Efficiently Identifying Working Sets in Block I/O Streams

Avani Wildani
Storage Systems Research Center
Univ. of California, Santa Cruz
Santa Cruz, California, USA
avani@cs.ucsc.edu

Ethan L. Miller
Storage Systems Research Center
Univ. of California, Santa Cruz
Santa Cruz, California, USA
elm@cs.ucsc.edu

Lee Ward
Computer Science Research Institute
Sandia National Labs
Albuquerque, NM, USA
lee@sandia.gov

ABSTRACT

Identifying groups of blocks that tend to be read or written together in a given environment is the first step towards powerful techniques for device failure isolation and power management. For example, identified groups can be placed together on a single disk, avoiding excess drive activity across an exascale storage system. Unlike previous grouping work, we focus on identifying groupings in data that can be gathered from real, running systems with minimal impact. Using temporal, spatial, and access ordering information from an enterprise data set, we identified a set of groupings that consistently appear, indicating that these are working sets that are likely to be accessed together. We present several techniques to obtain groupings along with a discussion of what techniques best apply to particular types of real systems. We intend to use these preliminary results to inform our search for new types of workloads with a goal of identifying properties of easily separable workloads across different systems and dynamically moving groups in these workloads to reduce disk activity in large storage systems.

Categories and Subject Descriptors

B.4 [**Input/Output and Data Communications**]: Miscellaneous

General Terms

Measurement

Keywords

File-system contents, grouping, automated management

1. INTRODUCTION

Grouping data on disk provides benefits such as being able to avoid track boundaries [23], isolate faults [24], and avoid power consumption from excessive disk activity [19]. On exascale systems, these benefits are magnified because every element in a group may be on separate disks, necessitating many extra spin-ups that both waste power and decrease the lifetime of the system [29, 20]. Identifying these groups has been a hot research area for some time, with significant work done to bridge the gap between the semantic information provided by the file system and the behavior of the physical disk controller [4, 25, 24]. We build on this work, but focus on performing the grouping on data that can be collected non-intrusively from a running system with no modifications to the system itself other than attaching a protocol analyzer to the disk bus. On a real system, it is frequently impossible to put in hooks to collect even file-level access data. It is even harder to label data in any way useful for grouping as the data is written. Our method assumes that all we have is enough physical access to place a protocol analyzer on the storage bus to gather our data.

Cache prefetching and clustering active disk data exploits the fact that recently accessed data is more likely to be accessed again in the near future on a typical server [26]. Unlike several techniques that group data based on popularity or "hotness," we group data by likelihood of contemporaneous and related access regardless of the likelihood for the group, or any of its members, to be accessed at all. We also present partitioning algorithms including graph theoretic techniques that have not yet been considered for predictive grouping.

Before we can do any system partitioning, we need to define what it means for two disk accesses to be similar, which leads to defining a distance metric for accesses. Defining a distance metric that captures the relationship between disk accesses without over-fitting or underestimating relationships is crucial to forming meaningful groups. We introduce a set of methods for defining similarity and partitioning data into working sets, and we apply these methods to a multi-week workload trace from a multi-application server used for computer science research to test our definitions of similarity and the utility of the working sets that the partitioning algorithms built over this similarity calculation return. While we do not delve into the high-level origins of working sets, these origins could include an application that always accesses a particular subset of data, a user who tends to stay in their home directory, or a class working together on a project stored in a particular space. Focusing only on the trace allows us to adapt as the use of the system evolves. Also, this method of characterization could expose previously undetected high level activity.

Our goal is to identify and rigorously define similarity between disk accesses from different applications and use

SYSTOR'11, May 30–June 1, 2011, Haifa, Israel.

this similarity to uncover groupings within the data ("working sets") with the eventual goal of dynamically rearranging blocks of data on disk such that working sets have physical locality. While single-use servers with predictable access patterns are common in large environments, smaller institutions and single users run several applications on the same server. This makes accesses on a shared disk appear random and makes it difficult to group data on the disk to take advantage of access locality. Large, long term storage systems that grow organically also develop heavily interleaved access patterns as more use cases are added to the system. Our grouping techniques are designed to untangle accesses made by different agents in order to provide better guidelines for on-disk placement of data in the future.

After reviewing other work in the field, we present our similarity metric and partitioning algorithms, along with a discussion of how parameters are selected and which methodology is best suited to which workload. We then present the working set groupings that each of these partitioning algorithms return. We then discuss the validity of our groupings, and conclude with a discussion of the implications of our groupings and the work we are currently doing to both improve groupings and characterize workloads based on ability to be grouped into working sets.

2. RELATED WORK

Grouping data for performance gains and other optimizations has a long history. The original BSD FFS aimed to localize associated data and metadata blocks within the same cylinder groups to avoid excessive seeks [16].

Subsequent projects have focused on specific workloads or applications where grouping could provide a benefit. The DGRAID project demonstrated that placing blocks of a file adjacent to each other on a disk reduced the system impact on failure, since adjacent blocks tend to fail together [24]. Localizing the failure to a specific file reduces the number of files that have to be retrieved from backups. Our grouping methodology will allow for failures to be localized to working sets, which represent use cases, allowing more of the system to be usable in case of failure. Schindler *et al.* show the potential gain from groupings by defining track-aligned extents and showing how careful groupings prevent accesses from crossing these borders [23]. They also demonstrate the prevalence of sequential full-file access, and make a strong case for predicting access patterns across data objects in environments with small files. Since most files in a typical mixed workload are still under 3000 bytes [27], we believe our technique can be used to help define track-aligned extents.

Arpaci-Dusseau *et al.* have made a variety of advances in semantically aware disk systems [25, 4]. Their online inference mechanism had trouble with the asynchronous nature of modern operating systems. We use a longer history of block associations to un-couple the relationships between applications, and we are working on implementing their inference techniques as a secondary classification layer. Their techniques for inode inference and block association gain a great amount of information by querying the blocks, however there is an implicit assumption here that we can identify and parse the metadata. We collect only the bare minimum of data, which allows our algorithms to work almost domain-blind.

Dorimani *et al.* discuss a need for characterization of HPC workloads for the purpose of file grouping [8]. They also demonstrate a grouping using static, pre-labeled groups, where the mean group size is about an order of magnitude larger than the mean file size. Pre-labeled groupings such as these are hard to obtain for general workloads, and they are susceptible to evolving usage patterns and other variation in workload. By focusing on the core issue of inter-access similarity, we hope to be able to form dynamic groups from real-time access data. Oly and Reed present a Markov model to predict I/O requests in scientific applications [18]. By focusing on scientific applications, their work bypasses the issue of interleaved groups. Yadwadkar *et al.* also use Markov modeling, and they apply their model to NFS traces, doing best when groups are not interleaved [30]. Their method is more difficult to adapt to online data than the algorithm we present.

Essary and Amer provide a strong theoretical framework for power savings by dynamically grouping blocks nearby on a disk [11]. Other predictive methods have shown good results by offering the choice of "no prediction," allowing a predictor to signal uncertainty in the prediction [2]. C-Miner uses frequent sequence matching on block I/O data, using a global frequent sequence database [15]. Frequent sequence matching is susceptible to interlaced working sets within data and thus best for more specialized workloads, whereas our technique is suitable for multi-application systems.

Caching can be defined as looking for groupings of data that are likely to be accessed soon, based on any one of a number of criteria. Caching algorithms can even be adaptive and pick a cache criteria based on what provides that best hit rate [3]. The cache criteria can involve file or block grouping [19], but typically only in the context of grouping together popular or hot blocks of the system [28]. This is necessary because cache space is precious, so placing related, but less accessed data into the cache would only serve to pollute it [31]. DULO biases the cache towards elements that have low spatial locality, increasing program throughput, but is affected by cache pollution issues for data that is rarely accessed [12].

Our work is strongly based on previous work in cache prefetching techniques that predict file system actions based on previous events. Kroeger and Long examined using a variant of frequent sequence matching to prefetch files into cache [13, 14]. Their work provides strong evidence that some workloads (Sprite traces, in this case) have consistent and exploitable relationships between file accesses. We are targeting a different problem, though with the same motivations. Instead of deciding what would be most advantageous to cache, we would like to discover what is most important to place together on disk so that when the cache comes looking for it, the data has high physical locality and can be transferred to cache with minimal disk activity. We assume that our methods will be used alongside a traditional cache because they complement each other, and it has been shown that both read and write caches amplify the benefits of grouping [17].

Minimizing disk activity for disk accesses is especially important on some types of systems such as MAID where data is distributed around mostly idle disks [5]. Diskseen performs prefetching at the level of the disk layout using a combination of history and sequence matching [7]. Pinheiro and Bianchini group active data together on disk to minimize

the total number of disk spin-ups, but they are vulnerable to workloads where a several blocks are typically accessed together, but accessed infrequently [19]. In a large system for long-term storage, the effect of these infrequent accesses can accumulate to be a large drain on power [29].

Our end goal is to tease apart the accesses instigated by separate applications in order to obtain sets of blocks that are likely to be read or written to together. In our data, we find that the read:write ratio is almost 10:90, implying that our workload is directly comparable to the workload for personal computers with single disks in Riska's workload characterization study [22]. Riska also suggests the idea of using a protocol analyzer to collect I/O data without impacting the underlying system.

3. DESIGN

The primary design goal of our system is to identify working sets in a variety of workloads that are persistent over time and under changing conditions. Our classification scheme has two components: the distance metric used for determining distance between data points and the partitioning algorithm that identifies working sets based on these distances. We offer three different partitioning algorithms and explain how each could fit a particular type of workload and environment.

To derive "points" from block I/O traces, we treat the block offset on disk as a unique identifier for a location on the physical disk. Distances are calculated between points of the form $\langle time, offset \rangle$ or $\langle time, (offset, size) \rangle$, as indicated. Throughout this work, we treat the offset an access was made to as a unique identifier. This assumption is generally true for application files and other static filesystem components, though it will break down for volatile areas such as caches. Making this assumption allows us to use very sparse data for our analysis.

In the course of our work we have discovered that, generally, obtaining data to do predictive analysis is easier if one can make a clear argument that sharing data will not create any privacy concerns for the source organization. This concern is part of the reason much modern research in predictive grouping uses data five to ten years out of date if they use real data at all. To ensure that our work is broadly applicable, we assume a workload that contains nothing more than a block offset, size, type, and timestamp for every access. Here, "type" refers only to whether the access was a read or a write. In addition to alleviating privacy concerns, this type of data is straightforward to collect without impacting the performance of high performance systems. Riska *et al.* demonstrate this by using a protocol analyzer to collect this data from the disk bus [22].

3.1 Calculating Distance

All but one of the partitioning algorithms we present depend on a pre-calculated list of distances between every pair of points, where points each represent single accesses and are pairs of timestamps and block offsets. We experimented with adding the size of the I/O to the points, but we found this decreased the signal to noise ratio of our data considerably. In a dataset with more fixed size accesses, using $(offset, size)$ should result in a tighter classification.

3.1.1 Distance Matrices

We present distance to our partitioning algorithms in two ways. The first is a simple $n \times n$ matrix that represents the distance between every pair of accesses (p_i, p_j), with $d(p_i, p_i) = 0$ and n being the number of accesses in our workload. We calculate the distances in this matrix using simple weighted Euclidean distance, defined as $d(p_i, p_j) = d(p_j, p_i) = \sqrt{tscale \times (t_i - t_j)^2 + oscale \times (o_i - o_j)^2}$ where a point $p_i = (t_i, o_i)$ and the variables are t =time, o = block offset, and $tscale$ and $oscale$ are weighting factors that are data dependent. We hypothesize that the scaling factors are dependent on the frequency of accesses in the workload, though we found that in our test workload altering the weighting factors by small amounts from had little effect on the result. We chose to use weighted Euclidean distance because we wanted to help offset the temporal bias in our distance metrics.

In this global comparison of accesses, we were most interested in recurring block offset pairs that were accessed in short succession. As a result, we also calculated an $m \times m$ matrix, where m is the number of unique block offsets in our data set. This matrix was calculated by identifying all the differences in timestamps $T = [T_1 = t_{i1} - t_{j1}, T_2 = t_{i1} - t_{j2}, T_3 = t_{i2} - t_{j1}, \ldots]$ between the two offsets o_i and o_j. Note that T is the difference in timestamps over the set of all pairs of accesses in the system. At this stage, we do not filter for temporal proximity in order to include weaker but possibly still relevant correlations in our workload. After some experimentation, we decided to treat the unweighted average of these timestamp distances as the time element in our distance calculation. Also, since we are interested in the relative weight between temporal and spatial difference, we set $tscale = 1$. Thus, the distance between two offsets is:

$$d(o_i, o_j) = \sqrt{\left(\frac{\sum_{i=1}^{|T|} T_i}{|T|}\right)^2 + oscale \times (o_i - o_j)^2}$$

We chose mean over median because of the variability in the potential workloads. Since our end goal is to group working sets together on disk and catch long tail accesses during a disk spin period, we can not disregard the outlier offset pairs.

3.1.2 Ranged and Leveled Distance Lists

Calculating the full matrix of distances is computationally prohibitive with very large traces and impossible in an online system. We need to handle real-time data where relationships within the data are likely to have to have a set lifetime, so we also looked into creating lists of distances between the most relevant pairs of offsets. To do this, we bias towards offsets that are close in time. For very dense workloads, we suggest choosing a range r in time around each point and calculating the distances from that point to all of the accesses that fall in range, averaging the timestamps for accesses that occur with the same offset, as in the previous section. For real-time traces, the range has to be large enough to capture repeated accesses to each central point to reduce noise.

For static traces, we have the ability to paint a more complete picture of how a given offset is related to other offsets. Instead of calculating ranges around each point, we calculate ranges around each instance of a given offset o_i by calculating the distance list around each of N instances of the offset,

$$rDist(o_{i1}) = [(o_j, d(o_{i1}, o_j)), (o_k, d(o_{i1}, o_k), \ldots]$$

. We then take the list that each instance returns and combine them. This gives us a better understanding of trends in our trace and strength of association. If an offset o_i appears next to o_j multiple times, we have more reason to believe they are related. To combine the list, we first create a new list of the offsets that only appear in one of our lists (these being elements that do not need to be combined). For the remaining elements, we take the sum inversely weighted by the time between their occurrences. For example, say we have an offset o that is accessed twice in our trace, at times t_1 and t_2, with distance lists: $[(o, o_i, d(o, o_i)_1), (o, o_j, d(o, o_j)_1]$ and
$[(o, o_i, d(o, o_i)_2), (o, o_m, d(o, o_m)_2]$. The combined distance list would then be:

$$[(o, o_i, d(o, o_i)_1 + \frac{d(o, o_i)_2}{|t_1 - t_2|}, (o, o_j, d(o, o_j)_1, (o, o_m, d(o, o_m)_2)]$$

This heavily favors offset pairs that occur near to each other, which results in dynamic groupings as these relationships change. Switching the inversely weighted sum to an inversely weighted average smoothes this effect, but results in groups that are less consistent across groupings.

For our data, accesses were sparse enough that it made more sense to define our range in terms of *levels* instead of temporal distance. A level is defined as the closest two points preceding and succeeding a given access in time. A k-level distance list around a point p_i is then the distance list comparing p_i to the k accesses that occurred beforehand and the k accesses that occurred afterwards. The distance lists are calculated the same way as they are for a set range.

We would like to eventually examine trace data in real time. This introduces an inherent bias towards accesses that are close in time versus accesses close in space, since accesses close in time are continuously coming in while accesses close in space are distributed across the scope of the trace. Intuitively, this is acceptable because the question we are trying to answer is "are these blocks related in how they are accessed," which implies that we care more about ten points scattered throughout the system that are accessed, repeatedly, within a second of each other than we do about ten points that are adjacent on disk but accessed at random times over the course of our trace.

3.2 Partitioning Algorithms

Our distance calculations return a definitive answer for the question "how far is offset a from offset b." With this similarity information pre-computed, we now look at the actual grouping of accesses into working sets.

3.2.1 Neighborhood Partitioning

Neighborhood partitioning is an on-line, agglomerative technique for picking working sets based on immediate locality. This is the only one of our techniques that does not use a pre-calculated distance list. Instead, we start with a set of accesses ordered by timestamp. We first calculate a value for the neighborhood threshold, T. In the online case, T must be selected *a priori* and then re-calculated once enough data has entered the system to smooth out any cyclic spikes. The amount of data you need depends on what is considered a normal span of activity for the workload. In the static case, T is global and calculated as a workload-specific weighting parameter times the standard deviation of the accesses, assuming the accesses are uniformly distributed over time. Determining the weighting parameter falls under the sphere

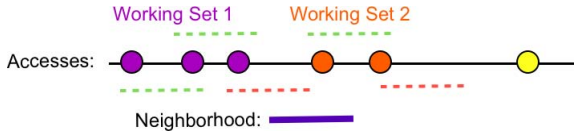


Figure 1: Neighborhood partitioning

of workload characterization, which is outside the scope of this paper. Once the threshold is calculated, the algorithm looks at every access in turn. The first access starts as a member of group 1. If the next access occurs within T, the next access is placed into group 1, otherwise, it is placed into a new group, and so on. Figure 1 illustrates a simple case with two working sets and the beginning of a third in yellow.

Neighborhood partitioning is especially well-suited to rapidly changing usage patterns because it operates on accesses instead of offsets. When an offset occurs again in the trace, it is evaluated again, with no memory of the previous occurrence. This is also the largest disadvantage of this technique: most of the valuable information in block I/O traces lies in repeated correlations between accesses. The groups that result from neighborhood partitioning are by design myopic and will miss any trend data.

The main use we foresee for neighborhood partitioning in static traces is to determine whether the trace is separable enough to make more computationally intensive methods worthwhile. It is computationally the fastest technique we explored; it runs in $O(n)$ since it only needs to pass through the access space twice: once to calculate the neighborhood threshold and again to collect the working sets. We also can easily influence the average group size by weighting the threshold value.

3.2.2 Nearest Neighbor

k-nearest-neighbor (k-NN) is a standard machine learning technique that relies on the identification of neighborhoods where the probability of group similarity is highest [10]. In the canonical case, a new element is compared to a large set of previously labeled examples using a distance metric defined over all elements. The new element is then classified into the largest group that falls within the prescribed neighborhood. This is in contrast to neighborhood partitioning where everything within a neighborhood is in the same group.

For this work, we modified the basic NN algorithm to be unsupervised (*i.e.*, not rely on pre-determined working set labels, which we cannot assume we have) and incorporate weights. The goal of weighting is to lessen the impact of access to offsets that occur frequently and independently of other accesses. In particular, in the absence of weights it is likely that a workload with an on-disk cache would return a single group, where every element has been classified into the cache group. Similar effects occur with a background process doing periodic disk accesses.

In our algorithm, we start with an $m \times m$ distance matrix as defined in Section 3.1. Instead of restricting the number of neighbors directly, we define a neighborhood set parameter k by taking the average distance between offsets in our dataset and multiplying it by a weighting factor. For the first offset, we label all of the offsets within k of that offset into a group. For subsequent offsets, we scan the elements within

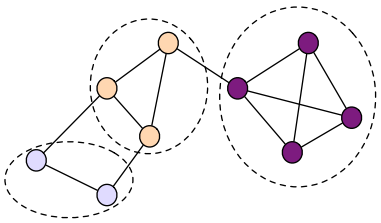


Figure 2: A clique cover of a graph of accesses. Nodes represent accesses while edges represent at least a threshold level of similarity between edges.

k of our offset and place our offset in the best represented group. The value of k is the most important parameter in our weighted nearest neighbor algorithm. If the workload consists of cleanly separable groups, it should be easier to see groupings with smaller values of k. On the other hand, a small value of k can place too much weight on accesses that turn out to be noise. Noisy workloads reduce the accuracy of nearest neighbor because with a large k, the groups frequently end up too large to be useful. We found that as long as we start above the average distance, the weighting factor on k did not have a large influence until it got to be large enough to cover most of the dataset.

3.2.3 Graph Covering

The next method we used begins with representing accesses as nodes in a graph and edges as the distance between nodes. Presenting this information as a graph exposes the interrelationships between data, but can result in a thick tangle of edges. A large, fully connected graph is of little use, so we determined a threshold of similarity beyond which the nodes no longer qualify as connected. This simplifies our graph and lowers our runtime, but more importantly removing obviously weak connections allows us to identify groups based on the edges that remain connected. This does not impact classification since these edges connect nodes that by definition bear little similarity to each other. Once we have this graph, we define a group as all sets of nodes such that every node in the set has an edge to every other node in the set; this is defined as a clique in graph theory. Figure 2 shows an example clique covering of an access graph. Note that every element is a member of a single working set which corresponds to the largest of the potential cliques it is a member of. The problem then of finding all such sets reduces to the problem of clique cover, which is known to be NP-complete and difficult to approximate in the general case [6].

Though clique cover is difficult to approximate, it is much faster to compute in workloads with many small groups and relatively few larger groups. We begin by taking all the pairs in a k-level distance list and comparing them against the larger data set to find all groups of size 3. This is by far the most time-intensive step, running in $O(n^2)$. We then proceed to compare groups of size 3 for overlap, and then groups of size 4, etc., taking advantage of the fact that a fully connected graph K_n is composed of two graphs $K_n - 1$ plus a connecting edge to reduce our search space significantly. As a result, even though the worst case for our algorithm is $O(n^G)$ (in addition to the distance list calculation), where n is the number of nodes and G the size of the maximal group, but our real runtime is closer to $n^2 + m^3 + r^4 + \ldots + z^G$, where $n \gg m \gg r \gg z$ and m, r, and z are the number of groups of size three, four, and G, respectively.

We discovered that in typical workloads, this method is too strict to discover most groups. This is likely because the accesses within a working set are the result of an ordered process. This implies that while the accesses will likely occur within a given range, the first and last access in the set may look unrelated without the context of the remainder of the set and thus lack an edge connecting them. We fix this by returning to an implicit assumption from the neighborhood partitioning algorithm that grouping is largely transitive. This makes intuitive sense because of the sequential nature of many patterns of accesses, such as those from an application that processes a directory of files in order.

In our transitive model, we use a more restrictive threshold to offset the tendency for intermittent noise points to group together otherwise disparate groups of points. We then calculate the minimum spanning tree of this graph and look for the longest path. We have to calculate the minimum spanning tree because longest path is NP-complete in the general case, but reduces to the much simpler negated shortest-path when working with a tree. We refer to this technique as the *bag-of-edges* algorithm because it is similar to picking up an edge and shaking it to see what strands are longest. Bag-of-edges is much less computationally expensive than a complete graph covering and is additionally more representative of the sequential nature of many application disk accesses than our previous graph algorithm. We found that in our small, mixed-application workload that this technique offered the best combination of accuracy and performance.

Table 1: Sample Data

Timestamp	Type	Block Offset	Size	Response Time
128166372003061629	Read	7014609920	24576	41286
128166372016382155	Write	1317441536	8192	1963
128166372026382245	Write	2436440064	4096	1835

4. EXPERIMENTS

We tested our partitioning algorithms on a publicly available dataset from MSR Cambridge. The MSR dataset represents one week of block I/O traces of enterprise servers used by researchers at Microsoft Research, Cambridge and are available from SNIA [17]. We chose these traces for two reasons: first, they allow us to simulate the bare bones block-timestamp trace we can collect from a protocol analyzer. Secondly, these traces were collected in 2007, making them more recent than most other publicly available block I/O traces. Among the traces in the collection, we selectively chose to test those that were from multi-purpose machines and could thus provide the most interesting results. Table 1 shows a sample set of accesses in our data set. For the sake of our preliminary analysis, we assume that block offset and access size form a unique, permanent identifier for a particular snippet of data. We realize this assumption does not hold true over the long term or in frequently re-written data, and we discuss how this could alter our results in section 5. We classify blocks based on the difference in timestamps, block offset, and by the order they appear in the trace.

The offsets accessed in our data were spaced between 581632 and 18136895488. Though there are gaps, likely at-

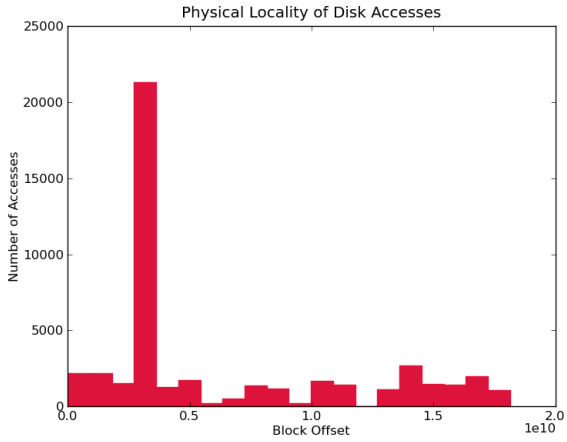


Figure 4: Visualizing the locality as a histogram exposes a localized spike in accesses.

tributable to bad sectors, Figure 3 shows that the offsets are approximately uniformly distributed, which allows us to use a consistent distance metric across our search space.

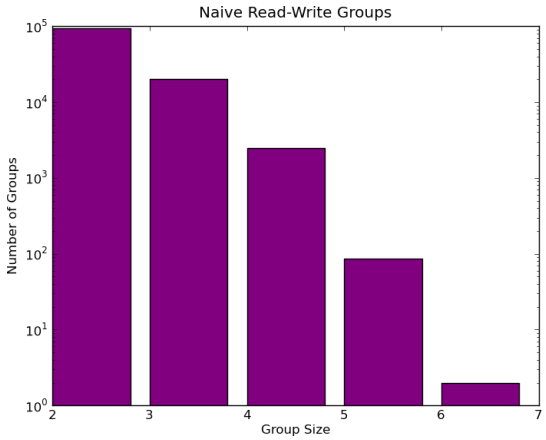
Figure 3(a) shows the accesses by block over time, and Figure 3(b) highlights the read activity. We see that despite some denser areas that likely correspond to a cache spike, the accesses in our data are approximately uniformly distributed across offsets.

This dataset was very write-heavy with a read/write ratio of 10:90. This ratio is almost entirely attributable to a small range of offsets that are likely to represent an on-disk cache. Figure 4 shows the access spike set against the otherwise near-uniform access distribution.

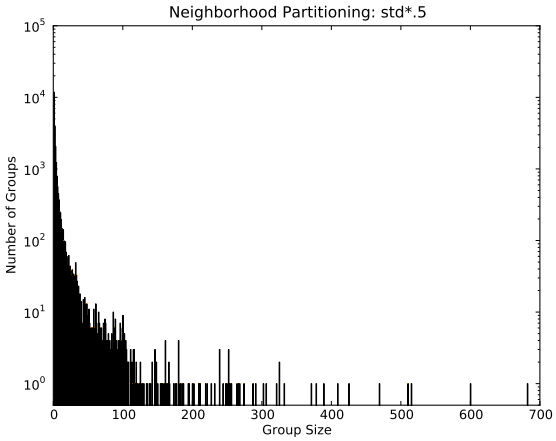
4.1 Results

We tested the neighborhood partitioning algorithm on our data first to get some visibility into what groupings were present in the data and whether it would be worthwhile to run our more computationally expensive algorithms. Neighborhood partitioning ended up being very susceptible to small fluctuations of its initial parameters and to the spike of writes in our workload. Figure 5 shows the working sets the algorithm returned with the neighborhood set to half a standard deviation, calculated over the entire trace. The read-write workload has a significantly tighter grouping because the prevalence of the writes in the cache area overtook any effect of the reads. Isolating the reads, we see in Figure 5(b) that the working sets become larger and more prevalent. This is due to the reduction in noise, leading to stronger relative relationships between the points that are left. We also notice that this technique is very fragile to the choice of neighborhood. For example, reducing the neighborhood to a quarter of a standard deviation (Figure 5(c)) causes the number of large groups to fall sharply and correspondingly increases the prevalence of small groups.

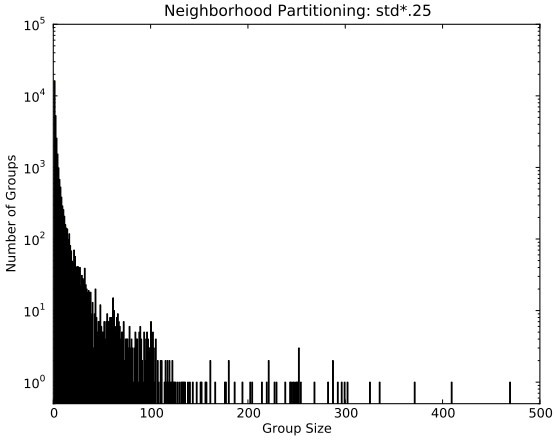
Figure 6 shows the working sets returned by running k-NN with k ranging from 3200 to 25600. The results for the k-NN working sets are more in line with expectations, with many more small groups and a few scattered large groups. The groups are fairly consistent across variation, with the larger neighborhoods resulting in somewhat fewer



(a) All Accesses

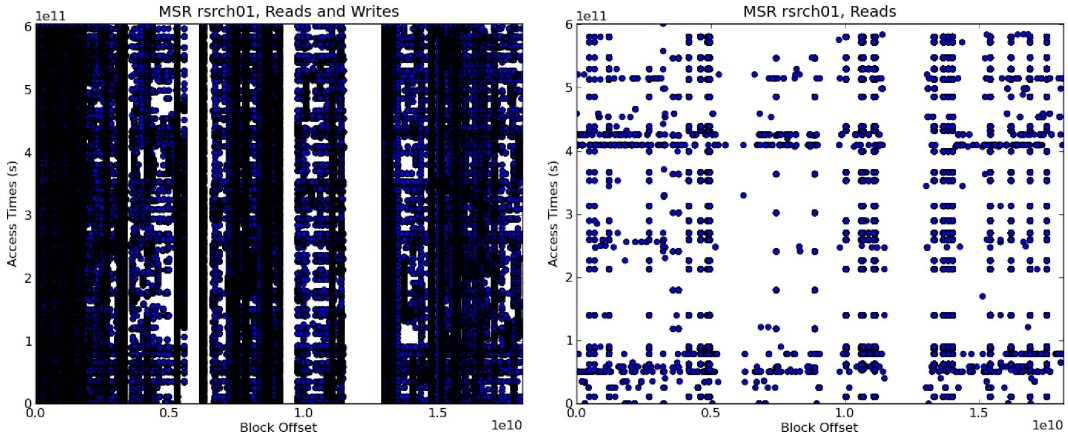


(b) Read-Only: $\sigma^2 * .5$



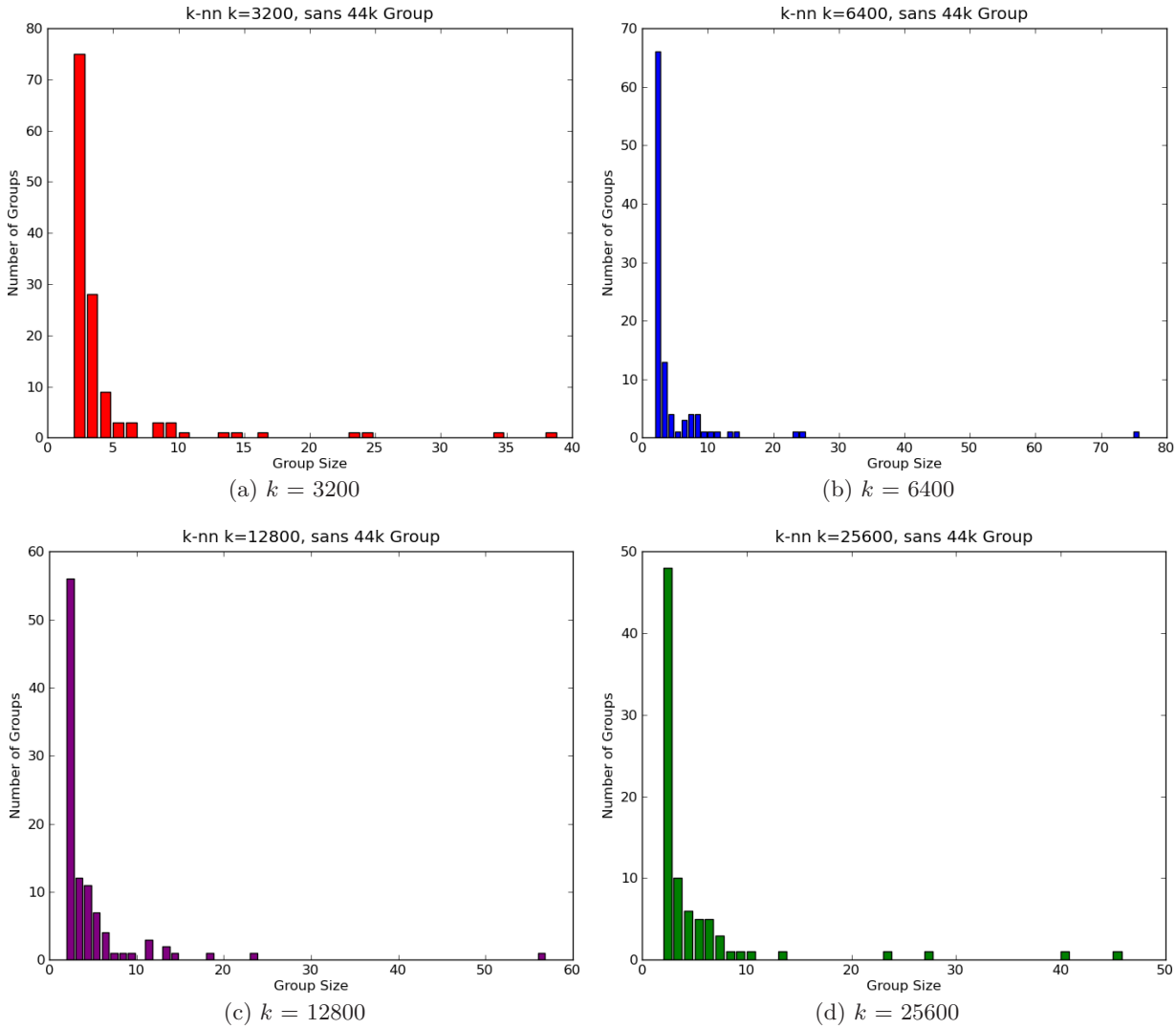
(c) Read-Only: $\sigma^2 * .25$

Figure 5: Working sets with Neighborhood Partitioning. Groupings vary drastically based on neighborhood size and workload density.



(a) All Accesses (b) Read Accesses Only

Figure 3: Block access over time for all accesses and only reads



(a) $k = 3200$ (b) $k = 6400$

(c) $k = 12800$ (d) $k = 25600$

Figure 6: Working sets with k-Nearest Neighbor. If k is very high or low, fewer large groups are found.

small groups compared to the smaller neighborhoods. Note that the graphs in Figure 6 are calculated after the cache group is taken out. The cache group is a group of size 44000 that was consistently identified by both the k-NN and bag-of-edges algorithms. The consistent identification of this group is a strong indicator of the validity of our grouping. For the sake of these graphs, however, removing it increases the visibility of the other groups and better highlights the differences between the variations in grouping parameters.

On this dataset, our clique-based graph algorithm failed to ever find a group above size two. This is useless for actually grouping data on a system since the potential benefit to prefetching one element is much smaller than the cost of doing the partitioning. The small groups are a result of the strong requirements for being in a group that this algorithm requires: namely that every member in the group be strongly related to every other member. What this tells us is that transitivity matters for grouping, *i.e.*, groups are a set of accesses that occur sequentially. Running the bag-of-edges algorithm on this data supports this hypothesis. This algorithm is built with sequentially related groups in mind, and it returned groupings comparable to k-NN in a fraction of the time. Figure 7 shows the groupings bag-of-edges returns. The levels in Figure 7 represent the levels for the n-level distance metric, where larger levels are equivalent to more lax thresholds. The majority of the groupings are similar to k-NN, though at higher levels of distance we lose the larger groups. This is due to the lack of cohesion in large groups versus smaller ones. We also suspect that there is noise interfering with the data when the search window is too large, similar to the read-write case for neighborhood partitioning.

Figure 8 provides an example of the stability of bag-of-edges under varying the weighting factors added to the parameters that make up the distance metric: namely time and the difference in offset numbers.

4.2 Validity

Working sets arise organically from how users and applications interact with the data. Consequently, there is no "correct" labeling of accesses to compare our results to. Instead, we focus on self-consistency and stability under parameter variation. As we saw in Figure 8, the working sets found by using the graph technique (or k-NN) are relatively stable under parameter variation as long as the search space for determining distance between access points remains fixed. We expect there to be variation here as a result of natural usage shifts or cyclic usage patterns.

We are analyzing our groupings using a variety of techniques that will be meaningful once we have additional datasets to compare statistics with. This includes calculating the direct overlap of elements between different groupings of working sets, calculating mutual entropy between different groupings, and calculating a discrete Rand index value across groupings [21]. In the absence of other data, the numbers tell us little more than our graphs do. Calculating these indices for two workloads would be a good first step towards characterizing workloads based on their separability into working sets. The final determinant of group validity will be the improvement in power consumption and system usability that results in re-arranging data in separable workloads to place working sets together on disk.

5. DISCUSSION

The most encouraging result of our study was the consistency of groupings in the data despite the sparsity of the traces. This indicates that it is worthwhile to look for groupings in similar multi-user and multi-application workloads even if the only data available is block offsets and timestamps. Being able to collect useful workloads without impacting privacy or performance is invaluable for continuing research in predictive data grouping. This also reduces the cost of our analysis substantially, since we can determine whether a workload will be separable before trying advanced techniques to identify groupings and disrupting the system to group working sets together on disk.

A concern early on was that the access groups we discovered would overlap, leading to a need to disambiguate and manually tune our models to the data set. It turned out, however, that in every classification scheme we used we never had overlapping group chains. This allowed us to keep our methodology general and more likely to be easily portable to other data. More importantly, this is a strong indication that our groupings represent separate access patterns. If they did not, it is likely some of them would have had over-lapping components since the accesses are uniformly distributed.

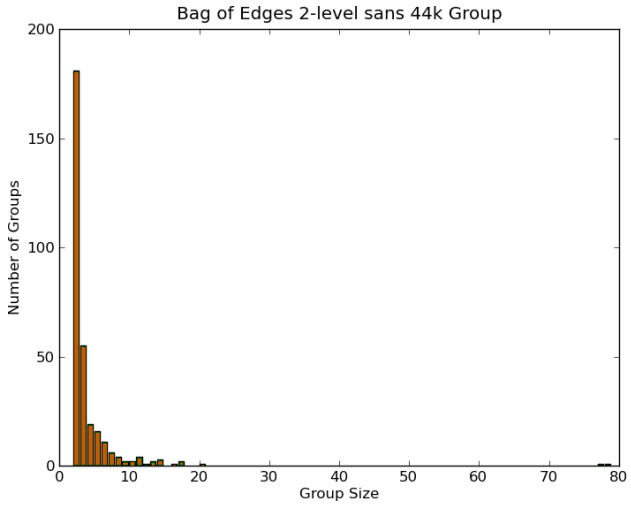
A pressing concern for the viability of this work is the speed of computations. At this stage, while testing algorithms, we wrote all code in Python and did not optimize for speed. Under this constraint, the localized distance techniques ran nearly instantaneously while the global techniques, particularly the graph techniques, took between 20 and 35 minutes per run. We believe that much of this speed can be regained by tighter code, and online implementations will handle less data per timestamp than our static test case.

We realize the assumption that block offsets do not uniquely identify a block is not strictly true in some systems. A majority of data that is frequently overwritten is in the cache block, however, and this is consistently identified as a single group by our algorithms. The less frequent overwrites that occur as a result of normal system use should be handled by the adaptability of our algorithms over time. If the content of a block offset changes, it will start being placed into different groups as the algorithms update the distance matrix.

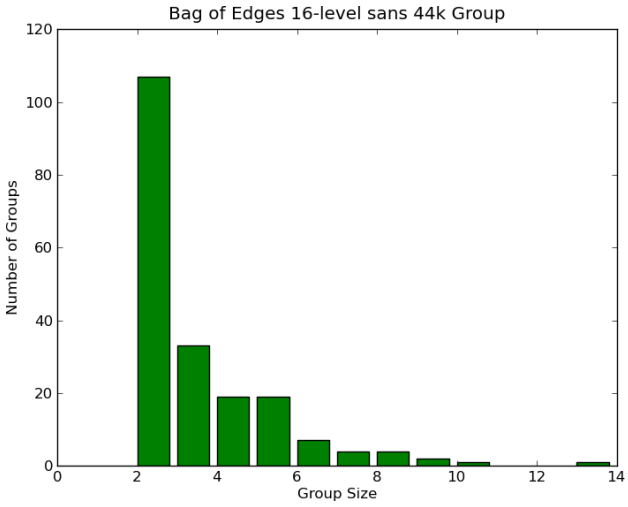
One surprising feature of our dataset was that it had a long list of consecutive writes to the same block. We believe that these writes are the result of overwrite activity in a log or a disk-based cache. These types of points are frequently present but filtered in other block I/O traces [1]. We intentionally include these points in our classifications to verify that they trivially classify into their own working set. Our k-NN and bag-of-edges methods can work around the noise of the spike to produce realistic working set groups given reasonable parameters.

6. CONCLUSIONS

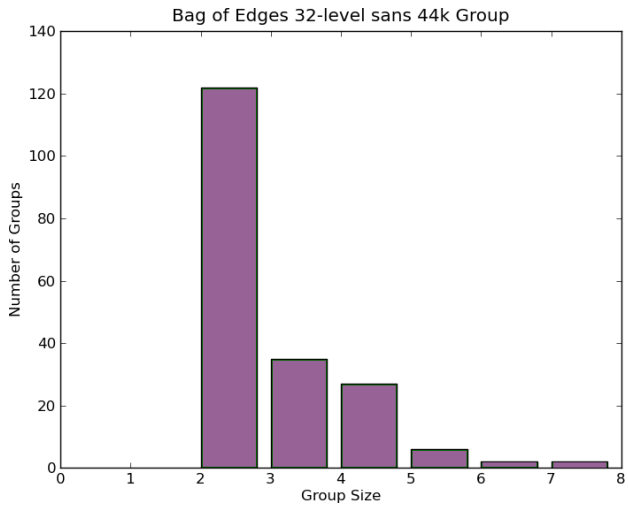
We have presented two distance metrics and three partitioning algorithms for separating a stream of I/O data into working sets. We have found that the working sets discovered by our method are stable under perturbation, implying that they have a high level basis for existing. Our methods are broadly applicable across workloads, and we have presented an analysis of how different workloads should respond



(a) 2-level



(b) 16-level



(c) 32-level

Figure 7: Working sets with the bag-of-edges algorithm. Higher levels result in much smaller groupings.

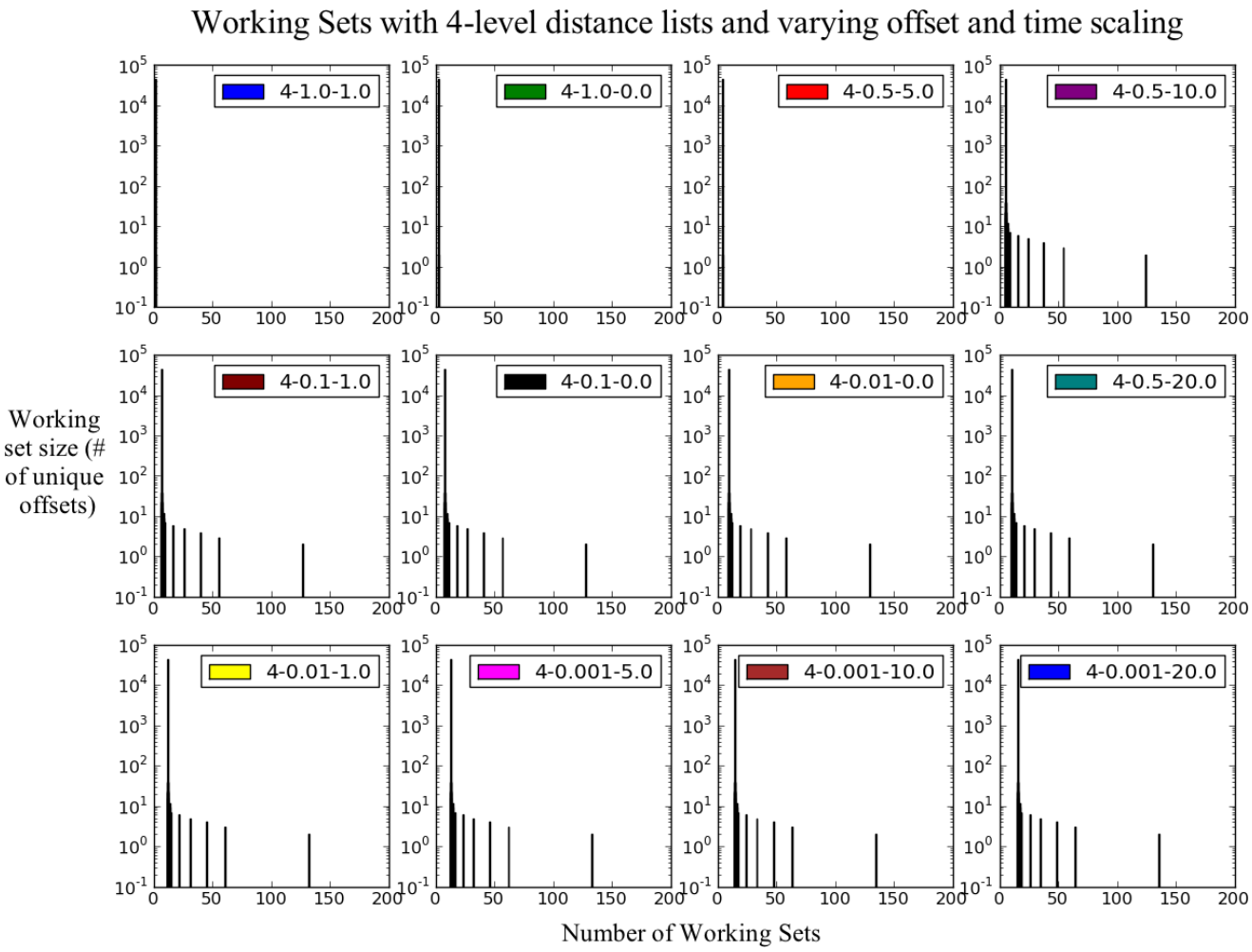


Figure 8: The Bag-of-Edges technique with varying offset and time scaling. Each graph is labeled as number of levels - offset scaling factor - time scaling factor.

to different partitioning methods. Unlike previous work in the field, we perform analysis using only data than can be collected without impacting the performance of the system. Our methods are also designed to separate working sets that are interleaved within the I/O stream. Finally, our methods are designed for use on disks instead of cache, changing the design goals from "likely to be accessed next" to "likely to be accessed together."

A consistent, easily calculable grouping that is not tied to a specific workload opens up two main avenues of work that is essential for the next stage of exascale system development. First, we will be able to characterize workloads based on how they are separated and how separable they are. Knowing a workload is likely separable allows us to move onto the next step, which is dynamically re-arranging data across a large storage system according to the working set classification of the data. Being able to re-arrange data to minimize spin-ups will be essential to keeping down power cost and increasing the long term reliability of these increasingly vital storage systems.

6.1 Future Work

Our current project is to use a protocol analyzer to collect block I/O data from a mixed-use, multi-disk educational storage system to provide a direct comparison and validity numbers to extend this work. With this data stream, we hope to implement working set detection in real-time, as well as track potential power savings and reliability gains from grouping the data together according to the assigned working set.

Once we have more data, our next step is to discover what about a workload makes it amenable to this sort of grouping. We believe that workloads with distinct use cases, whether they be from an application or a user, are the best bet for future grouping efforts, but many HPC and long-term storage workloads share some of the surface level properties that make the application servers good candidates. The goal of this line of questioning is to derive a set of characteristics of a workload that would indicate how easy it is to group along with what parameters to try first.

Another angle we are interested in is backtracking from our working sets to discover which sources tend to access the same offsets of data. Once we know this, we can implement more informed cache prefetching and, in large systems, physically move the correlated offsets near to each other on disk to avoid unnecessary disk activity. Previous work has led us to believe that even if files are duplicated across disks, the potential gain from catching subsequent accesses in large, mostly idle systems is high enough to make it worthwhile [29]. We are also interested in refining the graph covering algorithm to accept groups that are only partially connected instead of requiring complete cliques by implementing techniques from community detection [9].

7. ACKNOWLEDGMENTS

This research was primarily conducted at and funded by Sandia National Laboratories. This research was also supported in part by the National Science Foundation under awards CNS-0917396 (part of the American Recovery and Reinvestment Act of 2009 [Public Law 111-5]) and IIP-0934401, and by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768. We also thank the industrial sponsors of the Storage Systems Research Center and the Center for Research in Intelligent Storage for their generous support.

8. REFERENCES

[1] A. Amer and D.D.E. Long. Aggregating caches: A mechanism for implicit file prefetching. In *MASCOTS 2001*, pages 293–301. IEEE, 2002.

[2] A. Amer, D.D.E. Long, J.F. Paris, and R.C. Burns. File access prediction with adjustable accuracy. In *IPCCC 2002*, pages 131–140. IEEE Computer Society, 2002.

[3] I. Ari, A. Amer, R. Gramacy, E.L. Miller, S.A. Brandt, and D.D.E. Long. ACME: adaptive caching using multiple experts. In *Proceedings in Informatics*, volume 14, pages 143–158. Citeseer, 2002.

[4] A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, L.N. Bairavasundaram, T.E. Denehy, F.I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: past, present, and future. *ACM SIGMETRICS Performance Evaluation Review*, 33(4):29–35, 2006.

[5] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, page 11. IEEE Computer Society Press, 2002.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[7] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In *2007 USENIX ATC*, pages 1–14. USENIX Association, 2007.

[8] S. Doraimani and A. Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 153–164. ACM, 2008.

[9] J. Duch and A. Arenas. Community detection in complex networks using extremal optimization. *Physical Review E*, 72(2):027104, 2005.

[10] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*, volume 2. Citeseer, 2001.

[11] D. Essary and A. Amer. Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication. *Trans. Storage*, 4(1):1–23, 2008.

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST 2005*, page 8. USENIX Association, 2005.

[13] T.M. Kroeger and D.D.E. Long. Predicting file system actions from prior events. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, page 26. Usenix Association, 1996.

[14] T.M. Kroeger and D.E. Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX Annual Technical Conference, General Track*, pages 105–118, 2001.

[15] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File*

and Storage Technologies, pages 173–186. USENIX Association, 2004.

[16] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[17] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[18] J. Oly and D.A. Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[19] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *ICS '04*, pages 68–78. ACM, 2004.

[20] E. Pinheiro, W.D. Weber, and L.A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FASTâĂŹ07)*, 2007.

[21] W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[22] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–103, 2006.

[23] J. Schindler, J.L. Griffin, C.R. Lumb, and G.R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies*, 2002.

[24] M. Sivathanu, V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *ACM TOS*, 1(2):133–170, 2005.

[25] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, 2003.

[26] C. Staelin and H. Garcia-Molina. Clustering active disk data to improve disk performance. *Princeton, NJ, USA, Tech. Rep. CS–TR–298–90*, 1990.

[27] A.S. Tanenbaum, J.N. Herder, and H. Bos. File size distribution on UNIX systems: then and now. *ACM SIGOPS Operating Systems Review*, 40(1):104, 2006.

[28] J. Wang and Y. Hu. PROFS-performance-oriented data reorganization for log-structured file system on multi-zone disks. In *mascots*, page 0285. Published by the IEEE Computer Society, 2001.

[29] A. Wildani and E.L. Miller. Semantic data placement for power management in archival storage. In *Petascale Data Storage Workshop (PDSW), 2010 5th*, pages 1–5. IEEE, 2010.

[30] N.J. Yadwadkar, C. Bhattacharyya, K. Gopinath, T. Niranjan, and S. Susarla. Discovery of application workloads from network file traces. In *Proceedings of the 8th USENIX conference on File and storage technologies*, page 14. USENIX Association, 2010.

[31] X. Zhuang and H.H.S. Lee. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, pages 18–31, 2007.