Avoiding State-Space Explosion of Predictive Metadata with SESH

David Essary
Computer Science Dept.
University of Pittsburgh

Ahmed Amer
Computer Engineering Dept.
Santa Clara University

Computer Science Dept.
University of California, Santa Cruz

Abstract

With growing storage capacities, the amount of required metadata for tracking all blocks in a system becomes a daunting task. On mobile systems, the problem is compounded by a need to make the best use of available resources. Our previous work demonstrated a system software effort in the area of predictive data grouping for reducing power and latency on storage systems. Such efforts can reduce both power consumption and strain on underlying system hardware, while improving performance. Our work utilizes structures similar to prior efforts in prefetching and predictive caching, keeping a fixed number of immediate successors per block. While providing powerful predictive capabilities and being more space efficient in the required metadata than previous strategies, there remains a growing concern of how much data is actually required. We present a novel method of storing equivalent information, SESH, a Space Efficient Storage of Heredity, that is resistant to state-space explosion of predictive metadata, utilizing block-level predictability to reduce the overall metadata storage by up to 99% without loss of information. As a result, we are able to provide a predictive tool that is adaptive, accurate, and robust in the face of workload noise, for a tiny fraction of the metadata cost previously anticipated; in some cases, reducing the required size from 12 GB to less than 150 MB.

***Keywords*-data grouping; disk storage systems; metadata; power; power management; replication;**

I. Introduction

Hard disk and storage system performance concerns, both in terms of latency access and power consumption, continue to gain importance for modern computer systems. Both hardware and software efforts continue to try to reduce latencies and response times while demanding greener, more efficient use of resources. Many system software projects involve tracking and profiling workloads in order to predict and identify access patterns, both at the file and at the block level. Such techniques include data placement, both static as well as dynamic, caching and prefetching, as well as data replication and shuffling. But with growing disk and storage capacities, the volume required by metadata for tracking all blocks in a system becomes a daunting task in its own right. Further, while hardware advancements provide significantly larger storage systems, the ratio of used storage has been shown to remain stable, even with these larger disks [1].

In previous work, we have demonstrated a system software effort in the area of predictive data grouping for reducing power and latency on hard disks [2]. The structures used, very similar to prior efforts in prefetching and prefetch caching, track access successor information at the block level, keeping a fixed number of immediate successors per block. While providing powerful predictive expansion capabilities and being more space efficient in the amount of required metadata than many previous strategies, there remains a growing concern of how much data is actually required. In this paper, we present a novel method of storing equivalent information, *SESH*, a *Space Efficient Storage of Heredity*. This method utilizes the high amount of block-level predictability observed in a number of workload trace sets to reduce the overall metadata storage by up to 99% without any loss of information. As a result, we are able to provide a predictive tool that is adaptive, accurate, and robust in the face of workload noise, for a tiny fraction of the metadata cost previously anticipated; in some cases, reducing the required size from 12 GB to less than 150 MB.

978-1-4244-5736-6/09/$26.00 ©2009 IEEE

II. Motivation

Optimizing storage system performance in the face of varying workloads requires the accurate tracking and exploitation of patterns in data access behavior. Such information is useful for a broad range of applications, including caching, placement, workload shaping, data collocation and migration. Unfortunately, tracking access behavior and predicting future access behavior can result in large metadata demands. This is true when dealing with data at the granularity of files and objects, but quickly becomes unmanageable when attempting to monitor block-level access behavior in large storage systems. An explosion in metadata volume is doubly problematic when we consider that retrieving and updating such metadata can suddenly become an additional burden upon the storage subsystem. On the other hand, arbitrarily limiting the volume of metadata being maintained will only allow for optimizations to data within a current hotspot, the currently active working set, which is arguably less in need of pattern discovery and placement optimization (due to the effectiveness of even basic caching schemes on such subsets). This inevitably precludes the opportunity to discover longer-term patterns across less intensely active regions.

To improve the accuracy of placement and collocation decisions, and improve the overall performance of predictive analysis of data access patterns, we wish to maintain as much metadata as possible, but only if it is useful. Our previous work on predictive data grouping [2] demonstrates one such strategy that stores a number of direct block successors for each data access. Our strategy shows promise in the area of data grouping, and is similar to previously explored strategies in prefetching and prefetch-caching strategies adopted by Kroeger *et al* at the file level [3], [4]. We present a study of how it is feasible to reduce the metadata requirements of our strategy in the face of block-level I/O workloads. The structures used in our work are reminiscent of the limited-length queue of access successors in the *Recent Popularity* strategy adopted in [5]. Such single-successor strategies are often chosen for their efficiency benefits over multicontext modeling, yet still require huge amounts of storage. Minimally, we would need to track the root block's id, which could simply be a translated location within an array, and the queue of accesses, each of which is a block id. Thus, the total storage space would be the number of successors stored, s, times the total number of blocks, t. For modern systems, this metadata volume is too large. For a 4 TB disk array, assuming a block size of 4 KB, this would mean storing information for 1 billion blocks. Assuming a 64-bit address, this system would require 8 GB of space *just for storing a single successor*. We address the issue of metadata volume requirements in *SESH* by observing that most blocks share two properties...

1) They only have a single successor.
2) The only successor they have is the next sequential block.

Using this information, we are able to drastically reduce the total size needed for our predictive information while incurring little overhead. Further, our strategy scales better in the number of successors tracked.

The remainder of this paper is organized as follows. Section III discusses prior art related to *SESH*. Section IV briefly describes the structures used and details our experiments. Results are presented in Section V and we conclude with Section VI.

III. Background and Related Work

A study on graph-based access predictors was first presented by Griffioen and Appleton[6]. The use of the last successor model for file prediction, and more elaborate techniques based on pattern matching, were first presented by Lei and Duchamp[7]. Similar work has been done researching a last successor predictor, finite multi-order context modeling (FMOC) models from branch prediction methods, and a partitioned context model (PCM)[3]. While a "last successor" strategy predicted with surprising accuracy, there tends to be enough noise in an access stream to confuse it. A more stable predictor, Noah, was presented that removes this noise by predicting only if a stability condition is satisfied. General and specific accuracy were used to compare Noah with last successor and first successor[5]. It is noted that Noah suffers from non-decreasing general accuracy for high stability parameters. A new predictor, *Recent Popularity*, is shown to solve this problem. It is also noted that *Recent Popularity* adapts quicker with changing workloads than Noah. To benefit from this robustness and adaptability, our techniques use variants on *Recent Popularity* for gathering data for prediction.

Kroeger and Long compared the predictive performance of the last successor model, Griffioen and Appleton's graph-based strategy, and new techniques based on context modeling and data compression[8]. The earliest proposed use of data compression strategies to predict disk accesses was presented by Vitter and Krishnan[9], [4]. Shriver *et al.*[10] has provided analytical reasoning for the benefits of read-ahead buffer-

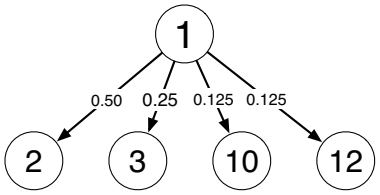
ing and prefetching. Other recent work on ASP[11] presents a study of a strip prefetching scheme for striped disk arrays. Any prefetching strategy must have a reasonable lead-time in order to retrieve data before it is actually requested. Additionally, any benefit from this prefetching, like spin-down techniques, lie directly on the data path. Our strategy enables the decoupling of the strategy from the data path, allowing us to shut down any regrouping while still benefitting from previous efforts to properly cluster data.

Recent work has shown advances toward utilizing device-level knowledge of physical data layout. DULO[12] presents a buffer cache management scheme that exploits both temporal and spatial locality, while DiskSeen[13] presents work utilizing similar table structures for use of predictive prefetching. DiskSeen fetches at the device level, and is designed to be synergistic with file-level prefetching strategies. More recent work on TaP[14] describes using a separate data structure to store previous addresses in order to identify sequential data streams without having to use precious cache space to do so. Our work seeks to increase the likelihood of spatially close blocks, and would be highly beneficial to such location- and stream-aware strategies.

Traditional research to improve performance of hard disks by modifying I/O workloads include scheduling strategies such as SSTF, SCAN[15], C-SCAN[16], and LOOK [17]. More recently, approaches for decreasing the growing impact of rotational delay have been presented[18], [19], [20]. These efforts are considered orthogonal to our current and previous work on prediction and data regrouping.

Access patterns can be used to rearrange tracks on the disk[21], a problem known to be NP-Hard[22], to improve on the organ-piping method[23], detailed and discussed in depth by Wong[24]. Such patterns can also be used to identify which files to move to tertiary storage[25]. Other forms of disk management include storing data that does not cross track boundaries[26] as well as how to extract that information and use it as stripe unit boundaries[27], storing inodes by embedding them in their directory, and grouping together small files on disk to be read as one[28].

Early data placement studies attempted to use frequency of access as an estimated likelihood in order to optimally place high-demand data. The necessary automation of optimum file arrangement, specifically by placing popular files near the center of the disk cylinder, has been addressed by Staelin and Garcia-Molina [29], whose work dealt with models that provided optimal placement of files where accesses were independent. However, data accesses often involve dynamic relationships, where access dependencies change over time. Berkeley's FFS[30], [31] includes attempts to cluster related data and metadata into cylinder tracks on a disk. However, these approaches typically require disjoint sets as groups. Our approach makes no such constraints, allowing replication between groups formed. Similar replication was performed by Akyürek and Salem in 1995, where popular "hot" blocks were copied to a common disk area[32]. However, this study was based only on global popularity rather than inter-file relationships. Examples of efforts in automated grouping include C-FFS[28] (collocating FFS), which bases grouping on a directory-membership heuristic, and Hummingbird[33], which utilizes the underlying structure of web files. In contrast, our model does not require any knowledge of underlying data structure, as our grouping mechanism can establish relationships based on observed access behavior, as opposed to inference from file location or content.



(a) *Visualization of OpExTree structure*

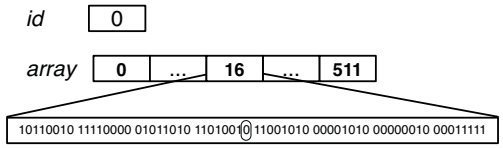
root	1			
successors	2	3	10	12
counts	4	2	1	1

(b) *OpExTree implementation*

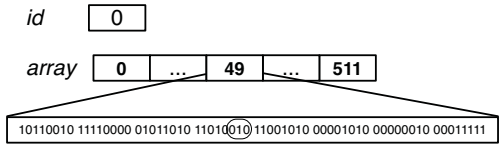
Fig. 1. Optimal Expansion Tree (OpExTree) example.

IV. Design and Experimental Setup

Our goal is to develop space-efficient structures for tracking metadata, specifically for predictive information. Ideally, these structures would incur little to no overhead while maintaining undiminished usefulness. Further, we seek to define, in a general case, what the expected benefits of these structures would be. Finally, we endeavor to verify our expectations by testing working implementations against realistic workloads in order to determine how effective our data pattern exploitation techniques would be at reducing



(a) *Dynamic Bitmap, key 1055*



(b) *Dynamic Region, key 1055*

Fig. 2. Dynamic Bitmap and Dynamic Region examples.

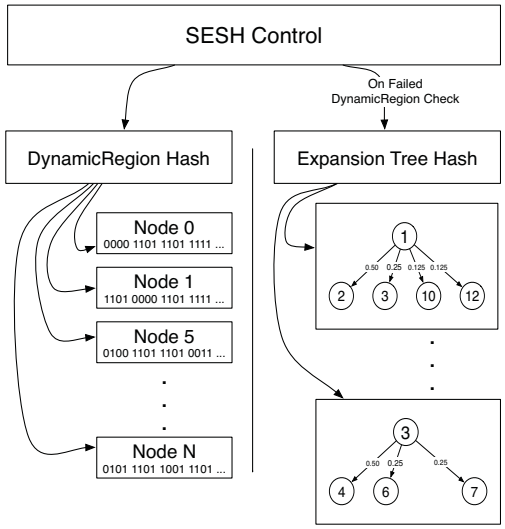


Fig. 3. SESH figure.

metadata volumes in real systems. This final goal can be met by testing our strategies on workload trace sets gathered from real systems, rather than drawing from a distribution or synthetic function.

This section describes our strategy by discussing the structures we have developed as well as an estimated reduction formula. We then detail the different trace sets used to evaluate our current implementations. Finally, we explain our metadata volume calculations.

A. Data Structures

Several new data structures were designed for this project. The *Optimal Expansion Tree*, or *OpExTree*, is the base structure used in our previous efforts for tracking metadata for predictive data grouping. The *Dynamic Bitmap* is a functional equivalent to a normal bitmap, but with the advantage of being dynamically allocated and able to spontaneously grow or shrink. The *Dynamic Region* is used to map a fixed number of bits to some id. Finally, the *SESH* structure is the combination of the above structures used to decrease the size of the necessary metadata required. Following is a brief discussion of each structure.

1) Optimal Expansion Tree: Our standard metadata storage structure consists of a root id, or the element's block number, and an array of immediate successors, or children. The structure is based on the *Recent Popularity* strategy from earlier work on predictive caching and prefetching [5], and was chosen for its robustness to signal noise and speed of adaptation to changing workloads.

Children are in the form of block numbers that occurred directly after the root id. While our structure allows this array to be unbounded, we limit the number of children for this project. Additionally, we track how often each child occurred.

Upon seeing a new event's successor, we add it to the tree by

1) Updating the appropriate count, or
2) Adding a new child to the successor array and setting the appropriate count to 1.

In the case of a bounded structure, once we reach the maximum number of children to track, we update the structure by choosing the lowest occurring successor and removing it from the structure. The new successor is then placed into the array and its count is set to 1. See Figures 1(a) and 1(b) for an example.

An alternate structure design replaces the successor array with a queue of children, in order of occurrence. Upon reaching the maximum capacity, a dequeue is performed before adding the new event. In this case, the counts are calculated by iterating through the queue on-the-fly. While this method will typically adjust to workload shifts easier, in practice we find the event counting to be a severe bottleneck.

A third alternate structure contains a queue as well as an array and counts. The queue is used in the same way as above, but dequeued items have their counts deducted, and are removed once their count reaches zero. In practice, we have found that our standard use of only an array very closely approximates this method, and the queue was removed from the standard version.

2) Dynamic Bitmap: The *Dynamic Bitmap* structure consists of a count of total number of entries and a hash table of nodes. Each node consists of a simple integer array that represents a region of the functional bitmap. Each Set, Unset, or Check of any particular

location is hashed and the appropriate node, if existent, is fetched. On a Set, the appropriate integer within the node's array is adjusted to update the map. If the node does not exist, it is created. Similarly, on an Unset, the appropriate integer is adjusted. If the Unset results in an empty node, equivalent to an array of all zeroes, the node is destroyed. On a Check, if the node does not exist, zero is returned. Otherwise, the appropriate bit within the existent array is returned.

As an example, assume we have a node consisting of 512 8-byte long long integers, and we are attempting a Check operation. The total number of entries in each node is equal to the number of bits; in this case, 32768, and each entry in the array, as an 8-byte integer, contains 64 bits. Given below are calculations of the node id, the array position within the node, and the bit location within the 8-byte integer. Note that all are integer division operations.

$$\begin{aligned} id &= key/total\_node\_size \\ ary\_loc &= key/single\_location\_size \\ bit\_loc &= key\%single\_location\_size \end{aligned}$$

In our example, $id = 1055/32768 = 0$, $ary\_loc = 1055/64 = 16$, and $bit\_loc = 1055\%64 = 31$. In this case, we calculate our id of 0, hash on that id to retrieve the node, if it exists. Assuming existence, we calculate the array location of 16, retrieve the 8-byte integer, calculate the bit location of 31, and perform a bit shift and bit mask to retrieve the value. Thus, the overhead of a single Check operation is a three integer division operations, a hash table retrieval, an array retrieval, a bit shift, and a bit mask, all of which are very efficient. See Figure 2(a) for clarification.

3) Dynamic Region: The *Dynamic Region* structure is very similar to a bitmap. Instead of each bit being used to represent some property of some event, a number of bits are used. This is achieved by utilizing a *Dynamic Bitmap*, and for each event id, we increment some region on the map. For our purposes, we required only that each region denote a count, or integer. All analogous operations follow easily from the Dynamic Bitmap structure. The only change needed is that we must multiply the key by the number of bits stored for each region. Using our example from earlier, assuming 3 bits per region, $id = 1055 * 3/32768 = 0$, $ary\_loc = 1055 * 3/64 = 49$, and $bit\_loc = 1055 * 3\%64 = 29$. See Figure 2(b) for clarification and comparison to the Dynamic Bitmap structure's analogous operation.

The overhead for the Dynamic Region is expected to be almost identical to the Dynamic Bitmap. Assuming that the region size is smaller than the number of bits in an array location, there are only two cases where significant differences occur. First, there can be an overlap of regions between two array locations, requiring an additional array position calculation and retrieval and additional bit location calculation. The other case involves node overlap, requiring and additional hash retrieval and array retrieval. Thus, in the worst case, we require two node id calculations and hash retrievals, two array location calculations and retrievals, and n bit shifts and masks, where n is the number of bits in each region. Note that all of these operations are expected to be very efficient, and that the number n is expected to be quite small, usually 3 to 5.

4) SESH, or Space-Efficient Storage of Heredity: During our work on prediction and data regrouping, we noted that many blocks have only a single successor. Most commonly, this successor happens to be the next block. The *SESH* data structure utilizes this observation by removing such *OpExTrees* from some successor table, typically a hash table, and utilizing a *Dynamic Region* to represent the tree. Some region being non-zero within the *Dynamic Region* structure represents a tree having only a single successor, which happens to be the block directly after the root block in question. We call the successors stored within the region *heir apparents*. These heir apparents occur the vast majority of the time, and each reduces the amount of metadata required from (minimally) several bytes to only a few bits (on average). See Figure 3 for clarification. As a realistic example, tracking eight successors (64-bit addresses, or 8 bytes) on a 256 GB hard drive with a block size of 512 bytes would require 32 GB of metadata.

$$8 * 8 * (256\ GB/512) = 32\ GB$$

However, each heir apparent would only require, on average, 3 bits. Given below is a estimated calculation for the reduced size, in bits, r, based on the number of blocks, b, the percentage of blocks that only contain heir apparents, p, and the number of successors tracked for each block, n.

$$r = b * (log(n) * p + (1 - p) * (64 + (64 * n)));$$

Note that this assumes 64-bit block numbers and ignores internal fragmentation within our *Dynamic Bitmap* structure. One note of interest presented by this formula is that when p is very high, the resulting size r becomes very scalable with respect to the number of successors, n. Since most blocks fit into the *Dynamic Region*, increasing n results in a $log(n)$ increase in the space necessary to store it. The larger structures increase linear to n. Even though these structures are

expected to represent only a small percentage of all items tracked, they *are* expected to dominate the space used fairly quickly. Figure 4 show a 3d plot of a 256 GB hard drive and the metadata required for storing information for all blocks, both before and after reduction, against the number of children tracked and the percentage of blocks that contain heir apparents.

The computational overhead of our *SESH* object's operations is expected to be quite small. The worst-case overhead is the sum of the worst-case overhead of a failed Dynamic Region operation and a hash retrieval of an OpExTree structure. However, most *SESH* operations will be a single Dynamic Region operation, as most blocks are expected to be heir apparents.

B. Traces

For this project, we used four different workload sets. The *mozart* set consists of a workstation trace gathered using the DFSTrace system [34]. These traces were converted into equivalent block-level traces with block sizes of 512, 4096 (4K), and 8192 (8K) bytes. There were four different original trace sizes; day length, week length, month length, and year length. This set has the appeal of allowing the analysis of our strategies over different definitive time periods as well as allowing us to convert easily to different block sizes.

The second set, *hplajw*, is a block-level workstation trace [35]. This set has the advantage of natively begin a block-level trace, and therefore does not require conversion. However, there is only a single trace length, and lacks any information of original file-system level access information, and therefore cannot be converted to traces of differing block sizes.

The third set, *ranin*, is a trace set we gathered using the standard fs_usage command found on Mac OS X. The traces were gathered from November to December, 2007 on a Mac PowerBook G4 running Mac OS X 10.4. The workload represents a typical graduate student workstation, and was used for internet browsing, file editing, code compiling, and running and testing experiments (predominantly C++ programs). While there were a few trace interruptions due to rebooting, including one major software update, the inaccuracies introduced would be negligible. Additionally, the software update had no impact on the fs_usage command itself, and any system-level workload shifts due to this update would represent realistic workload shifts experienced by users updating their operating system. Cache activity was gathered, but for these traces they were ignored; only device-level requests were used. These requests were in the form of read and write data and metadata as well as page ins and outs.

The final set, *playlist*, is a trace set gathered using the same fs_usage command. This set was gathered on two different Mac mini G4 workstations, each with 512 MB of memory and running Mac OS X 10.3.9. A playlist of 148 songs, with a runtime of approximately 14.8 hours, was run on each machine. Traces were gathered from August 31, 2008 to March 23, 2009, resulting in play counts over 300. All disk activity due to the mp3 software was isolated and recorded. One trace gathered information on a sequential playlist, while the other playlist was shuffled. These traces represent one extreme of predictability, an estimated upper bound on how predictable a realistic workload can be.

Similar to the *mozart* traces, our *ranin* and *playlist* workloads include information about how large an access was requested, and therefore could easily be converted to equivalent block-level workloads. Perhaps the most interesting block size is 512 bytes, which is the natively preferred block size of the hard drives, both for the PowerBook and the Mac minis. However, we included runs on 4K and 8K block sizes for consistency.

Since the fs_usage command collects information on *all* devices, these traces do require a bit of attention to what raw device is being accessed. Some devices, such as /dev/NOTFOUND, were pruned. All devices that seem viable were included in the test run and mapped to a single device. This mapping was done by giving a 200 GB range to each device. Table I summarizes the devices found in the *ranin* traces and how often each occurred, as well as noting which of these were ignored.

TABLE I. Devices found in *ranin* trace.

Device	Occurrence Count	Included?
/dev/disk0s3	3746392	Yes
/dev/NOTFOUND	571206	No
/dev/disk2s1	185933	Yes
/dev/disk2	73166	Yes
/dev/disk2s0	10621	Yes
/dev/disk1s1	954	Yes

All of these traces consist of data gathered from actual systems, and as such contain real-world predictability due to user, program, and system behavior, rather than being drawn from a distribution or synthetic function.

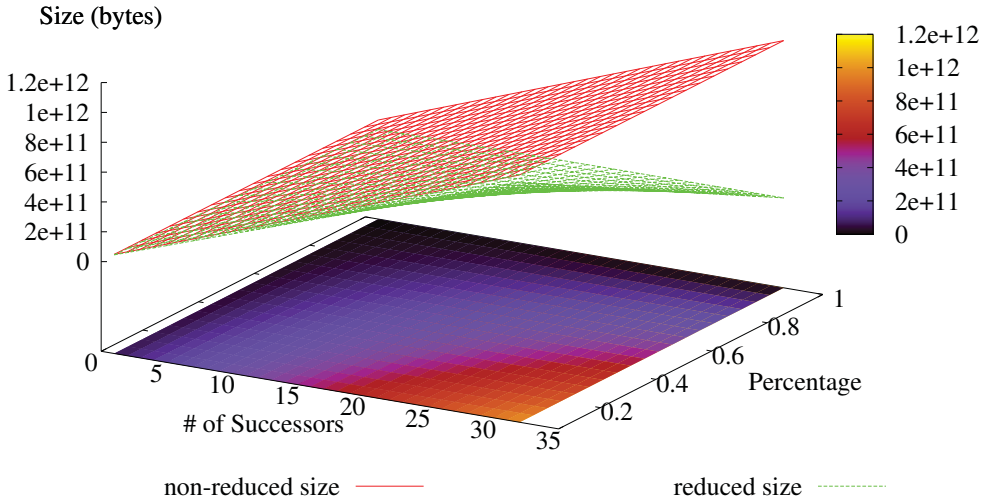


Fig. 4. Estimated calculation of metadata storage space savings on a 256 GB hard drive with a block size of 512 bytes.

C. Calculating Metadata Requirements

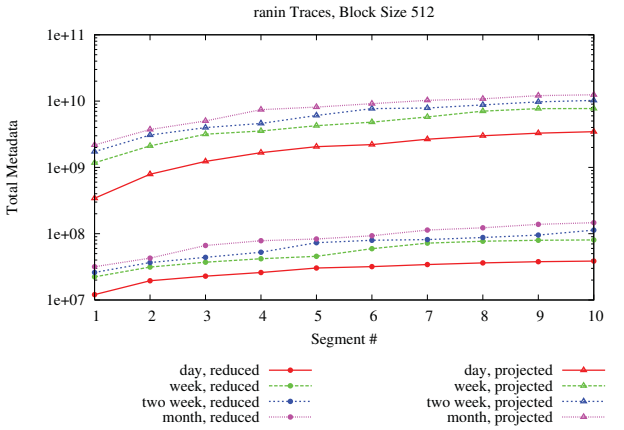
Each workload was split into ten sequential segments of approximately equal access counts. The trace was then run through our simulator. Each run consisted of the first segment, followed by running the first and second segments together, and so on until the entire trace was run. At the end of each segment run, the total metadata space used was recorded. Verification results on each individual segment were also run, but for sake of brevity are not reported.

Each recorded metadata requirement consisted of the calculation of total space used by our *SESH* structure. This includes any and all extra metadata we used for sake of statistics gathering, though these extra object fields are negligible. In calculating these metadata requirements, we count all nodes of all *Dynamic Bitmaps* used in our *Dynamic Regions*, rather than estimating a number of bits per heir apparent as in Figure 4. In order to calculate the projected size of metadata using a hash table of *OpExTrees*, we multiply the number of heir apparents by the total size of the same number of single-child *OpExTrees* and add the appropriate hash table metadata needed to track the extra trees.

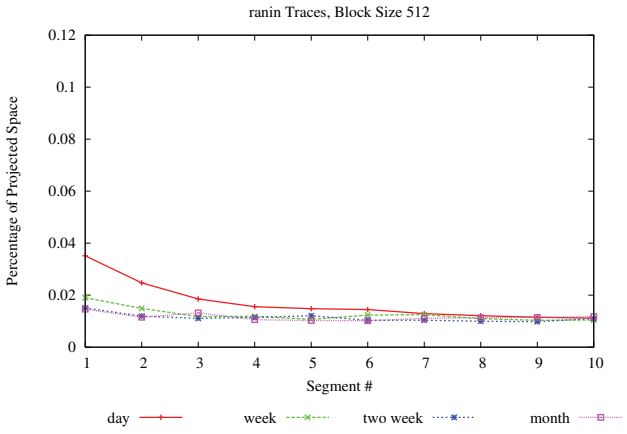
V. Results

Our results show that almost all traces of non-trivial size show a drastic decrease in necessary metadata. For most workloads, we can reduce this storage space to only a small percentage of the original space, typically between 1 and 3 percent for smaller block sizes. Table II summarizes the sizes recorded at the very end of the *ranin* workloads, while Table III summarizes the reductions and savings. Figure 5(a) illustrates the difference between the projected metadata requirements and the reduced space on the *ranin* traces with 512 byte blocks, while Figure 5(b) shows the reduced size in terms of projected volume's percentage. Figures 6(a) and 6(b) show the respective results for the *mozart* traces, again with 512 byte blocks. The *hplajw* trace showed results similar to these 512 byte block traces, with reductions falling between 91% and 97%. The interesting difference is that the *hplajw* trace does better early on, then quickly falls to 91% reduction before flattening out. The *playlist* traces showed reductions similar to the *ranin* workloads, exceeding 98% reductions for small (512 byte) blocks.

Figure 7 shows the amount of space that *SESH* requires for a selection of our traces, as a percentage of the total storage volume. Note that the total amount of

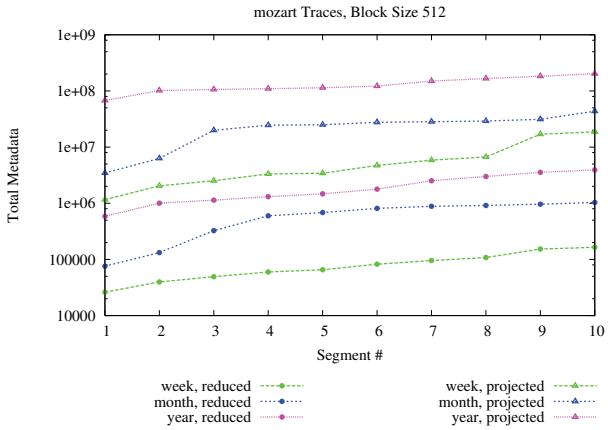


(a) *total space*

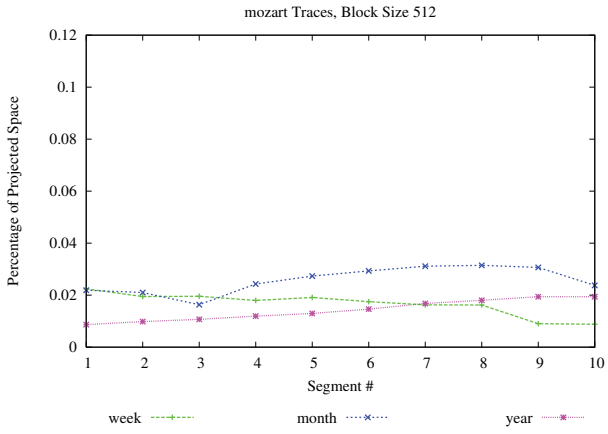


(b) *space reduction*

Fig. 5. Comparison of total projected metadata storage versus reduced storage for all *ranin* traces with 512 byte blocks.



(a) *total space*



(b) *space reduction*

Fig. 6. Comparison of total projected metadata storage versus reduced storage for various *mozart* traces with 512 byte blocks.

space across all traces and block sizes is less than half a percent. Also notice that the actual space required by SESH is higher than our estimate. However, this is not unexpected, as our implementations of OpExTrees keeps additional information than what is accounted for in our estimate.

As expected, larger traces show higher consistency in the necessary data required for storage. Smaller data sets would not adequately capture the larger picture, and would have new blocks introduced quite frequently, while larger sets would add only the occasional new block.

An interesting result to note is that total required storage space, after reductions, is reasonably consistent across block sizes, varying only by about 20%, while the total number of blocks that need tracked increase 14-fold. For instance, the full *ranin* trace, at roughly a month in length, requires about 150 to 189 MB, depending on block size, while the total number of blocks jumps from about 12 million (for 8 KB blocks) to 119 million (for 512 byte blocks). It is also interesting to note that, for *reduced* sizes, it is the middle block size of 4096 that requires the most space. As expected, the smallest block size has a much higher reduction rate, as it would exhibit a far greater amount of predictability, while the largest block size has far fewer blocks to track.

VI. Conclusions

In this paper, we have presented a simple yet novel strategy for tracking first-successor metadata information at the block level. We have provided an estimating

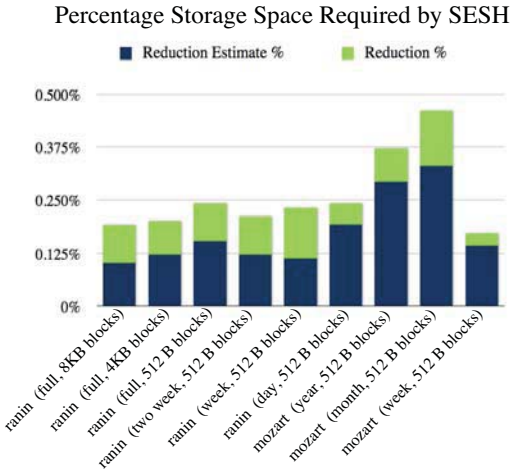


Fig. 7. Percentage of total storage volume that SESH requires, compared to the estimate.

TABLE II. Comparison of total space of all *ranin* traces. Block size, projected size, and reduced size are given in number of bytes.

Trace	Block Size	# Blocks	Projected	Reduced
day	512	33263953	3456650364	38585580
week	512	74221832	7712726204	80445044
two week	512	98709435	10257544012	113061012
full	512	119983696	12468052516	146756740
day	4096	4264887	436965644	42856668
week	4096	12033929	1232706268	96565836
two week	4096	16387433	1674557580	138910036
full	4096	22905900	2338128692	188674100
day	8192	2152146	217221140	42618692
week	8192	6271892	633318764	95528612
two week	8192	8586028	862968436	136909676
full	8192	12193122	1223027532	185049796

function for showing the expected savings, based on the number of successors that are tracked and the percentage of blocks that have the next sequential block as their only successor, called heir apparents. Estimates show savings of around 90 to 99% over total space usage from prior strategies for heir apparent percentages exceeding 90%. This space usage, according to our formula, also scales reasonably well, growing linearly in the number of successors tracked. These estimates were then verified with simulations tracking this successor information and calculating how much space is used by our implemented structures, both before and after our reduction strategy. Multiple workloads with multiple block sizes were run, each showing large

TABLE III. Comparison of reduction by percentage and savings of all *ranin* traces. Block size and savings are given in number of bytes.

Trace	Block Size	Savings	Reduction %
day	512	3418064784	0.9888372916
week	512	7632281160	0.9895698302
two week	512	10144483000	0.9889777697
full	512	12321295776	0.9882293775
day	4096	394108976	0.9019221108
week	4096	1136140432	0.9216635475
two week	4096	1535647544	0.9170467247
full	4096	2149454592	0.9193055110
day	8192	174602448	0.8038004404
week	8192	537790152	0.8491618796
two week	8192	726058760	0.8413503087
full	8192	1037977736	0.8486953146

reductions, verifying that heir apparent percentages are quite commonly very high. We have shown that, with an overhead of only a few hash table lookups, we are able to reduce the required metadata size up to 99%, and in all cases tested reducing required space to less than 200 MB, even for the largest workloads used. This first order successor information has been shown in previous work to be useful for prefetch prediction and caching as well as disk layout management.

VII. Acknowledgements

This work was supported in part by the National Science Foundation under Award #0720578.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *Trans. Storage*, vol. 3, no. 3, p. 9, 2007.

[2] D. Essary and A. Amer, "Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication," *Trans. Storage*, vol. 4, no. 1, pp. 1–23, 2008.

[3] T. M. Kroeger and D. D. E. Long, "The case for efficient file access pattern modeling," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*. Rio Rico, Arizona: IEEE, 1999, pp. 14–9.

[4] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, D. C., May 1993, pp. 257–266.

[5] A. Amer, D. D. E. Long, J.-F. Pâris, and R. C. Burns, "File access prediction with adjustable accuracy," in *Proceedings of 21st International Performance, Computing, and Communications Conference (IPCCC 2002)*. Phoenix, Arizona: IEEE, 2002.

[6] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer Technical Conference*, Jun. 1994, pp. 197–207.

[7] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, CA, Jan. 1997, pp. 275–88.

[8] T. M. Kroeger and D. D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, 2001.

[9] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, no. 5, pp. 771–93, Sep. 1996.

[10] E. Shriver, C. Small, and K. Smith, "Why does file system prefetching work?" in *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, Jun. 1999, pp. 71–83.

[11] S. H. Baek and K. H. Park, "Prefetching with adaptive cache culling for striped disk arrays," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 363–376.

[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'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2005, pp. 8–8.

[13] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: exploiting disk layout and access history to enhance i/o prefetch," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–14.

[14] M. Li, E. Varki, S. Bhatia, and A. Merchant, "Tap: table-based prefetching for storage caches," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–16.

[15] P. J. Denning, "Effects of scheduling on file memory operations," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 9–21.

[16] P. H. Seaman, R. A. Lind, and T. L. Wilson, "On teleprocessing system design part iv: An analysis of auxiliary storage activity," *IBM Systems Journal*, vol. 5, no. 3, pp. 158–170, 1966.

[17] A. G. Merten, "Some quantitative techniques for file organization," Ph.D. dissertation, 1970.

[18] L. Huang and T. Chiueh, "Implementation of a rotation latency sensitive disk scheduler," Tech. Rep. CS-TR-283-90, 2000.

[19] L. Reuther and M. Pohlack, "Rotational-position-aware real-time disk scheduling using a dynamic active subset (das)," in *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 2003, p. 374.

[20] D. M. Jacobson and J. Wilkes, "Disk scheduling algorithms based on rotational position," Concurrent Computing Department, Hewlett-Packard Company, Tech. Rep. HPL-CSP-91-7, 1991.

[21] C. Ruemmler and J. Wilkes, "Disk shuffling," Tech. Rep. HPL-CSP-91-30, October, 1991.

[22] S. D. Carson and P. F. R. Jr., *Adaptive disk reorganization*. Technical report UMIACS-TR-89-4 and CS-TR-2178. Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, and Department of Computer Science, University of Virginia, 1989.

[23] G. Hardy, J. Littlewood, and G. Pólya, *Inequalities*, 2nd ed. Cambridge University Press, 1952, reprinted 1983.

[24] C. K. Wong, *Algorithmic Studies in Mass Storage Systems.* Computer Science Press, 1983.

[25] T. J. Gibson, "An improved long-term file usage prediction algorithm." 1999.

[26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-aligned extents: Matching access patterns to disk drive characteristics." 2002.

[27] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger, "Atropos: A disk array volume manager for orchestrated use of disks." 2004.

[28] G. R. Ganger and M. F. Kaashoek, "Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files," in *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, CA, Jan. 1997, pp. 1–17.

[29] C. Staelin and H. Garcia-Molina, "Clustering active disk data to improve disk performance," Department of Computer Science, Princeton University, Tech. Rep. CS-TR-283-90, Feb. 1990, revised June 1990.

[30] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–97, Aug. 1984.

[31] K. A. Smith and M. Seltzer, "A comparison of FFS disk allocation policies," in *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, Jan. 1996, pp. 15–25.

[32] S. Akyürek and K. Salem, "Adaptive block rearrangement," *ACM Transactions on Computer Systems*, vol. 13, no. 2, pp. 89–121, 1995.

[33] E. Shriver, E. Gabber, L. Huang, and C. Stein, "Storage management for web proxies," in *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, Jun. 2001, pp. 203–16.

[34] L. Mummert and M. Satyanarayanan, "Long term distributed file reference tracing: Implementation and experience," *Software - Practice and Experience (SPE)*, vol. 26, no. 6, pp. 705–736, 1996.

[35] C. Ruemmler and J. Wilkes, "Unix disk access patterns." in *USENIX Winter Technical Conference*, 1993, pp. 405–420.