Untangling Block Allocation Policies of ZFS

Swaminathan Sundararaman
Sriram Subramanian

Computer Sciences Department
University of Wisconsin, Madison

December 21, 2007

Abstract

The ZFS file system from Sun is the latest buzz word in the file system community. The creators of ZFS claim to have re-designed the file system from scratch, providing new features and levels of reliability, performance and efficiency uncommon in traditional file systems. This includes dynamic block allocation that changes the blocks sizes based on workloads. In this paper we have primarily focused on the Block allocation policy of ZFS under varied workloads. We have built our infrastructure based on semantic block analysis and found that ZFS allocates block based on the file offset that are being written and not based on the workload. Block allocation policy works poorly for random writes. We also found that ZFS merges smaller blocks to one big block and as a result a single block write gets converted to read-modify-write of a bigger block.

ZFS Intent log also has a poor block allocation policy and for small block writes it wastes a significant amount of storage space in the file system. Overall we find that ZFS tells more that it actually does.

1 Introduction

ZFS is the latest buzz word in the file system community. The developers of ZFS claim that its the last word in file systems [10]. Sun actively advertises ZFS as their next generation file systems. ZFS also had been ported to OSX 10, Linux (through FUSE), FreeBSD, and NetBSD.

ZFS has been designed to provide simple administration, transactional semantics, end-to-end data integrity and immense scalability. Sun also claims that ZFS is not an incremental improvement to existing technology; it is a fundamentally new approach to data management. They have blown away 20 years of obsolete assumptions, eliminated complexity at the source, and created a storage system that is actually a pleasure to use [10].

Like other modern file systems there is no good documentation on the policies of ZFS. We are still in the dark about how ZFS provides these new features. Even after a wide scale acceptance of ZFS by other operating systems only a very high level documentation is provided by Sun Micro Systems [10] and few random blogs by ZFS developers exists [6, 7].

In order to uncover some of the policies of ZFS we have used gray-box techniques [1]. Gray box technique is an approach where the tester has information about some of the inner workings of the system. We have specifically used Semantic Block Analysis (SBA) [8] . SBA was used to uncover polices of modern journalling file system such as Ext3 [11], JFS [5], and ReiserFS [9]. SBA is a gray box approach where the tester has information about the on-disk layout and data-structures of the file system. In SBA, the tester runs a customized workload and observes the disk workload by inserting a pseudo device driver between the file system and the disk. By co-relating the workload with the observed disk traffic SBA uncovered the journalling policies of the file systems.

Unlike other file systems, ZFS allocates blocks of varying sizes based on the workload. Hence, we were specif-

ically interested in finding out the block allocation policy of ZFS. Also, ZFS never overwrites any block on the disk, it always performs Copy-On-Write of blocks whenever they are over written. We wanted to find out how ZFS reacted to synchronous workloads as it would have to create new copies of all block at the leaf level up to the root (or the uberblock). We take the SBA approach to uncover the policies of ZFS by inserting a pseudo block-device driver between ZFS and the disk. We ran specific workloads that exercised some of ZFS's allocation policies and analyzed the disk traffic to infer the internal block allocation policies.

From our analysis we found that ZFS has a naive block allocation policy and does not dynamically change block sizes on workloads. Contrary to what ZFS developers claim, we found that the block allocation policy of ZFS is purely based on the file offset to which a new block is written to. Also, ZFS block allocation policy performs poorly for random workloads.

ZFS dynamically re-organizes small blocks into one single large block. As a consequence of this policy ZFS suffers from reading additional blocks under some workloads. To be precise, ZFS reads the contents of the smaller blocks even if blocks are not in memory, to merge then into one single large block for small files. The dynamic re-organization policy was not observed for large files.

From our analysis of ZFS's performance on synchronous workloads, we find that the current block allocation policy for ZFS Intent Logs is very inefficient. ZFS wastes on-disk space for the logs by allocating blocks of larger sizes (an additional 4K block in most cases) than required. It also wastes a significant amount of on-disk space under some workloads.

The rest of this paper is as follows. In section 2 we talk about the block allocation mechanism in ZFS and also discuss the mechanism in ZFS to handle synchronous workloads. Section 3 explains the infrastructure that was used to uncover the block allocation policies of ZFS. In Section 4 we discuss the block classification strategy that we used to segregate blocks from the observed disk traffic. Sections 5 and 5.5 explains our workloads, analysis and inferences on ZFS's block allocation and dynamic reorganization strategy. Section 6 explains our workload that helped in uncovering the ZIL block allocation policy along with the analysis on the observed disk traffic. Section 7 talks about the related work and conclude in Section 8.

2 Background

Traditional block allocation and free space management strategies like bitmap and btrees don't scale well and perform poorly when subjected to random frees (as they lack locality). With a 128-bit file system like ZFS, these become bottlenecks for performance. So instead, ZFS takes an approach similar to the Log Structured File system. As blocks are freed, they get added to a list of recently freed blocks and this list represents the free space available. The space on the virtual devices are divided into Meta-slabs, each having a space map that represents the free space available. The space map is a time ordered log of allocations and frees. When space is freed, the extent gets appended to the space map, and each allocation is represented as an extent in the space map. The space map and its associated allocations and frees are maintained in memory in the form of an AVL tree of free space sorted by offset. By reconstructing the space map in memory can also help in compacting the space map.

2.1 Dynamic Block Allocation

ZFS also has a dynamic block allocation policy. Unlike traditional file-systems which have their block sizes fixed at format time, ZFS can either dynamically select block sizes depending on the nature of the workload or allow the user to specify one (for example, for applications like databases which have fixed size records, its beneficial to fix the block size manually). Block sizes are allocated dynamically up-to a 128k after which it remains constant.

2.2 ZFS Intent Logs

In certain applications, like databases, certain disk writes are synchronous. For example, when the commit records get flushed to disk, the database forces this to disk and waits for the write to complete before it can proceed. In these cases, ZFS would perform poorly with its copy-on-write semantics as each synchronous write would require all the blocks in the hierarchy up to the uber-block to be re-written before the write system call can return. Instead ZFS chooses to use ZIL - ZFS Intent Logs. ZILs

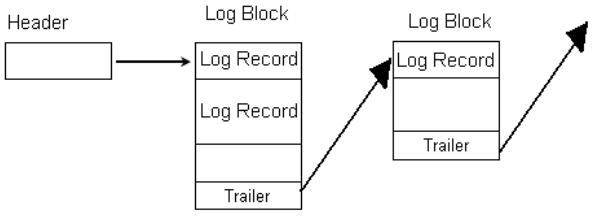

Figure 1: **ZIL Block Chaining.** *This figure illustrates the ZIL Block Chaining mechanism. Each ZIL block has a pointer to the next ZIL block. During crash recovery, the ZIL Blocks are replayed starting from the ZIL Header can be reached from the uber-block*

accumulate in memory and when some data needs to be synchronously written, then the ZILs are flushed to disk. These can be later replayed in the event of a crash. ZFS writes the data along with the log for data block size up to 64KB. Beyond 64K it synchronously writes the data block to the disk and blocks pointer to the log record.

ZIL consists of a ZIL header, ZIL blocks and ZIL trailer. The header block is the starting point from which the logs have to be replayed during crash recovery. The current or the latest uber-block contains the pointer to the ZIL header. The header points to a list of chained ZIL blocks 1. The trailer (which is present in each ZIL block) is responsible for establishing the chains. This raises the interesting question of how these chains are maintained. When a ZIL block gets flushed to disk, the blocks trailer should contain the location of the next block. So ZIL blocks are preallocated and their addresses chained. Preallocation results in wastage/internal fragmentation of ZIL records. At the moment blocks are preallocated the size of the current block is the only metric to go by to determine the size of the next block. So if the current block is 33k in size, then the next block is also 33k long.

The ZIL blocks are dynamically allocated and there is no fixed location for these blocks, which translates to no pre-defined limitation on the size of ZFS logs.

3 Infrastructure

We ran all our tests on a Sun ultra sparc 20 workstation with 1 GB of memory and two Ide disks each of 75GB in size. We installed the latest Solaris community version available from the sun's website (build 70b). In order to capture the block traffic between the file system and the disk we have implemented a pseudo block-device driver (PBDD) in Solaris 11 using Layered Device Interface (LDI). The LDI layer in Solaris provides a cleaner interface for developers to implement pseudo device drivers and hides all the complex details of implementing the hooks to access and translate requests to the underlying device. During implementation we found that Solaris did not allow ioctls to block devices that are not registered as a block device. We desperately needed Ioctl support for our device driver to control collection of statistics inside the driver. In order to overcome the limitation we used the popular solution of adding another level of indirection between the file system and our pseudo block device driver. We implemented a dummy pseudo-character device driver that opened our pseudo block device driver and redirected the ioctl requests. Later on we found that Solaris allowed a particular device to be exported as both character and a block device at the same time, which would have avoided the additional transfer of ioctl messages from the character driver to the block driver and vice versa.

Our block driver does selective classification of blocks based on the flags set in the driver through ioctl calls. After classification each block the PBDD asynchronously writes a log record about the analysis to a log file. The format of the log records are discussed in the following section.

4 Block Classification Strategy

In order to co-relate the block traffic that reach the disk with the workload, we devised a simple yet powerful way of identifying different ZFS blocks. ZFS blocks can be broadly classified as Uberblock (aka super-block in FFS like file systems), ZIL blocks, data blocks and meta-data blocks. The meta-data blocks are basically Meta-Object Set (MOS), which can contain sets of meta-objects. Objects sets in ZFS are used to group related objects such as objects in a file system, clone, snapshot, and volume. For our workloads we did not have to look into the meta-data structures of ZFS. We observed that ZFS first compresses and writes the meta-data blocks before writing back to the disk. The block classification methodology for uber, ZIL,

data and meta-data blocks are explained in section 2.2.

4.1 UberBlock

ZFS writes a new uberblock whenever it wants to create a new persistent version on the disk. The uber-blocks are 1024bytes in size and has a 64 bit magic flag to identify the uber-blocks. The value of the magic flag is 0x00bab10c (oo-ba-block). PBDD exploits this information to identify uber-blocks by its magic flag which is written at a constant offset in each uberblock.

4.2 ZIL Blocks

ZFS does not overwrite any blocks on disk. It always does Copy-On-Write (COW) in order to write back modified disk blocks. The COW mechanism causes problems for ZFS for synchronous workloads as it needs to write back a chain of block starting from the data block (leaf node) up to the uberblock (root node). This would bring down the performance of ZFS for synchronous workloads. To over come this problem ZFS writes the data to the log file and flushes the logs back to the disk before returning back to the user. This way the data is persistent (even if a crash occurred before the data block is written to its new location). The ZIL blocks are identified by the magic flag which is present in the ZIL trailer structure of every ZIL block. The ZIL trailer block contains a zio_block_tail structure which contains the magic flag field. In short, the magic flag is always written at a constant offset from the end of every ZIL block. PBDD uses this information to identify ZIL blocks.

4.3 Data Blocks

Majority of the block that gets written in most of the workloads are data blocks. In order to identify data blocks we add a special 64 bit pattern at the start of every 512 byte offset. The 512 byte offset is chosen because its the smallest block size that can be written to the disk. In order to identify individual data blocks an unsigned long long integer was added after the special pattern. The application that generates the workload maintains a counter that keeps track of number of 512 byte blocks written to the disk. The application appends this value after the special pattern for every 512 byte block. PBDD uses this information to identify individual blocks within a large data block. These sequence numbers are very useful to detect if duplicates blocks are written back to the disk and also for co-relating ZIL blocks writes with the actual data block that is written after a brief interval of time.

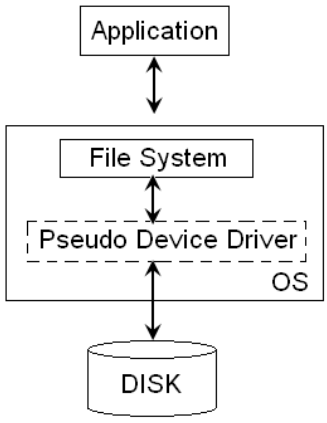

Figure 2: **Experimental Setup.** *This figure shows the higher level setup of our experiments. The applications run customized workloads and the pseudo device driver observes the disk traffic from the file system generated by the applications. It then tries to infer policies from the generated block traces.*

4.4 Meta-data Blocks

The blocks that are not classified as uber, ZIL or data blocks are automatically classified as meta-data blocks. As mentioned before our analysis did not require understanding of the meta-data blocks. Hence we do not uncompress and extract information from the meta-data blocks.

5 Block Allocation Policy

Our first goal was to untangle the block allocation policy in ZFS. ZFS claimed that blocks of various sizes are dynamically allocated based on the workload. We were curious to know how exactly blocks were allocated in ZFS.

In this section we first describe each experiment, its goal, inferences from each experiment.

5.1 Sequential Writes to a Large File

In our first experiment we ran a sequential workload that wrote blocks of varying sizes asynchronously to the disk in each run. The goal of this experiment was to find the default block size in ZFS. We varied the block size in each run from 4KB to 1MB. In runs where the application wrote block sizes that are lesser than 128KB, we observed that ZFS cached subsequent block writes in memory till the cached block size reaches 128KB. ZFS then writes a single 128KB block instead of writing many blocks of smaller size. This also helps ZFS to read a single block while fetching the data back from the disk making it more efficient by reducing the fragmentation of blocks. Block writes greater than 128KB were still written in multiples of 128KB blocks by ZFS. From this experiment we concluded that the maximum size of blocks in ZFS was 128KB.

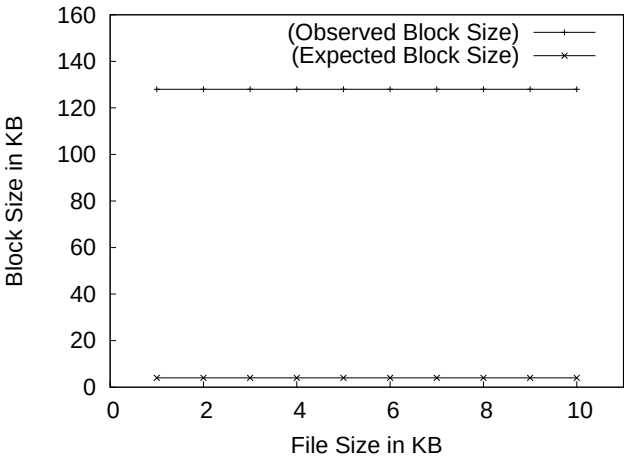

Figure 3: **Impact of Large Random Workloads on ZFS.** *The figure shows the effect of random 4K blocks written inside a file of size 4GB. This graph shows that ZFS always allocates blocks of 128K even when the application performs 4KB writes.*

5.2 Random Writes to a Large File

In our next experiment we wrote 4K blocks to a large file by randomly seeking to different offsets (the maximum seek offset was set to 4GB) within the file. We were expecting blocks writes of 4KB sizes but we observed that ZFS wrote blocks of 128KB for each 4K block write. This pattern of writing larger block sizes continued till block sizes were less than 128K in size beyond which it always wrote blocks of 128KB in size. This was also observed in the previous experiment where we sequentially wrote blocks of varying sizes to a file. Figure 3 shows the results for block writes of 4KB size by the application. The bottom line shows the expected block sizes (in this case 4KB) but the top line shows the observed block size (128KB). From this experiment we conclude that ZFS block allocation policy does not work really well for small random writes to a large file.

5.3 Random Writes to a Small File

In order to get a clear understanding of how ZFS allocated blocks we wrote 4KB blocks to small files (maximum seek offset was set to 128KB). Figure 4 shows one such run. Table 1 contains the offsets generated by the applications and the block sizes observed. From table 1 we can see that when the first block is written at 36K offset the block written by ZFS was 40K (36K plus the additional 4K data). The block sizes remain the same for the next three writes of 4K blocks as offsets generated were less than 40K. For the forth block which was written at 84K byte offset the block size generated by the file was 88KB. From this experiment we see ZFS does not do a good job in allocating blocks even for small random writes to files whose sizes are lesser than 4KB in size.

From the previous experiment an observant reader would have noticed that the block size generation could be very naive in ZFS. It could just be based on file offsets. In order to verify our assumption we wrote 512 bytes at different offsets to a small file. Figure 5 shows the block sizes allocated by ZFS when 512 bytes were written to a file at various offsets. Table 2 contains the offsets and the block sizes generated by ZFS. From Figure 5 we can clearly see that the block allocated by ZFS are directly proportional to file offsets the blocks are written to.

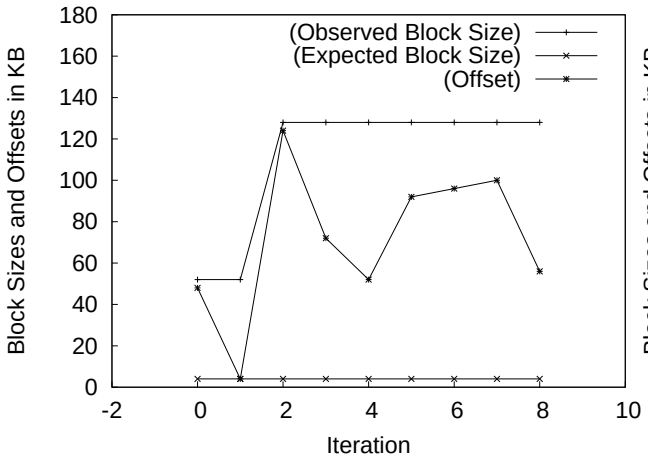

Figure 4: **Impact of small random Workloads on ZFS.** *The figure shows the effect of random 4k byte blocks written inside a file of size 128KB. This graph shows that ZFS always allocates blocks based on file offsets.*

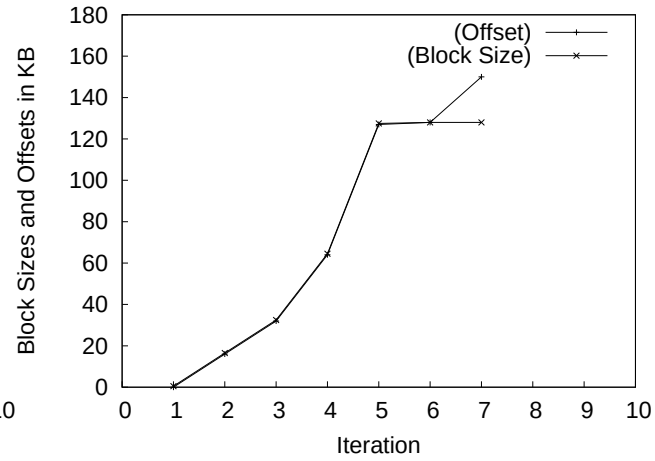

Figure 5: **Random of 512 bytes to a small file.** *The figure shows the effect of random 512 byte blocks written at various file offsets. This graph clearly shows that ZFS block allocation policy is purely based on file offsets.*

Offsets in KB	Observed Block Sizes in KB
36	40
36	40
20	40
84	88
0	88
20	88
52	88
16	88
4	88

Table 1: File Offsets Vs Observed Block sizes for Random 4k write to a small file

Offsets in KB	Observed Block Sizes in KB
0	0.5
16	16.5
32	32.5
64	64.5
127	127.5
128	128
150	128

Table 2: File Offsets Vs Observed Block sizes for Random 512 byte writes to a small file

5.4 Block Allocation Policy

From the previous experiments it can be seen that ZFS has very naive block allocation algorithm. The block sizes are allocated based on the file offsets that the blocks are written to. For file sizes that are lesser than 128K the block size allocated to the file is the offset to which the block is written to. For files larger than 128KB a new block of 128K is allocated invariant of the amount of data written to the file. This block allocation policy does not work for random write workload.

5.5 Dynamic Block Reorganization

While running our experiments we observed to ZFS smartly merges small blocks to one single large blocks. In order to understand this block re-organization policy of ZFS we designed a few workload that would help us get a better insight into it. We made sure the block written to a file reaches the disk before the next write to the file proceeds. This was achieved by making the application sleep for a small duration of time (we observed that

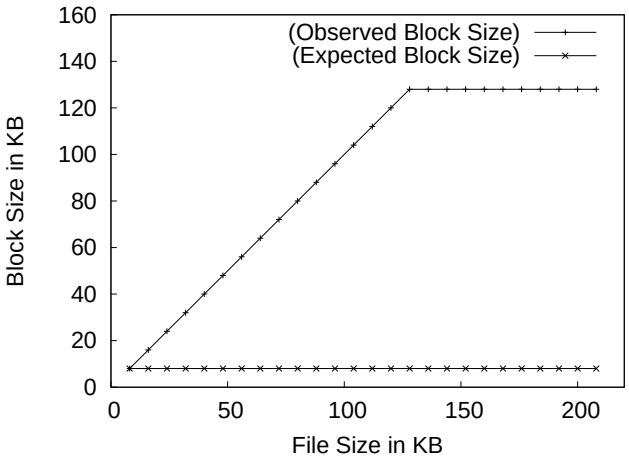

Figure 6: **Effect of Small Sequential Writes on Block Allocation.** *This graphs shows the observed block sizes writing 8k blocks at regular time intervals.*

ZFS wrote back the dirty blocks to disk ever 10 seconds, hence we made the application sleep for 12 seconds before every write). In our first experiment we wrote blocks of 8KB sizes sequentially to newly created file. As mentioned before we forcefully introduced a 12 second delay between consecutive writes to the file. Figure 6 shows the block sizes observed by our PBDD. It can be seen that observed block sizes are significantly different from expected blocks sizes. This is because ZFS merges the previously written small block with the newly appended data and writes back one single block. The linear graph for observed block sizes show this policy of merging smaller blocks of a file into a larger block for every subsequent writes to a file. ZFS merges blocks till the file size reaches 128KB and after that it always allocated and blocks of 128KB. We varied the block sizes written to the file from 4K to 128K and observed that ZFS merged blocks in all the cases whenever the total file size was lesser than 128KB.

In order to explain this more clearly we show the observed block sizes when 32KB blocks were written to a new file. From figure 7 we can see that when the first 32K block was written, the observed block size is the same as the expected block size (i.e, 32KB). When the next block is written, the expected block size is 32KB but since ZFS merged the previously written data with the new data that is appended to the file it wrote a 64KB block instead of a new 32KB block for the second write. We identified dynamic merging of blocks in ZFS as we added a counter to every data block that is written to ZFS. In the above mentioned experiments the previously observed counter numbers (i.e 0 to 64 as the we increment the counter for ever 512 byte block) were repeated even during the subsequent block write and the block size for the write was 64KB. When the third block was written to the file the observed blocks sized increased from 64K to 96K. This is because it merged the previously written 64K with the new 32K append and wrote one 96K big block to the disk. For the fourth block the observed block size increased from 96K to 128K in size. For the subsequent block writes the block size remains a constant as the file size is larger than 128KB.

In is important to differentiate these 128K block writes with the first 128K block observed in PBDD. The first 128K was written by merging the previously written 96K with the new 32K write to the file. Where as the next 128K block only contained 32K of data in it and the next subsequent write to this 128K block resulted in a COW to this 128K block. We can see that ZFS wastes a significant amount of space for small block writes to file that are greater than 128KB. Even though the previously written blocks would be reclaimed at a later time the blocks would not be available until a background ZFS process checks these blocks to reclaim them back.

Now that we have understood the block merging/reorganization policy of ZFS. We wanted to check if ZFS does this in a smart way. i.e., merge block only if they were in memory as it would be inefficient to read back the previously written block before appending the newly written data to the existing data and writing back a larger block to the disk. We devised a new workload to check how ZFS reacted in this case. In order to observe this the previous written block should not be in memory. We unmounted and remounted the file system between subsequent writes. Another was of achieving this would be have a large number of small files whose combined size would exceed the size of the memory and then subsequent writes to the files in round robin fashion would ensure that the previous blocks would no longer be cached in memory. We chose the first approach as it was much easier to implement. Also, the file we reopened again in O_APPEND mode to ensure ZFS did not read

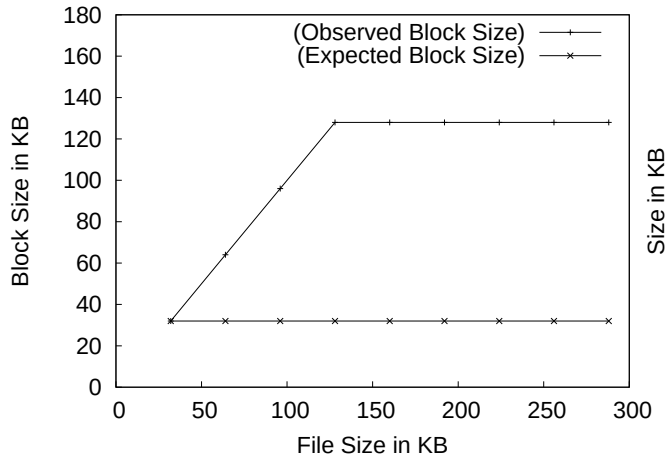

Figure 7: **Effect of Small Sequential Writes on Block Allocation.** *This graphs shows the observed block sizes writing 32k blocks at regular time intervals.*

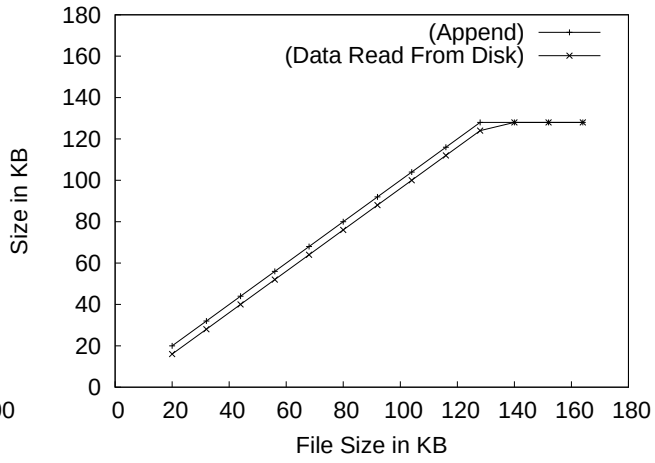

Figure 8: **Block re-organization in ZFS.** *This graphs shows sizes of block read from the disk while appending 4K blocks to a file. Before appending a block, the file system is unmounted and remounted again to ensure that the previously written blocks are not cached in memory.*

back the data to cache it in memory. In this experiment we appended 4KB block to the file, slept for some time to ensure that blocks reaches the disk. Unmounted and remounted the ZFS file system. Finally opened the file in O_APPEND mode and repeated the above mentioned steps.

From Figure 8 we can see that during block writes, ZFS reads back the previously written data to the file to append the newly written data to it and allocates and write back a larger block back to the disk.

From all these experiments we observe that ZFS dynamically resizes the block till it file size is smaller than 128K after which its always allocates 128K and performs copy-on-write on these blocks for writes of smaller block size. It was also observed that the dynamic reorganization policy would have terrible performance impact when the smaller blocks are not in memory and ZFS would suffer from small appends being converted to Read-Modify-Write of blocks to the disk. This is similar to the performance problem in RAID-4 where the parity block has to read back from the disk to recompute the parity and is written back with the new parity for every small block write to the RAID.

6 ZIL Block Allocation Policy

ZFS writes intent logs for blocks that are synchronous written by means of *fsync* call or opening the file *O_DSYNC* mode. In order to avoid writing block from leaf to the root, ZFS writes the data to the log before returning to the caller. The goal of the following experiments was to find out how ZFS allocated ZIL blocks when the blocks sizes varied. We also wanted to find out how large synchronous block writes were handled by ZFS.

In the first workload we opened a file in O_DSYNC mode and wrote blocks of a fixed sizes till the file size reached 1MB. In each run of the experiment we varied the block size from 512 bytes to 512KB. Figure 9 shows the ZIL block allocated for a few different block sizes written by the application. It can be seen that for block sizes lesser than 4KB, ZFS allocates 4KB sized ZIL blocks. To be precise we observed this pattern till block sizes were lesser than 4KB - 192 bytes. This is because ZFS also writes some additional bytes that helps in replaying the log information. The 192 bytes is the log record size for writes and it varies with the different log record types (e.g, create, delete, rename, access control information). We observed that ZIL added the data after the log records

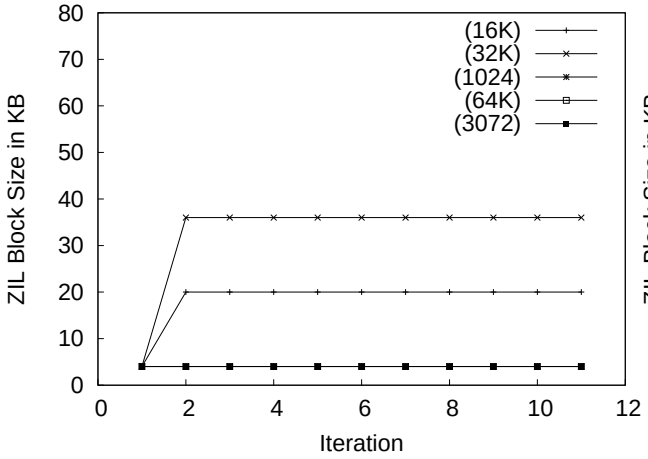

Figure 9: **Effect of Block Writes on ZIL Block Sizes.** *This graph shows the observed ZIL block sizes Vs Data block Sizes. From the graph we observe that ZFS always allocates blocks in multiples of 4K.*

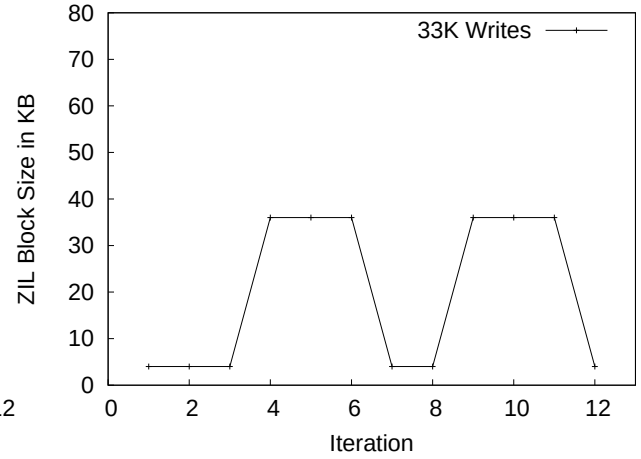

Figure 10: **ZIL Block Allocation for 33KB Synchronous Writes.** *This graphs shows the observed ZIL Block sizes for 33K synchronous writes. ZIL Performs poorly because of block chaining, re-ordering of blocks and copy-on-writes.*

till the block sizes were lesser than 64K after that the blocks were flushed to the disk and only the pointer to the block were written to the log record. We believe this is a good policy when the block sizes get larger as you have to rewrite (or copy) the data blocks again to its original position.

During the tests we also observed that ZFS always flushed the first data block to the disk invariant of the block size being written and creates a new on disk version (by writing a new uberblock). We were puzzled why ZFS only writes the first data block to the disk and from then on uses the ZIL blocks to write data to it. When we looked at the source code we found that that it pre-dirties the first block so that subsequent writes to blocks that are written synchronously to disk sync to convergence faster.

We found that ZIL log chaining performs inefficiently under some workloads (especially when the block sizes are between 32k and 64k). Figure 10 shows the problem with ZIL block allocation and with log chaining. As mentioned in section 2.2, ZIL blocks are chained (i.e., each ZIL block contains the pointer to the next block) and ZFS never overwrites any block on disk. As a result it has to pre-allocate the next block when the current log block is written. The first ZIL block observed is the ZIL header block. Even though the data contained in the header is less than 1K of data it still writes 4K blocks to the disk. In this particular experiment ZFS writes 33K block sizes the first three 33K writes does not generate ZIL blocks. This is because the first 33k data block does not get written in ZIL it is directly flushed to the disk. When the second 33k write occurs due to dynamic re-organization policy in ZFS it merges this block write with the previous 33K block to write one large 66K block. Since this block size is greater than 64K ZFS flushes the data-block and writes the block pointer in the log record. Even though the total data in the ZIL block is less that 1K ZIL writes a 4K ZIL block as the minimum block size of ZIL is 4K. During the third 33K block write ZFS merges the current data block with the previous written 66K data and writes a 99K block. Once again as the data is greater than 64K, ZIL only stores the block pointer to the newly written data block.

When the next 33K write occurs the amount of data crosses the 128K block boundary, hence ZFS writes back the data block of 128K size and also a log block that contains the data written to first 128K block. It writes the remaining 4K as a separate log entry in the same ZIL block. For the fifth 33K write we observe that its still writes 128K block (i.e, it does a COW for the previous created 128K block). This pattern keeps repeating. When

the pattern repeated again the ZIL does need to be created again.

We also some more strange behavior for which we do not have a logical explanation. The seventh ZIL log block is the copy of the second log block. Also, the twelfth ZIL log record is the duplicate of sixth ZIL block. This pattern keeps repeating. Also, when the new 33K is appended to the file whose size is greater than 33K it ZFS flushes the 128K data block to the disk but still writes the data to the log when the write overlap two 128K blocks of the file.

7 Related Work

Traditionally file systems have been benchmarked using specialized workloads. Some of the popular file system benchmarking utilities are PostMark [4], IOzone [12], Bonnie [2] , and Andrew benchmark [3]. IOzone benchmarks perform synthetic read/write tests to determine throughput. Andrew and Postmark benchmark are designed to model realistic application workloads. All the above mentioned benchmarks measure overall throughput or runtime to draw high-level conclusions about the file system. In contrast to our approach of using SBA, none of these are intended to yield low-level information about the internal policies of the file system.

8 Conclusions

Semantic block analysis provides a powerful method to analyze and extract policies of file systems without actually looking at the file system code. We found that SBA helped us to accurately figure out the block allocation policies of ZFS. We found that ZFS currently has a very poor block allocation policy. It allocates blocks based on the file offsets the block is written to, till the file size is less than 128k and allocates blocks of 128k for any block write greater than 128k offset. This was true even when the smaller block is not in memory. It was surprising to see that when ZFS appends data to a smaller file, it read the previous block back to the memory, then appends the new data and writes a larger block to the disk. We also found that ZFS constantly merges smaller blocks into a larger block till the file sizes reaches 128KBytes. The ZIL block allocation is also poor in its current form. It creates and writes blocks in multiples of 4k blocks and due to this it ends up writing blocks of larger sizes that necessary. We only have touched upon the tip of the iceberg and more analysis needs to be done to other policies of ZFS. Some other interesting features that would be worth investigating are versioning policy, meta slab, and RAID-Z.

9 Acknowledgment

We would like to thank Prof. Remzi for his guidance and for his constant encouragement to break his machines. We would like to thanks Lakshmi and Nitin for helping us fix the broken machines. Finally we would like to thank Sun Micro Systems for donating Sun Ultra-20 workstation to ADSL group for their research projects.

References

[1] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[2] T. Bray. The Bonnie File System Benchmark. www.textuality.com/bonnie/.

[3] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[4] J.Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Applicance Inc. www.netapp.com/tech_library/3022.html/, October 1997.

[5] D. Kleikamp and S. Best. How the Journaled File System handles the on-disk layout, May 2000. www-106.ibm.com/developerworks/library/l-jfslayout/.

[6] N.Nadgir. Neelkanth Nadgir's Blog on ZFS, ZIL etc. www.blogs.sun.com/realneel/.

[7] N.Perrin. Neil Perrin's Blog on ZFS, ZIL etc. blogs.sun.com/perrin/.

[8] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[9] H. Reiser. ReiserFS. www.namesys.com/, October 2004.

[10] Sun Micro Systems. ZFS - Open Solaris Community. www.opensolaris.org/os/community/zfs/, 2007.

[11] T. Y. Ts'o and S. Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–243, Berkeley, CA, USA, 2002. USENIX Association.

[12] W.Norcutt. The IOZone Filesystem Benchmark. www.iozone.org/.