executed in parallel. We note that it is not necessary that all the iterations scheduled for parallel execution to be free of cross-iteration dependences. In some cases, the user/compiler can decide to execute the iterations with cross-loop dependences in parallel and enforce correctness of semantics through explicit synchronization. In this work, we do not make any new contribution to the parallelization step. We assume that either (i) the target loop nest has already been parallelized by hand or using a parallelizing compiler, or (ii) the target code is sequential. In the latter case, we apply a default parallelization strategy which first places all data dependences into inner loop positions (to minimize synchronization costs) and then parallelizes the outermost loop that that does not carry any data dependence.

The main contribution of this paper is a novel iteration-to-processor assignment scheme. The novel aspect of this scheme is its cache hierarchy awareness. In other words, in assigning the set of loop iterations (to be executed in parallel) to available processors, our scheme considers the storage cache hierarchy of the underlying architecture. As such, our approach can work with any loop parallelization strategy (that is, we do not care how the loops have been parallelized) and choice of parallelization strategy is actually orthogonal to the main focus of this paper. The input to our approach is the set of loop iterations to be executed in parallel and our proposed scheme assigns those iterations to processors.

It is also important to observe that if a loop is explicitly parallelized, iteration assignment/mapping may or may not have already been performed (depending on the parallelization style/library employed). If it is the former, our proposed scheme performs a re-assignment (re-mapping).

The execution model we have is that the loop iterations to be executed in parallel are distributed across available client nodes. That is, each client node is assigned a set of iterations to execute. Figure 1 illustrates the sketch of an example storage hierarchy with three levels: *compute nodes*, *I/O nodes*, and *storage nodes*. This type of layered architectures have become very common in recent high-end parallel computing architectures such as the IBM Blue Gene/P machine at the Argonne National Laboratory [36, 35] and the Cray XT system at the Oak Ridge National Laboratory [25]. Such systems typically exhibit increased architectural complexity with tens of thousands processors. While the computational power of such systems keeps increasing with every generation, the same is not true for their I/O subsystems. Therefore, as in the I/O forwarding layer in Blue Gene/P [1], introducing another layer in the I/O stack (in form of I/O nodes) is one way of maintaining scalability in

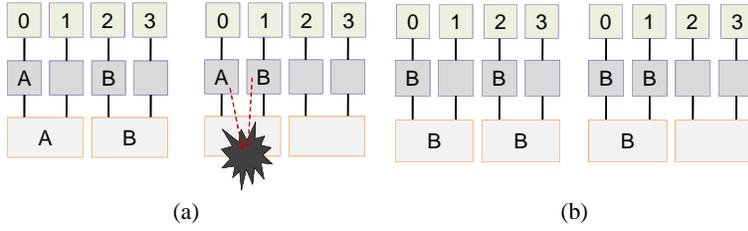


Figure 2: Two scenarios illustrating what happens when our two rules are and are not followed. A and B denote two different data chunks. The client nodes are numbered from 1 to 4.

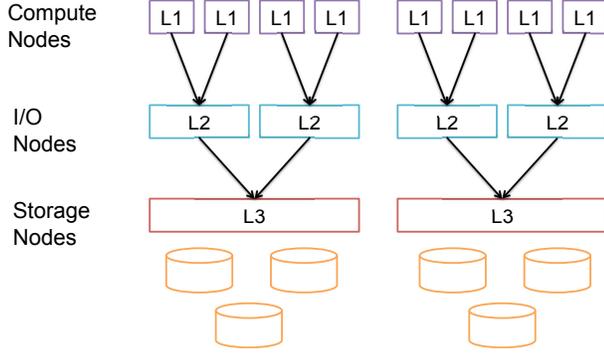


Figure 1: Three-level storage cache hierarchy. Each rectangle represents a storage cache and the arrows capture the structure of the hierarchy. L1, L2 and L3 refer to the storage caches at the compute node layer, I/O node layer and the storage node layer, respectively.

such systems. In Blue Gene/P, for example, compute nodes are partitioned into subsets and each subset is mapped to an I/O node. The ratio of I/O nodes to compute nodes can be varied from 1:8 to 1:64. And, the I/O nodes are connected to the file system server through a 10GigE network. Therefore, depending on the configuration, each I/O node handles accesses from different compute nodes, and the same is true for each storage node. While not specifically focusing on the systems like IBM Blue Gene/P or Cray XT, multi-level caching has also been studied in several prior studies [51, 49, 18]. The main motivation is that the caches in modern storage systems often form a multilevel hierarchy, and simple approaches to maximize cache hits on a particular layer in such a hierarchy will not necessarily improve the overall system performance. Therefore, one needs to explore hierarchy aware cache management schemes [49, 18].

In Figure 1, the rectangles represent storage caches.¹ For illustrative purposes, we assume eight compute nodes (also referred to as “processors” and “client nodes” in this paper), each having a private storage cache (denoted using L1 in the figure). We further assume that each of the four I/O nodes maintains a storage cache (denoted using L2), shared by a pair of compute nodes, as indicated by the arrows. Finally, each of the two storage nodes maintains a cache (denoted using L3), shared by two I/O nodes. Consequently, each of the L1, L2 and L3 storage caches are shared by 1 (private), 2 and 4 client nodes, respectively. We first make the following definition:

Two client nodes are said to have “affinity at storage cache L_i ” if both have access to it.

As far as iteration-to-processor mapping is concerned, this stor-

¹We use the term “storage cache” even for the caches that are attached to compute and I/O nodes.

```
float A[1..N1,1..N2,1..N3];
...
for i1 = 2 to N1
  for i2 = 1 to N2
    for i3 = 1 to N3-1
      ... A[i1-1, i2, i3+1] ...
```

Figure 3: A sample code fragment.

age cache hierarchy implies two rules to follow:

- If two iterations do not share any data element, they should not be mapped to two client nodes that have affinity at some storage cache.
- If two iterations do share data on the other hand, it is better to map them to clients that have affinity at some storage cache.

We note that, if the first rule is not followed, this will typically reduce the shared cache utilization, as the total amount of data that compete for the same cache is typically increased in this case. In contrast, if the rule is followed, pressure on caches can reduce and this in turn helps to improve overall application performance. If the second rule above is not followed, this increases data replication across different caches, reducing the effective cumulative storage cache capacity. Figure 2 illustrates what happens when the two rules discussed above are and are not followed. In (a), two iterations access different data chunks (blocks), that is, no sharing. In this case, it is better that the clients that execute these iterations do not have an access to a common storage cache. The left part of the figure shows this case. The right part on the other hand illustrates potential conflict (competition for the same space) when these iterations are assigned to two clients that share a cache. In (b) on the other hand, two iterations do share a data chunk. Assigning these iterations to clients that do not share a storage cache may increase data replication across the storage cache hierarchy (as depicted on the left portion of the figure), thereby reducing effective cache capacity. In comparison, if these iterations are assigned to the clients that share a storage cache, we may be able exploit data block reuse (that is, convert data reuse into data locality [cache hit]) across the accesses coming from these clients.

4. TECHNICAL DETAILS OF OUR PROPOSED SCHEME

4.1 Polyhedral Model

We use a *polyhedral model*² to represent loops, arrays and references to arrays within loop bodies. For example, the loop nest shown in Figure 3 can be expressed within the polyhedral model as follows:

$$\mathcal{G} = \{(i_1, i_2, i_3) \mid 2 \leq i_1 \leq N_1 \ \&\& \ 1 \leq i_2 \leq N_2 \ \&\& \ 1 \leq i_3 \leq N_3 - 1\},$$

where $\&\&$ denotes the “and” operator. In a similar fashion, we can represent the array declaration in the same code fragment as:

$$\mathcal{H} = \{(k_1, k_2, k_3) \mid 1 \leq k_1 \leq N_1 \ \&\& \ 1 \leq k_2 \leq N_2 \ \&\& \ 1 \leq k_3 \leq N_3\}.$$

²In a polyhedral model, objects of interests are represented as integer valued points in various regions of different spaces and the mappings are used to connect these spaces.

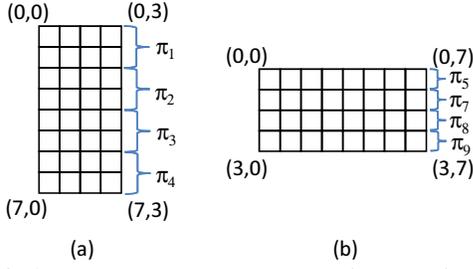


Figure 4: A sample data space that contains two disk resident arrays and its partitioning into data chunks.

Finally, the array reference shown in the code can be expressed as:

$$\begin{aligned} \mathcal{L} &: \mathcal{G} \longrightarrow \mathcal{H} \\ &= \{(i_1, i_2, i_3) \rightarrow (k_1, k_2, k_3) \mid (i_1, i_2, i_3) \in \mathcal{G} \ \&\& \\ &\quad (k_1, k_2, k_3) \in \mathcal{H} \ \&\& \\ &\quad k_1 = i_1 - 1 \ \&\& \ k_2 = i_2 \ \&\& \ k_3 = i_3 + 1\}. \end{aligned}$$

While there are many publicly-available tools and libraries that can be used to manipulate polyhedral sets and its choice is really orthogonal to the main focus of our approach, in this work we use the Omega Library [42]. In the rest of this paper, we use symbol σ to denote a loop iteration. Recall that, in a nest with n loops (also called n -deep nest), an iteration is a vector with n entries, where the first entry represents the value of the iterator of the outermost loop and the last entry captures the value of the iterator of the innermost loop. Also, as stated earlier, we use the terms “client node”, “compute node” and “processor” interchangeably.

4.2 Data Sharing across Loop Iterations and Tags

Two iterations σ_1 and σ_2 of a loop nest share data if there are two references \mathcal{R}_1 and \mathcal{R}_2 in the loop body (not necessarily distinct from each other) such that $\mathcal{R}_1(\sigma_1) = \mathcal{R}_2(\sigma_2)$. We can extend this definition of data sharing to a *data block (chunk) level* as follows. Let us divide an entire data space³ into r equal-sized *chunks*, and label them as follows: $\pi_1, \pi_2, \dots, \pi_r$. We then assign a *tag* to each iteration as follows. We assign an r -bit tag $\Lambda_j = \lambda_0 \lambda_1 \dots \lambda_{r-1}$ to iteration σ_i where, $\lambda_k = 1$ if σ_i accesses data chunk π_k ; otherwise, λ_k is set to 0 (where $0 \leq k \leq r-1$). As an example, a tag such as 0011 indicates that the corresponding iteration accesses the last two data chunks but does not access the first two data chunks ($r = 4$).

The important point is that the tag associated with an iteration describes its *data access pattern at a chunk level* (though it does not indicate which particular elements in a chunk are accessed). We then define an *iteration chunk* as the set of iterations with the *same tag*. This means that all the iterations in an iteration chunk have the same “data chunk access pattern.” We use symbol γ^Λ to denote the iteration chunk γ with tag Λ . The significance of the iteration chunk concept in our work is two-fold. First, all iterations that belong to the same iteration chunk are executed successively when that chunk is scheduled, thereby exploiting data reuse. Second, the tags of iteration chunks provide some sort of measure for “similarity”. In particular, if the tags of the two different iteration

³What we mean “data space” in this context is the set of data elements of all disk-resident arrays combined. The arrays can be ordered arbitrarily and each can be divided in equal-sized chunks. We require that no chunk is shared across arrays. In other words, each array is partitioned separately. But, in numbering chunks, we just increment the chunk label by one as we move from the last chunk of the array t to the first chunk of array $t + 1$, as illustrated in Figure 4.

chunks do not have any common bit between them (that is, zero Hamming Distance), this means that these two iteration chunks do not share any data. Consequently, they should not be assigned to two client nodes that have access to a common storage cache (that is, have affinity at some storage cache). Conversely, if the tags of two iteration chunks are similar (e.g., large number of “1”s in the same bit positions), we can exploit data locality if they are assigned to two client nodes that have access to a common storage cache. The algorithm given in the next subsection exploits this observation. Before going into the discussion of our algorithm though, we want to mention that iteration chunks can be obtained from data chunks using the polyhedral model as follows.

Let us focus, without loss of generality, on γ^Λ , where $\Lambda = \lambda_0 \lambda_1 \dots \lambda_{i-1} \lambda_i \lambda_{i+1} \dots \lambda_{r-2} \lambda_{r-1} = 11 \dots 100 \dots 00$, that is, the first i bits are 1 and the rest are 0. Then, assuming that there are R references in the loop body ($\mathcal{R}_0, \dots, \mathcal{R}_{R-1}$), we can express γ^Λ as follows:

$$\begin{aligned} \gamma^\Lambda &= \{\bar{I} \mid \forall q, 0 \leq q \leq (i-1) [\exists 0 \leq s \leq R-1 \text{ s. t. } \mathcal{R}_s(\bar{I}) \in \pi_q] \\ &\quad \text{and } \neg \exists q', i \leq q' \leq (r-1) [\exists 0 \leq s \leq R-1 \text{ s. t. } \mathcal{R}_s(\bar{I}) \in \pi_{q'}]\}. \end{aligned}$$

The first line of this expression captures the iterations that access *all* data chunks π_q where $0 \leq q \leq (i-1)$, while the second line indicates that *none* of these iterations access any data chunk $\pi_{q'}$ where $i \leq q' \leq (r-1)$. In our approach, the iteration distribution across the client nodes (processors) is carried out at an iteration chunk granularity. That is, each client node is assigned a number of iteration chunks. Therefore, once we determine the set of iteration chunks assigned to a client node, we need to generate code that enumerates the iterations in those chunks. We note that, for a given γ^Λ , the Omega Library can be used for generating code for it. Specifically, the *codegen(.)* utility provided by the Omega Library helps us to generate the code (typically in form of a set of nested loops) that enumerates the iterations in γ^Λ (and we repeat this for all iteration chunks based on the scheduling determined for the client node).

4.3 Loop Distribution Algorithm

We now discuss the details of our proposed loop distribution algorithm. Figure 5 shows the different steps of this algorithm. Inputs to this algorithm include the set of iterations to be distributed across the client nodes and the storage cache topology. The output of the algorithm is the iteration chunks/clusters to be scheduled on each client node such that the performance of the underlying storage cache hierarchy is maximized. There are three main steps involved in our algorithm, namely, *initialization*, *clustering*, and *load balancing*. We explain each of these steps in detail below. We note that our approach operates at a loop nest granularity. In other words, we handle each loop nest (which may have any number of loops in it) in isolation, though in principle it can be extended to optimize neighboring nests together.

Initialization: The first step starts out by grouping the loop iterations into iteration chunks. This grouping is done based on the similarity of their tags. Recall that the different data chunks accessed by an iteration are captured by the tag of that iteration. The second part of the initialization step involves building a *graph* whose nodes are the iteration chunks and the edge weight between any two nodes indicate the degree of data sharing between these iteration chunks. More specifically, the weight of an edge between any two nodes of the graph is the number of common “1”s between the tags of the two nodes (iteration chunks).

Clustering: The goal in this step is to cluster the iteration

Input :
I/O System Description, $\mathcal{A} = \{T, k\}$
 T is the storage cache hierarchy tree with the storage node as the root node and k is the number of compute nodes
Loop Iteration Set, $\mathcal{I} = \{\sigma_0, \sigma_1, \dots, \sigma_m\}$
 \mathcal{I} is the set of all iterations in the loop nest
Data chunk Set, $\mathcal{D} = \{\pi_0, \pi_1, \dots, \pi_{r-1}\}$
 \mathcal{D} is the set of all equal sized data element chunks accessed by the loop nest
 $BThres$ = Maximum tolerable imbalance in iteration counts

Output :
Iteration Chunk Set, $\mathcal{C} = \{c_0, c_1, \dots, c_k\}$
 $c_i = \{\sigma_j, \sigma_l, \dots, \sigma_m\}$, k is the number of compute nodes

Algorithm

Initialization :
Initialize tags:
Assign a tag $\Lambda_j = \lambda_0 \lambda_1 \dots \lambda_{r-1}$ to iteration σ_i
where, $\lambda_k = 1$ if σ_i accesses data chunk π_k
iteration chunk $\gamma^\Lambda = \{\sigma_k, \text{ such that } \sigma_k \text{ has tag } \Lambda\}$
Size of the iteration chunk, $\gamma^\Lambda, S(\gamma^\Lambda) = |\gamma^\Lambda|$

Build Graph:
Build a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$,
 $\mathcal{V} = \{\gamma^{\Lambda_1}, \gamma^{\Lambda_2}, \dots, \gamma^{\Lambda_{2^r}}\}$
 $\mathcal{E} = \{(\gamma^{\Lambda_i}, \gamma^{\Lambda_j}) \text{ such that } \omega(\gamma^{\Lambda_i}, \gamma^{\Lambda_j}) = \text{number of "1" bits in } \Lambda_i \wedge \Lambda_j\}$

Hierarchical Iteration Distribution:
 $HLevel$ = root of the storage cache hierarchy tree, T
 $NumClusters$ = degree of nodes at level " $HLevel$ "
Cluster Set, $\mathcal{C} = \{cs\}$, $cs = \{c\} \forall c \in \mathcal{V}$
While $HLevel \neq \text{leaf level}$:
New Cluster Set, $\mathcal{NC} = \{\}$
For Each cluster $cs_i \in \mathcal{C}$:
 $\mathcal{C} = \mathcal{C} - cs_i$
 $TotalIterations$ = total number of iterations in cs_i

Stage 1 (Clustering):
While($|cs_i| > NumClusters$):
For two clusters $c^{\alpha_p}, c^{\alpha_q} \in cs_i$,
 $c^{\alpha_p} = \{\gamma^{\Lambda_a}, \gamma^{\Lambda_b}, \dots, \gamma^{\Lambda_c}\}$
 $\alpha_p = BitwiseSum(\Lambda_a, \Lambda_b, \dots, \Lambda_c)$
 $S(c^{\alpha_p}) = |\gamma^{\Lambda_a}| + |\gamma^{\Lambda_b}| + \dots + |\gamma^{\Lambda_c}|$
/* $S(c^{\alpha_p})$ is the total no. of iterations in c^{α_p} */
Merge $c^{\alpha_p}, c^{\alpha_q}$ in to a new cluster $c^{\alpha_{new}}$ such that, $\alpha_p \bullet \alpha_q$ (dot product) is maximized
 $\therefore c^{\alpha_{new}} = c^{\alpha_p} \cup c^{\alpha_q}$

If ($|cs_i| < NumClusters$):
/*Case when the current number of clusters is less than the required number of clusters at this level*/
While($|cs_i| \neq NumClusters$):
Select $c^{\alpha_q} \in cs_i$, such that $S(c^{\alpha_p})$ is max
Break c^{α_q} into two clusters

Stage 2 (Load balancing):
After clustering, $cs_i = \{c^{\alpha_1} \dots c^{\alpha_{NumClusters}}\}$
/*Use greedy approach to balance cluster sizes*/
 $ULim = \frac{TotalIterations}{NumClusters} + BThres$
 $LLim = \frac{TotalIterations}{NumClusters} - BThres$
While $\exists c^{\alpha_p} \in cs_i$, such that $S(c^{\alpha_p}) > ULim$:
Select $c^{\alpha_q} \in cs_i$ such that,
 $S(c^{\alpha_q}) < LowLimit$
Evict some γ^{Λ_a} from c^{α_p} to c^{α_q} such that,
 $LLim < S(c^{\alpha_p}) < ULim$
 $LLim < S(c^{\alpha_q}) < ULim$
and, $\Lambda_a \bullet \alpha_q$ is maximum
If no such node exists, split γ^{Λ_a} such that
 $S(c^{\alpha_p})$ and $S(c^{\alpha_q})$ are within limits and evict as described above

For Each $c^{\alpha_p} \in cs_i$:
 $\mathcal{NC} = \mathcal{NC} + \{\{\gamma^{\Lambda_a}\} \forall \gamma^{\Lambda_a} \in c^{\alpha_p}\}$
 $\mathcal{C} = \mathcal{NC}$
 $HLevel = HLevel + 1$,
Update $NumClusters$ to the degree of nodes at " $HLevel$ "
After $h = \log_2 k$ hierarchical levels, $\mathcal{C} = \{c_0, c_1, \dots, c_k\}$
where, k is the number of compute nodes

Return \mathcal{C}

Figure 5: Cache hierarchy-conscious loop iteration distribution algorithm.

chunks in accordance with the underlying target storage cache hierarchy. This way, clustering is customized to a given target storage

cache hierarchy, and as a result, our approach can work with any storage cache hierarchy. The main inputs to this step of the algorithm are the *storage cache hierarchy tree* and the graph built in the previous (initialization) step. The storage cache hierarchy tree captures the storage cache hierarchy from the storage nodes, through I/O nodes, to the client nodes, in a tree form. If there is only one storage node, it (its cache) represents the root of the tree. If on the other hand there are multiple storage nodes, we create a dummy node as the root node, signifying a hypothetical last level unified storage. In such a scenario, the dummy node will have the multiple storage nodes as its children. After that, the graph built in the initialization step is considered and the nodes of this graph are hierarchically clustered based on the storage cache hierarchy tree.⁴ Clustering is done beginning with the root of the hierarchy tree and further, level by level, until the leaf level (the client node level). In this clustering step, we consider the *dot product* (\bullet) of the tags of two clusters, which quantifies the degree of data chunk sharing between the two clusters, as the qualitative measure of *affinity* (similarity). The "tag" of a cluster is the bit-wise sum of the tags of all the nodes in the cluster. At each level, the number of clusters formed is equal to the number of child nodes in the storage cache hierarchy tree.

In case the number of clusters is less than the number of children nodes at the current level, the clusters are split continually until the number of clusters is increased to be equal to the number of child nodes. We want to make it clear that the child nodes here refer to the nodes of the storage cache hierarchy tree (not the nodes of the iteration chunk graph). Therefore, at the end of this clustering step, we have the required number of iteration chunk clusters for the current level of the storage cache hierarchy.

Load Balancing: In this step, we try to balance the sizes of the iteration chunk clusters formed in the previous clustering step using a greedy strategy. We note that the size of an iteration chunk is equal to the total number of iterations assigned to that iteration chunk, and the size of a cluster is sum of the sizes of the iteration chunks that belong to that cluster. We compute the lower and upper limits on the size of each cluster using a tunable parameter called the "balance threshold", which is the maximum tolerable imbalance across the iteration counts of any two client nodes. In order to balance the sizes, the iteration chunks are evicted progressively from the largest-sized cluster to the smallest-sized cluster in each step. Eviction is performed only if the donor cluster size does not drop below the lower limit and the recipient cluster size does not go above the upper limit after eviction. Importantly, since we use a greedy approach, eviction is done such that the dot product of the tags of the evicted iteration chunk and the recipient cluster is maximized. An iteration chunk is split according to the balance threshold requirements prior to the eviction process if no eligible iteration chunk is found. This eviction step is repeated until the iteration chunk sizes comply with the balance threshold.

In order to consider the data chunk sharing at each level of the I/O cache hierarchy, the clustering and load balancing steps of our algorithm are repeated at each level of the storage cache hierarchy tree. The final output of this loop distribution algorithm is a set of iteration chunk clusters. The number of iteration chunk clusters in this set is equal to the number of client nodes in the target architecture. Therefore, our algorithm determines the iteration chunks to be assigned to each client node based on data sharing.

4.4 Example

⁴We note that the graph we build captures similarity between iteration chunks, whereas the tree we build represent the underlying storage cache hierarchy.

We now illustrate how our loop iteration distribution algorithm works using a simple code fragment. The code fragment shown in Figure 6 accesses a disk-resident array (A), which is assumed to be divided into 12 data chunks, each of size d . We note that, during each iteration of the loop, 4 data references are accessed. In this example, we consider the target storage cache hierarchy depicted in Figure 7. This hierarchy has three layers, with four client nodes, two I/O nodes, and a single storage node.

```

...
int A[m];
...
for i = 0 to m - 4d - 1
{
    int x = i % d;
    ...
    A[i] = A[x] + A[i+4d] + A[i+2d];
}

```

Figure 6: Example code fragment.

The initialization step of the hierarchical loop distribution algorithm divides the set of all loop iterations (iterations $i = 0$ to $i = m - 4d - 1$ in Figure 6) into iteration chunks based on their tags. As described earlier, a tag is the signature of the data chunks accessed by an iteration/iteration-chunk. The different iteration chunks and their corresponding tags are shown in Figure 8. Figure 8 also shows the initial graph. This is the graph built during the initialization process with the iteration chunks as its nodes. We note that, for the sample code fragment considered here, this graph is actually a complete graph with an edge between every distinct pair of nodes. Since the edges with a weight of 1 do not contribute anything significant to our clustering algorithm, we do not show those edges explicitly to make the graph legible.

After the initialization step, our hierarchical loop iteration distribution algorithm performs iteration chunk clustering at each level of the storage cache hierarchy, starting from the root (storage cache) level. In this case, clustering is performed both at the I/O node level and the client node level (as there is only one cache in the storage node level of the target system). Firstly, the I/O node level is considered and the iteration chunks are clustered into two clusters, one for each of the I/O nodes present. After this, the second level of clustering at the client node level is performed. To start with, the cluster assigned to I/O node IO_0 is considered. The iteration chunks contained in this cluster are further clustered into two, one for each client node under I/O node IO_0 . This process is repeated for the cluster assigned to I/O node IO_1 . Therefore, at the end of two levels of clustering, we are left with four clusters, one for each of the four client nodes present in the system. We note that, after each of the clustering steps, the clusters are balanced during the corresponding load balancing step. In this example, since the iteration chunks are balanced in terms of number of iterations they contain, there is no need to split the iteration groups during the load balancing step. Figure 9 summarizes this hierarchical clustering activity and shows the formed clusters.

5. EXPERIMENTS

5.1 Setup

We used the Phoenix compiler infrastructure from Microsoft [43] to implement (automate) our proposed scheme as well as the alternate data locality optimization scheme we evaluated. We used

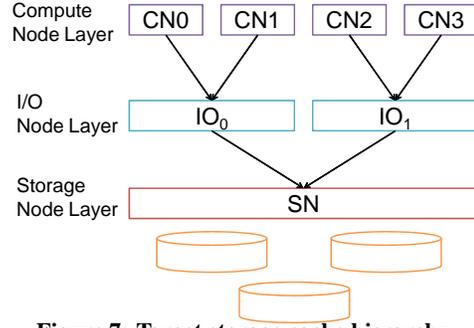


Figure 7: Target storage cache hierarchy.

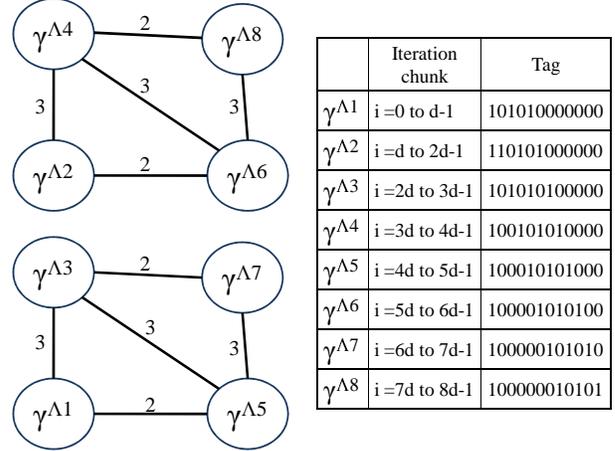


Figure 8: Initial graph description with tags. Each vertex denotes an iteration chunk with a particular tag, and the weight on each edge denotes the number of “1” bits in $\Lambda_i \wedge \Lambda_j$, where ‘ \wedge ’ refers to “bitwise and”.

a real platform to test the effectiveness of our proposed mapping scheme. Phoenix is a framework for developing compilers as well as tools for code testing, program analysis and optimization, to be used as the back-end for future compiler technologies from Microsoft. It defines an intermediate representation (IR) for application programs, using control flow graph, abstract syntax trees, and an exception handling model. We observed that including our approach at compile time increased the original compilation times by 46%-87%, depending on the application program being compiled. We performed our experiments using a platform configured to have compute, I/O and storage nodes and to run MPI-IO [23] on top of the PVFS parallel file system [7]. MPI-IO is the parallel I/O component of MPI-2 and contains a collection of functions designed to allow easy access to files in a patterned fashion. PVFS is a parallel file system that stripes file data across multiple disks in different nodes in a cluster. It accommodates multiple user interfaces which include the MPI-IO interface, traditional Linux interface, and the native PVFS library interface. In all our experiments (except for one application), we used the MPI-IO interface. In our setting, MPI-IO runs on the client nodes along with the PVFS client stub and the PVFS server components run on the storage nodes. The I/O nodes on the other hand execute only the forwarding daemon, relaying the calls from the client nodes to the storage nodes. We note that this structure and execution model is very similar to one employed by IBM Blue Gene/P. We also implemented storage caches at compute, I/O and storage nodes. For this purpose, a portion of the main memory in the node is reserved to keep copies of the

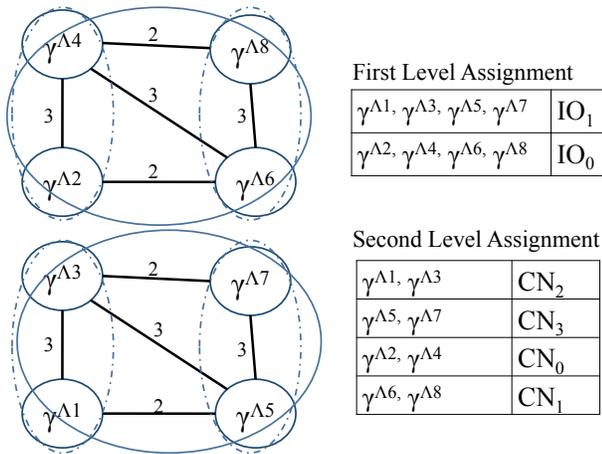


Figure 9: After clustering steps at the client node level and the I/O level.

Table 1: System parameters and their default values for our target architecture.

Parameter	Value
Number of Client Nodes	64
Number of I/O Nodes	32
Number of Storage Nodes	16
Data Striping	Uses all 16 storage nodes
Stripe Size	64KB
Storage Capacity/Disk	40 GB
RPM	10,000
Data Chunk Size	64 KB
Cache Capacity/Node (client,I/O,storage)	(2GB,2GB,2GB)

frequently-used data chunks. The unit of granularity for managing these caches is a data chunk, whose value is the same as the stripe size (at the storage node level). These storage caches are managed using the LRU policy. We note that, while the results we present depend on the specific caching policy employed, our approach itself can work with any storage caching policy, that is, how the storage caches in the system are managed is orthogonal to the problem of how loop chunks are distributed across client nodes to maximize data sharing.

Table 1 gives the important parameters and their default values for the target I/O subsystem we consider in our experimental evaluation. This architecture is similar to that shown in Figure 1 except that the number of compute nodes, I/O nodes and storage nodes are 64, 32 and 16, respectively. *Later, in our sensitivity experiments, we change the values of some of the parameters listed in Table 1 and evaluate the impact of doing so.* We note that in this table the cache capacities are given as per-node values. Therefore, for instance, the total cache capacity in the storage nodes (cumulative L3 size) is 32GB. Later, we also present the results with different cache capacities.

Table 2 gives the set of I/O intensive application programs used in this study. We used 8 applications that exhibit a variety of data access patterns. We note that *apsi* and *wupwise* are the parallel, out-of-core versions of well-known SPEC applications of the same name [45]. The total sizes of the disk resident data sets manipulated by these applications vary between 189.6GB (in *sar*) and 422.7GB (in *wupwise*). The last three columns of this table give the miss rates of these applications (the original version to be explained shortly) for different storage caches, under the parameters listed in Table 1. We see from these columns that different applications

Table 2: Application programs.

Name	Brief Description	Miss Rates (%)		
		L1	L2	L3
hf	Hartree-Fock Method	21.3	40.4	47.9
sar	Synthetic Aperture Radar Kernel	16.0	23.3	44.4
contour	Contour Displaying	15.3	39.3	67.1
astro	Analysis of Astronomical Data	28.4	54.4	76.4
e_elem	Finite Element Electromagnetic Modeling	8.3	33.6	49.9
apsi	Pollutant Distribution Modeling	17.7	25.4	36.0
madbench2	Cosmic Microwave Background Radiation Calculation	20.6	34.7	56.5
wupwise	Physics/Quantum Chromodynamics	20.8	36.3	52.8

have different distributions of miss rates. A general trend we want to emphasize though is that, as we go deeper in the storage cache hierarchy, we observe an increase in miss rates, primarily due to destructive interferences on shared storage caches among the data streams coming from different client nodes. For example, in our Hartree-Fock application, the cumulative miss rates in L1, L2 and L3 layers are 21.3%, 40.4%, and 47.9%, respectively. The highest miss rate is observed at the L3 layer because it is the most heavily shared one (each L3 cache is shared by $64/16=4$ client nodes). Our goal is to convert this destructive sharing of the common storage cache space into constructive sharing by careful distribution of loop iterations across the client nodes in the system.

For each application program in our experimental suite, we performed experiments with three different versions. The first of these is the *original* version in which the application is used as it is without any specific storage cache optimization.⁵ In this version, the set of iterations to be executed in parallel is first ordered lexicographically (which is the default order implied by the sequential execution) and then divided into K clusters, where K is the number of client nodes. Each cluster is then assigned to a client node. The second version is obtained by applying to the code well-known data locality enhancing transformations. These transformations include loop permutation (changing the order in which loop iterations are executed) and iteration space tiling (also known as blocking, which implements a blocked version of the code to improve temporal reuse in outer loop positions). To approximate the ideal tile size (blocking factor), we experimented with different tile sizes and selected the one that performs the best. After these locality optimizations, the iterations are divided into k clusters and each cluster is assigned to a client node (as in the original version). In this section, we refer to this version as *Intra-processor*, as it is an extension of state-of-the-art data locality strategy, developed originally in the context of single processor machines, to multiple processors. More specifically, this version tries to optimize storage cache behavior for each client in isolation *without* considering data sharing among clients or affinities they may have at different storage caches in the I/O subsystem. The third version used for each application program in our experimental suite is our proposed scheme explained in detail in Section 4. We refer to this scheme as *Inter-processor* in this section. We want to emphasize that the total set of loop iterations executed in parallel is the same in all three versions we experimented with; the only difference among the different versions is in the set of iterations assigned to each processor.

In the rest of this section, we present three types of results. The first of these is the miss rates for the different types of storage caches in the target architecture. The second one is the I/O latency, that is, the total time spent by the application in performing disk I/O. We note that this time also includes the cycles spent in

⁵However, the application is parallelized if it is sequential.

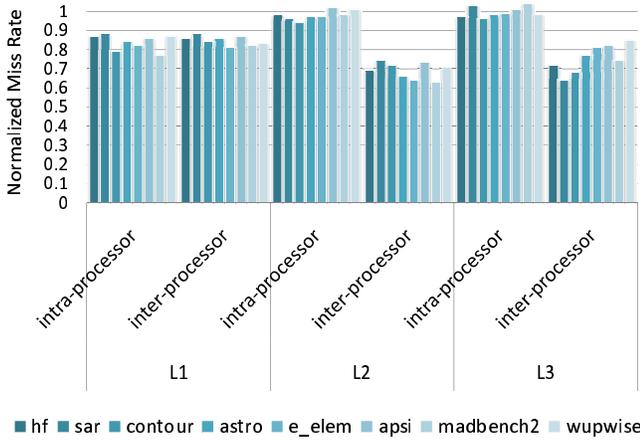


Figure 10: Normalized miss rates for the L1, L2, and L3 caches.

accessing storage caches. The third type of result we present is the overall application execution time. Our aim in giving this last type of statistics is to measure the impact of optimizing storage cache performance in deep hierarchies on overall application performance. Unless otherwise stated, the results presented below, for the intra-processor and inter-processor schemes, are given as *normalized values* with respect to the *original version* explained above. Also, the I/O latency and execution latency results presented below include all the runtime overheads incurred by our approach.

5.2 Results with the Default Values of the Configuration Parameters

We now present the results collected using the system characterized by the default values given in Table 1. In these experiments, the load balance threshold mentioned in Section 4.3 is set to 10%. Our first set of results, given in Figure 10, shows the normalized miss rates for L1, L2 and L3 caches (recall that the absolute miss rates for the original version are given earlier in Table 2). In this plot, for each application, the miss rate of the original version is set to 1. One can make two main observations from these results. First, the intra-processor scheme reduces the L1 miss rates but do not have much impact on L2 or L3 miss rates. This is not surprising since (as mentioned earlier) this scheme is an extension of single-processor centric data locality optimization and does not take into account data sharing across processors or the underlying shared cache components in the system. In contrast, our proposed inter-processor scheme reduces miss rates of the caches in all three layers (L1, L2 and L3). Later in Section 5.4, we discuss an enhanced version of this inter-processor scheme, which follows the algorithm in Figure 5 with a restructuring (scheduling) step that further improves storage cache behavior. Overall, we see from Figure 10 that the average cache miss reduction brought by the intra-processor scheme is 16.2%, 2.1% and 0.5% for L1, L2 and L3, respectively. The corresponding improvements with our inter-processor scheme are 15.3%, 31.0% and 24.6%, in the same order.

While these improvements in cache hit/miss statistics are important, one would be more interested in quantifying the impact of our approach on application wide I/O latencies. This is because the latencies incurred by some of the storage cache misses can be hidden during parallel execution, and consequently, savings in miss rates do not always translate exactly to savings in I/O latencies. The right side of Figure 11 gives the normalized I/O latency values for both the intra-processor and inter-processor schemes. It can be observed that the intra-processor and inter-processor schemes bring average

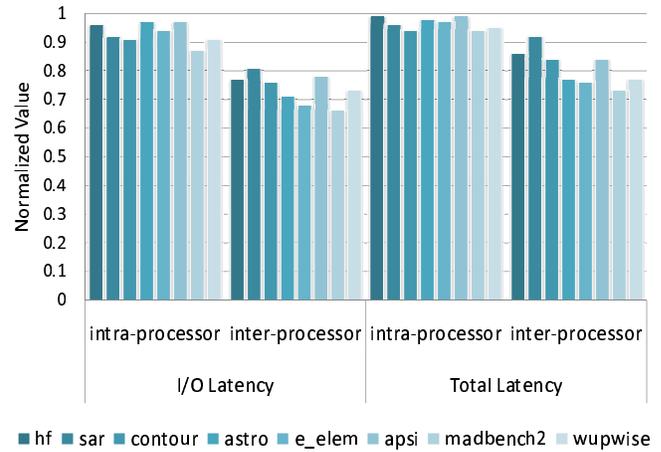


Figure 11: Normalized I/O latency and total execution time values for both the intra-processor and inter-processor schemes.

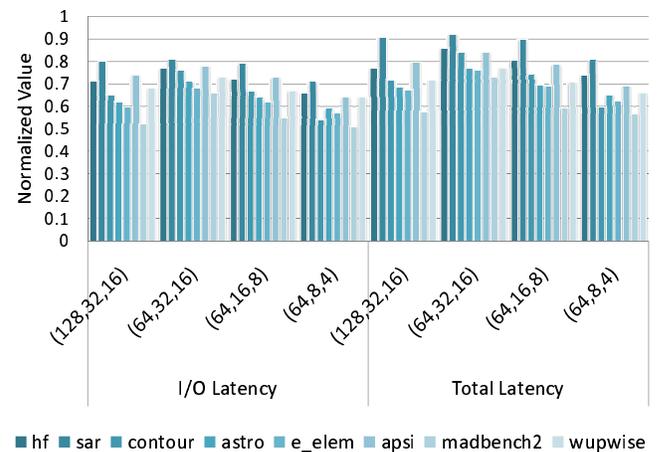


Figure 12: Normalized I/O and total execution latencies with different topologies. On the x-axis, (w,x,y) represent a configuration with w compute nodes, x I/O nodes, and y storage nodes.

improvements of 6.8% and 26.3%, respectively. This result clearly underlines the importance of careful distribution of loop iterations across the client nodes to maximize I/O performance. The right side of Figure 11 on the other hand presents the normalized overall execution times of our applications. We see that the average improvements brought by the intra-processor and inter-processor schemes are 3.5% and 18.9%, respectively, meaning that improving the performance of storage cache hierarchy by careful distribution of loop iterations (iteration chunks) across the client nodes can have significant impact on parallel execution time of an application.

5.3 Results from the Sensitivity Experiments

In this subsection, we study three parameters in detail: 1) the number of client, I/O and storage nodes; 2) storage cache capacities; and 3) data chunk size. As before, all improvements are with respect to the *original version*. Figure 12 plots the normalized I/O and total execution latencies of our inter-processor scheme under different topologies. In this bar-chart, (64,32,16) refers to our default configuration used so far. In general, for a configuration (w,x,y), each I/O node serves w/x client nodes and each storage nodes serves x/y I/O nodes (and consequently w/y client nodes). It can be observed from the results in Figure 12 that our approach

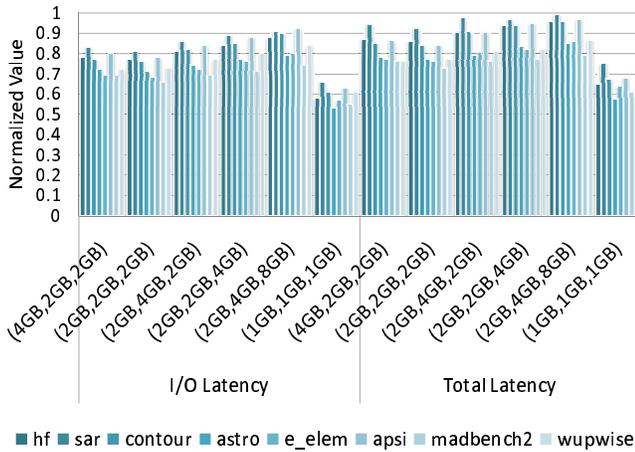


Figure 13: Results with different cache capacities. On the x-axis, (W,X,Y) indicate that per node client, I/O and storage node caches are of capacities W, X and Y, respectively.

brings more benefits when either w/x or x/y is increased. This is because an increase in any of these ratios implies that a given storage (or I/O) cache will be shared by more client nodes, and this causes the original version to suffer more. Since our results are normalized with respect to the original version, we witness an improvement. The results seem particularly encouraging when we consider the configuration (128,32,16), as they show that our approach generates better results (with respect to the original scheme) as we increase the number of client nodes while keeping the storage and I/O node counts fixed.

Our next set of experiments study the sensitivity of our results to the cache capacity. In these experiments, we use our default configuration (64,32,16). Recall from Table 1 that the default cache capacity used in our experiments so far per client node, I/O node and storage node was 2GB. Figure 13 plots the results for our inter-processor scheme with different cache capacities. A (W,X,Y) on the x-axis of this bar-chart indicates that the storage cache capacities per client node, I/O node and storage node are W, X and Y, respectively. Our main observation is that, as we increase any cache capacity (in client, I/O or storage node), the savings over the original version get reduced. This is mainly because the original version takes more advantage of extra cache capacity. This is more pronounced with the increases in cache capacities of I/O and storage nodes, as these caches are shared by client nodes and an increase in them brings relief to client nodes. We also see that reducing cache capacities by half (that is, configuration (1GB,1GB,1GB)) boosts the effectiveness of our approach. These results in Figure 13 are actually encouraging in the sense that the increases in data set sizes of I/O intensive applications outmatches the increases in storage cache capacities. Therefore, we can expect our approach to be even more effective in the future.

We next study the sensitivity of our savings to the data chunk size. Recall that the default chunk size used in our experiments so far was 64KB. One can observe from Figure 14 that, as expected, decreasing chunk size improves the improvements brought by our approach. This is because a smaller data chunk size typically leads to smaller iteration chunks, which in turn results in a finer granular clustering by our algorithm (Figure 5). While this result motivates for small data chunk sizes, we should mention that, a smaller chunk size also increases overall compilation time. For example, we observed that, as we move from 64KB to 16KB, the overall compilation time increased by more than 75%. An important trend

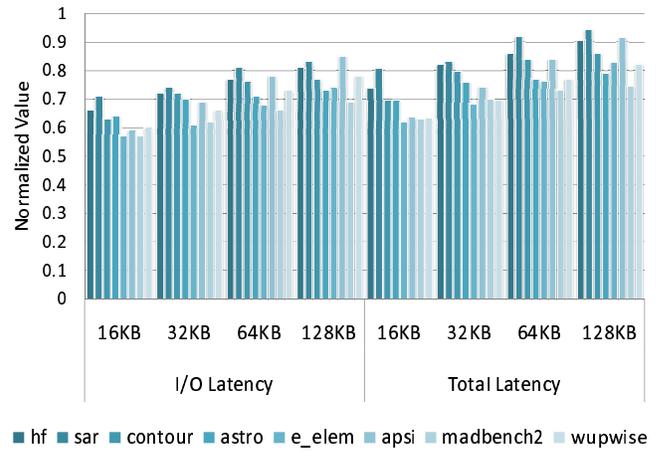


Figure 14: Results with different data chunk sizes.

from Figure 14 is that the chunk size can play an important role in determining the magnitude of our savings, and therefore, its value should be selected carefully. Determining the optimum data chunk size for a given (application, storage cache topology) pair is in our future research agenda.

5.4 Discussion

In this section, we discuss three issues related to our proposed mapping strategy: *local scheduling*, *handling data dependencies*, and *handling multiple loop nests at the same time*.

We first present an enhancement to our baseline implementation given in Figure 5. Recall that our approach discussed so far partitions the set of iterations to be executed in parallel into iteration chunks and assigns a number of chunks to each client node with the goal of improving the performance of the underlying storage cache hierarchy. However, this strategy does not say anything about the order in which the iteration chunks assigned to a client node will be executed (in the inter-processor scheme used so far we executed them randomly). However, we note that this order can also make a difference in performance. Once the iteration chunks are assigned (mapped) to client nodes, for each client node, we can reorder the iteration chunks assigned to it to maximize data reuse. We note that, after this reordering, the iteration chunks will be scheduled to execute in the resulting order. Therefore, one can use the terms “re-ordering” and “scheduling” interchangeably.

In reordering the iteration chunks, our main metric is the *chunk level data reuse*. This reuse concept however has *two dimensions* in this scheduling step. First, for each client, we want the next iteration chunk to be scheduled to reuse as much data as possible with the currently-scheduled iteration chunk (vertical dimension). And, second, we want the iteration chunks scheduled for the same slot at different clients to reuse as much data as possible among them (horizontal dimension), so that we can take advantage of shared caches. Figure 15 gives the sketch of our scheduling strategy. To re-iterate, this scheduling algorithm is applied after our iteration distribution algorithm finishes.

We start by noting that Hamming Distance is a measure that can be used to schedule the iteration chunks such that the data chunk reuse is improved. More specifically, by scheduling the iteration chunks of a client node such that the tags of the contiguously scheduled iteration chunks have the least possible Hamming Distance, data chunk reuse can be further improved. Our proposed scheduling algorithm is given in Figure 15. The algorithm starts out by considering each level in the storage cache hierarchy individually.

```

Input :
Iteration cluster,  $C_x = \{\gamma^{A_a}, \gamma^{A_b}, \dots, \gamma^{A_c}\} \forall 0 < x < k$ 
 $k$  is the total number of client nodes
/*  $C_x$  is the cluster of iteration groups assigned to client node  $x$  */
 $\alpha$  = I/O level cache reuse factor
 $\beta$  = Client node cache reuse factor

Output :
Scheduled Iteration Chunk Set,  $SC_x = \{\gamma^{A_p}$ 

Algorithm :
For Each I/O cache level  $ST$ 
 $n$  = Number of client nodes sharing the I/O level cache  $ST$ 
 $j$  = First client node under I/O level cache  $ST$ 
 $SC_i = \{\}$ , for all  $j < i < j + n$ 
 $s_i = 0$ , for all  $j < i < j + n$ 
/*  $s_i$  is the total number of iterations in  $SC_i$  */
While  $\exists C_k$ , such that  $C_k \neq \emptyset$ 
For Each  $i$  from  $j$  to  $j + n$ 
If ( $C_i == \emptyset$ ) Continue;
If ( $i == j$  and  $SC_i == \emptyset$ )
 $SC_i = SC_i + \gamma^{A_a}$ ,  $s_i = s_i + S(\gamma^{A_a})$ , such that,
 $\Lambda_a \in C_i$ , and,  $\Lambda_a$  has the least number of "1" bits
Else If ( $i > j$  and  $SC_i == \emptyset$ )
 $SC_i = SC_i + \gamma^{A_a}$ ,  $s_i = s_i + S(\gamma^{A_a})$ 
such that,  $\alpha \times (\Lambda_a \bullet \Lambda_x)$  is maximum
where,  $\gamma^{A_x}$  = Last element added to  $SC_{i-1}$ 
Else If ( $i == j$  and  $SC_i \neq \emptyset$ )
While  $s_i < s_{j+n}$ 
 $SC_i = SC_i + \gamma^{A_a}$ ,  $s_i = s_i + S(\gamma^{A_a})$ 
such that,  $\beta \times (\Lambda_a \bullet \Lambda_y)$  is maximum
where,  $\gamma^{A_y}$  = Last element added to  $SC_i$ 
Else
While  $s_i < s_{i-1}$ 
 $SC_i = SC_i + \gamma^{A_a}$ ,  $s_i = s_i + S(\gamma^{A_a})$ , such that,
 $\alpha \times (\Lambda_a \bullet \Lambda_x) + \beta \times (\Lambda_a \bullet \Lambda_y)$  is maximum
where,  $\gamma^{A_x}$  = Last element added to  $SC_{i-1}$ 
and,  $\gamma^{A_y}$  = Last element added to  $SC_i$ 
Return  $SC_x \forall 0 < x < k$ 

```

Figure 15: Cache hierarchy-conscious iteration scheduling algorithm.

Later, an iteration chunk schedule is computed for each client node considering the I/O nodes. The iteration chunk that accesses the least number of data chunks is selected and scheduled for the first client node. For the other client nodes, iteration chunk which has the minimum Hamming Distance with the last scheduled group on the previous client node is scheduled. We note here that the client nodes are considered progressively from the first to the last in order. We start the second round of scheduling after the first round of scheduling is finished for all the client nodes in order. Until all the iteration chunks are scheduled, these scheduling rounds are repeated. For all other rounds of scheduling after the first round, while scheduling for a given client node, we select the iteration chunk that maximizes both the dot product (\bullet) with the last scheduled iteration chunk on this client node as well as the dot product with the last scheduled iteration chunk on the previous client node. This way, the data chunk reuse is improved at the client node level as well as the I/O level. Two tunable parameters, α and β , are used to weigh the dot products. Therefore, to schedule an iteration chunk for a client node, we consider its left and upper neighbors (indicated by dotted circles in Figure 16). For instance, we pick γ^{A_r} (circled) to be scheduled next for client node 1 in Figure 16. This is because selecting γ^{A_r} maximizes the value $\alpha \times (\Lambda_c \bullet \Lambda_r) + \beta \times (\Lambda_q \bullet \Lambda_c)$. In our example code fragment discussed earlier in Section 4.4 (see Figure 6), after the clustering and assignment steps, the scheduling algorithm decides the scheduling order. This final schedule for all the client nodes is shown in Figure 17.

One of the major concerns in such a scheduling scheme is the balance of the iteration counts. If the iteration counts are not approximately balanced, the cache improvements may take a hit. Therefore, in order to balance the iteration counts, while scheduling on

any given client node, we go on scheduling iteration chunks as long the number of iterations assigned to this client node is equal to or just exceeds the number of iterations assigned to the previous core. The balance is maintained in a circular fashion. Therefore, at the beginning of a round, iteration count of the first client node is matched with the iteration count of the last client node in the previous round.

Figure 18 gives the improvements in L1 miss rates, I/O latencies and overall execution latencies when this scheduling algorithm is applied after the proposed loop distribution scheme (the additional improvements brought by the scheduling algorithm in L2 and L3 miss rates were limited – less than 3% each). In these experiments, the values of the α and β parameters discussed above are both set to 0.5 (that is, equal weights). We observe from the first column, which gives normalized L1 miss rates, that this scheduling strategy generates about 27.8% reduction (on average) in L1 miss rates (compared to the original version), and as a result, the improvements in I/O latency and total execution time jump to 30.7% and 21.9%, respectively.

Although not presented here in detail due to space concerns, we also performed experiments with different values for the α and β parameters. We observed that giving them equal values generate the best results we were able to collect. Specifically, if β is too big, the potential locality in the shared caches are missed, and if α is too big, L1 locality starts to suffer. Again, studying the impacts of these parameters in detail is in our future research agenda.

We next discuss how our scheme is extended to handle loops with data dependencies. In other words, when the user/compiler decides to execute a set of iterations that have cross-loop dependencies between them. In the cache hierarchy aware loop iteration assignment scheme described so far, we restricted ourselves to fully-parallel loops, that is, loops in which there are no loop-carried dependencies across iterations. However, in cases where distributing the loop iterations with dependencies could bring high benefits, our scheme can be extended by distributing the loop iterations with dependencies across available processors and ensuring correctness through inter-processor synchronization.

There are at least two ways of extending our approach to handle loops with dependencies. First, we can ensure that the clustering step clusters all iteration chunks with loop carried dependencies together in a single cluster. This can be achieved by associating an infinite edge weight between iteration chunks that have dependencies between them. This ensures correctness without the need for inter-core synchronization. However, conservatively clustering all dependent iteration chunks together may reduce the benefits of parallelism in iteration chunk execution, and therefore, this approach may not be very effective when we have large number of dependencies. Alternatively, the clustering step can treat loop carried dependencies between iteration chunks as normal data block sharing. Therefore, in the presence of dependencies across loop iterations, the data sharing resulting from these dependencies is accounted for by the edge weights used to quantify the sharing of data between the iteration chunks containing the respective iterations (provided they are different). Therefore, we can use the same cache hierarchy aware loop iteration distribution algorithm described above to improve data sharing. However, to ensure correctness, the dependencies can be detected during the local reorganization/scheduling step (explained above in this section) and corresponding inter-core synchronization directives can be inserted to respect the dependencies. Our current implementation employs this second alternative.

Finally, it is easy to extend our approach to handle multiple loop nests at the same time. Since our approach uses polyhedral algebra (which works with sets and operations on sets), it does not mat-

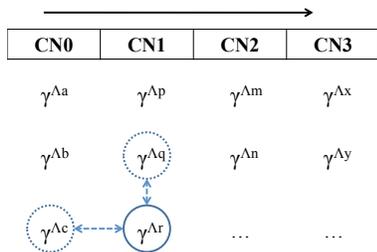


Figure 16: Scheduling order of iteration chunks.

	Time
Compute Node 0	$\gamma^{\wedge 2}, \gamma^{\wedge 4}$
Compute Node 1	$\gamma^{\wedge 6}, \gamma^{\wedge 8}$
Compute Node 2	$\gamma^{\wedge 1}, \gamma^{\wedge 3}$
Compute Node 3	$\gamma^{\wedge 5}, \gamma^{\wedge 7}$

Figure 17: Final schedule after all levels of clustering.

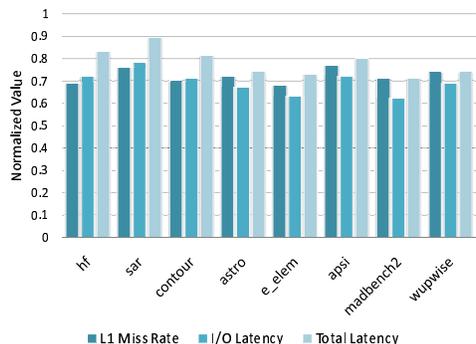


Figure 18: Improvements in L1 miss rates.

ter where (which loops) individual iterations are coming from. If we want to handle, say, two nests together, we simply form \mathcal{G} set (described earlier) to contain iterations of both the nests and the rest of our approach does not need any modification. However, we observed during our experiments that, most of existing data reuse (more than 80%) are intra-nest rather than inter-nest. Therefore, an extension of our approach to handling multiple nests together brought only an additional 3% improvement in cache hits in our application codes.

6. RELATED WORK

There have been prior research efforts aimed at optimizing different aspects of storage cache performance such as overall I/O response time and QoS. We discuss these research efforts in this section.

Caching is one of most frequently used ways of improving I/O performance, and there has been significant past work in that regard [21, 32, 52, 9, 26]. Gniady et al. propose a pattern based buffer caching, in which they use program-counter based prediction technique to correlate the I/O requests with program context [21]. Kim et al. show that commonly used LRU chunk replacement scheme does not exploit reference regularities such as sequential and loop references and consequently propose a unified buffer management scheme which exploits such regularities [32]. Jiang et al. propose a client-directed, coordinated file chunk placement and replacement protocol which takes into account the locality of file chunks of non-uniform strength [28]. They also proposed a protocol for effective coordinated buffer cache management in a multilevel cache hierarchy which reduces chunk redundancy and improves locality [27]. Chen et al. propose an eviction-based placement policy for lower level storage caches instead of the traditional access-based placement policy [9]. Jiang et al. present and evaluate an improved version of CLOCK replacement policy, which keeps track of a limited number of replaced pages [26]. Zhou et al. show that LRU replacement algorithm is not suitable for the second level cache and propose a new multi-queue cache replacement algorithm to address the second level buffer cache [52]. While these approaches showed certain degree of performance improvement, they are mainly focused on a single layer of cache hierarchy whereas our approach aims for multi layers.

To handle multi-level cache hierarchy efficiently, several cache management schemes have been proposed [50, 49, 18]. Yadgar et al. propose two storage cache management schemes to target multi-level cache hierarchy. The first local scheme operates on the local client and tries to save the most valuable chunks in cache using the access profile. On the other hand, global scheme uses the same access profile information of the clients to manage the shared

cache space so as to reduce the negative interference [50]. In [49], Wong and Wilkes explore the exclusive cache policies against the prevalent inclusive ones. Bigelow et al. consider the complete path followed by data, from client’s cache, through network, to server cache. Further they propose specific cache management policies such that each job gets the performance it requires [3]. Our approach also targeted multi-level storage cache hierarchies but is directed by the compiler. Therefore, our approach can complement these approaches by shaping the data access patterns at the application layer.

There have also been efforts targeting performance isolation and QoS aspects of shared storage caches [22, 34, 17, 46, 39]. For example, Goyal et al. note that due to sharing of resources in modern storage systems, applications with widely different QoS requirements interact in such shared resources. They further propose a scheme to guarantee QoS in such systems [22]. Ko et al. isolate the performance in order to provide service differentiation in shared storage cache system [34]. Forney et al. propose storage-aware algorithms that partition the cache with one partition per device [17]. Thiebut et al. propose to partition fully associative disk cache into disjoint chunks, sizes of which are decided by the process locality [46]. Mattson also proposes a partitioned cache to improve response time [39]. These approaches mainly use cache partitioning in order to reduce cache conflicts in the shared storage cache. Our approach, however, tried to reduce cache conflicts by mapping computations to appropriate processors. In addition, our mapping strategy improves cache locality.

Prefetching and its effects on shared storage caches have also been extensively studied. To prevent prefetched pages from being evicted before being used, Li and Shen propose a new memory management scheme that handles prefetched pages differently from the normally fetched ones [37]. Ding et al. argue that logical file-level prefetching cannot fully realize the benefits of prefetching and therefore propose to perform prefetching directly at the level of disk layout [15]. Gill and Modha propose a self-tuning, low overhead and locally adaptive cache management policy that dynamically and adaptively partitions the cache space amongst sequential and random streams [20]. Kimbrel et al. study the effects of several combined prefetching and caching strategies for systems with multiple disks [33]. Gill and Bathen study sequential prefetching and explore adaptive asynchronous algorithms [19]. Li et al. propose a sequential prefetching and caching technique to improve the read-ahead cache hit rate and the overall system response time [38]. Our work is different from these prior studies, as we explore a compiler directed strategy that improves data locality in the context of multi-level storage cache hierarchies. There have been a large number of compiler-based works on blocking/tiling for multi-level caches

(e.g., [5, 29, 48]), but they are mainly intended for caches in clients and servers. Recently Kandemir et al. [30] propose a compiler-based approach to optimize cache behavior in the I/O server only. To the best of our knowledge, this is the first work that enlists compiler's support in optimizing the storage cache behavior for multi-level hierarchies. Our proposed scheme is fully automated and very flexible in that it can work with any storage cache hierarchy given as input.

7. CONCLUDING REMARKS AND FUTURE WORK

The main contribution of this paper is a compiler directed scheme that distributes loop iterations automatically across the client nodes of a parallel system that employs multi-level storage cache hierarchy. The main goal of this distribution is to maximize the performance of the underlying cache hierarchy. We tested this scheme using a set of eight I/O intensive applications and found that it is able to improve the I/O performance of original applications by 26.3% on average, and this leads to an average of 18.9% reduction in overall execution latencies of these applications. Our experiments also showed that the proposed scheme performs significantly better than a state-of-the-art data locality optimization scheme, and performs consistently well under different values of our major experimental parameters. We also presented an enhancement to our baseline implementation that performs local scheduling once the loop iteration distribution is performed. We observed that, when using this enhancement, the improvements in I/O latency and total execution time jumped to 30.7% and 21.9%, respectively. Our future work involves integrating our storage cache conscious loop iteration distribution strategy with different loop-level parallelization techniques. Work is also underway in extending the proposed approach to handle loops that contain irregular data access patterns.

Acknowledgments

This work is supported by NSF Grants #0937949, #0833126, #0720749, #0724599, #0621402, and a grant from Department of Energy. This work was also supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

8. REFERENCES

- [1] N. Ali et al. Scalable I/O Forwarding Framework for High-Performance Computing Systems, In *Proc. CLUSTER*, 2009.
- [2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
- [3] D. O. Bigelow et al. End-to-End Performance Management for Scalable Distributed Storage. In *Proc. PDSW*, 2007.
- [4] M. Blaze and R. Alonso. Dynamic Hierarchical Caching in Large-Scale Distributed File Systems. In *Proc. ICDCS*, 1992.
- [5] R. Bordawekar et al. A Model and Compilation Strategy for Out-of-Core Data Parallel Programs. In *In Proc. PPOPP*, pages 1-10, 1995.
- [6] P. Cao et al. Implementation and Performance of Application-Controlled File Caching. In *Proc. OSDI*, 1994.
- [7] P. H. Carns et al. PVFS: A Parallel File System for Linux Clusters. In *Proc. ALSC*, 2000.
- [8] F. Chang and G. A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proc. OSDI*, 1999.
- [9] Z. Chen et al. Eviction-Based Placement for Storage Caches. In *In Proc. USENIX ATC*, 2003.
- [10] J. Choi et al. An Implementation Study of a Detection-Based Adaptive chunk Replacement Scheme. In *Proc. USENIX ATC*, 1999.
- [11] J. Choi et al. Towards Application/File-Level Characterization of chunk References: A Case for Fine-Grained Buffer Management. In *Proc. SIGMETRICS*, 2000.
- [12] Community Climate System Model. <http://www.cesm.ucar.edu>.
- [13] Cosmic Microwave Background Analysis Tools. <http://aether.lbl.gov/www/software/combat.html>.
- [14] M. D. Dahlin et al. Cooperative caching: Using Remote Client Memory to Improve File System Performance. In *Proc. OSDI*, 1994.
- [15] X. Ding et al. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proc. USENIX ATC*, 2007.
- [16] FLASH. <http://wiki.bgl.mcs.anl.gov/wiki/Flash>.
- [17] B. Forney et al. Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems. In *Proc. FAST*, 2002.
- [18] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *Proc. FAST*, 2008.
- [19] B. S. Gill and L. A. D. Bathen. AMP: Adaptive Multi-stream Prefetching in a Shared Cache. In *Proc. FAST*, 2007.
- [20] B. S. Gill and D. S. Modha. SARC: Sequential Prefetching in Adaptive Replacement Cache. In *Proceedings of the USENIX Annual Technical Conference*, 2005.
- [21] C. Gniady et al. Program-Counter-Based Pattern Classification in Buffer Caching. In *Proc. OSDI*, 2004.
- [22] P. Goyal et al. Cachecow: Providing QoS for Storage System Caches. In *Proc. SIGMETRICS*, 2003.
- [23] W. Gropp et al. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, MA, 1999.
- [24] INCITE Applications. <http://www.er.doe.gov/ascr/incite/INCITEfaq.html>.
- [25] Jaguar. <http://www.nccs.gov/jaguar/>.
- [26] S. Jiang et al. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proc. USENIX ATC*, 2005.
- [27] S. Jiang et al. Coordinated Multilevel Buffer Cache Management with Consistent access Locality Quantification. *IEEE Transactions on Computers*, Vol. 56, No. 1, 2007.
- [28] S. Jiang and X. Zhang. ULC: A File chunk Placement and Replacement Protocol to Effectively Exploit Hierarchical Locality in Multi-Level Buffer Caches. In *Proc. ICDCS*, 2004.
- [29] M. Kandemir et al. Compilation Techniques for Out-Of-Core Parallel Computations *Parallel Computing*, pages 597-628, 1998.
- [30] M. Kandemir et al. Improving I/O Performance of Applications through Compiler-Directed Code Restructuring. In *Proc. FAST*, 2008.
- [31] P. Kegelmeyer et al. Mathematics for Analysis of Petascale Data. Report on a Department of Energy Workshop. June 3-5, 2008. <http://www.er.doe.gov/ASCR/ProgramDocuments/Docs/PetascaleDataWorkshopReport.pdf>
- [32] J. M. Kim et al. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. OSDI*, 2000.
- [33] T. Kimbrel et al. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proc. OSDI*, 1996.
- [34] B. Ko et al. Scalable Service Differentiation in a Shared Storage Cache. In *Proc. ICDCS*, 2003.
- [35] G. Lakner and C. P. Sosa. IBM System Blue Gene Solution: Blue Gene/P Application Development. IBM 2008.
- [36] S. Lang et al. I/O Performance Challenges at Leadership Scale. In *Proc. SC*, 2009.
- [37] C. Li and K. Shen. Managing Prefetch Memory for Data-Intensive Online Servers. In *Proc. FAST*, 2005.
- [38] M. Li et al. TaP: Table-Based Prefetching for Storage Caches. In *Proc. FAST*, 2008.
- [39] R. L. Mattson. Partitioned Cache – A New Type of Cache. In *IBM Research Report*, 1987.
- [40] J. Mellor-Crummey et al. Software for leadership class computing. In *SciDAC Review*, pages 36-45, 2007.
- [41] S. S. Muchnick. *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997.
- [42] Omega Library. <http://www.cs.umd.edu/projects/omega>.
- [43] Phoenix. <http://research.microsoft.com/phoenix/>
- [44] R. H. Patterson et al. Informed Prefetching and Caching. In *Proc. SOSP*, 1995.
- [45] SPEC. <http://www.spec.org/cpu2000/>.
- [46] D. Thiebaut et al. Improving Disk Cache Hit-Ratios through Cache Partitioning. In *IEEE Trans. Computers*, 1992.
- [47] M. Wachs et al. Argon: Performance Insulation for Shared Storage Servers. In *Proc. FAST*, 2007.
- [48] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- [49] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. USENIX ATC*, 2002.
- [50] G. Yadgar et al. MC2: Multiple Clients on a Multilevel Cache. In *Proc. ICDCS*, 2008.
- [51] G. Yadgar et al. Karma: Know-It-All Replacement for a Multilevel Cache. In *Proc. FAST*, 2007.
- [52] Y. Zhou et al. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proc. USENIX ATC*, 2001.

The submitted manuscript has been created in part by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.