

5-2021

Performance Implications of Memory Affinity on Filesystem Caches in a Non-Uniform Memory Access Environment

Jacob Adams

Follow this and additional works at: <https://scholarworks.wm.edu/honorstheses>



Part of the [Computer and Systems Architecture Commons](#), [Data Storage Systems Commons](#), and the [OS and Networks Commons](#)

Recommended Citation

Adams, Jacob, "Performance Implications of Memory Affinity on Filesystem Caches in a Non-Uniform Memory Access Environment" (2021). *Undergraduate Honors Theses*. Paper 1678.
<https://scholarworks.wm.edu/honorstheses/1678>

This Honors Thesis -- Open Access is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Performance Implications of Memory Affinity on Filesystem Caches
in a Non-Uniform Memory Access Environment

Jacob Allen Adams

Fairfax, Virginia

A Thesis presented to the Faculty of
William & Mary in Candidacy for the Degree of
Bachelor of Science

Department of Computer Science

College of William & Mary
May, 2021

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Science



Jacob Allen Adams

Approved for Honors by the Committee, May 2021




Committee Chair

James Deverick, Senior Lecturer, Computer Science
William & Mary



Dan Parker, Associate Professor, Linguistics
William & Mary



Pradeep Kumar, Assistant Professor, Computer Science
William & Mary



Jiajia Li, Assistant Professor, Computer Science
William & Mary

ABSTRACT

Non-Uniform Memory Access imposes unique challenges on every component of an operating system and the applications that run on it. One such component is the filesystem which, while not directly impacted by NUMA in most cases, typically has some form of cache whose performance is constrained by the latency and bandwidth of the memory that it is stored in. One such filesystem is ZFS, which contains its own custom caching system, known as the Adaptive Replacement Cache. This work looks at the impact of NUMA on this cache via sequential read operations, shows how current solutions intended to reduce this impact do not adequately account for these caches, and develops a prototype that reduces the impact of memory affinity by relocating applications to be closer to the caches that they use. This prototype is then tested and shown, in some situations, to restore the performance that would otherwise be lost.

TABLE OF CONTENTS

Acknowledgments	iv
List of Figures	v
Chapter 1. Introduction	1
Chapter 2. Filesystems	3
2.1 Blocks	3
2.2 Volume Managers	4
2.3 Mirrors and RAID	5
2.4 State Consistency	6
Chapter 3. ZFS	7
3.1 Block Pointers	7
3.2 ZIL	9
3.3 Checkpoints and Snapshots	10
3.4 On-Disk Structure	10
3.5 VDEVs	12
3.6 SPL	13
3.7 The Many Components of ZFS	14
3.7.1 ZPL	14
3.7.2 ZAP	14
3.7.3 DMU	15
3.7.4 ARC	15
3.7.5 ZIO	15

3.8	A Read Operation in ZFS	15
3.8.1	ZPL	16
3.8.2	DMU	16
3.8.3	Prefetch	17
3.8.4	ARC	17
3.8.5	The ZIO Pipeline	17
3.8.6	Completion	19
3.9	Adaptive Replacement Cache	19
3.9.1	ARC Buffer Data	21
3.9.2	Memory and the ARC	22
Chapter 4.	What is NUMA?	23
4.1	Measuring the Impact of NUMA	24
4.2	Linux and NUMA	25
4.2.1	Memory Policy	25
4.3	Scheduling	26
4.4	Memory Reclamation	26
4.5	How Does NUMA Affect Programs?	26
Chapter 5.	Mitigating the Effects of NUMA on ZFS	27
5.1	Test Environment	27
5.1.1	Automatic Testing	27
5.2	Demonstrating the Effects	29
5.3	How to Mitigate the Effects	32
Chapter 6.	NUMA Balancing	33
Chapter 7.	Task Migration	36
7.1	Implementing Task Migration	36

Chapter 8.	Results	44
8.1	Impact of Process Size	47
8.2	Impact of Read Size	50
Chapter 9.	Conclusion	52
9.1	Future Work	53
9.1.1	Read Size Limitations	53
9.1.2	Impact on Arbitrary Programs on Long-Running Systems	53
9.1.3	Interactions with NUMA Balancing and Swapping	54
9.1.4	More Complex NUMA Topologies	54
9.1.5	Applicability to More General File Caching Systems	55
9.1.6	Global View of the ARC and of Files	55
Chapter A.	Source Code	57
A.1	Automatic Testing Code	57
A.2	Linux Changes	62
A.3	ZFS Changes	62
Chapter B.	Low-Level Memory Allocation in the SPL	68
B.0.1	kmem	68
B.0.2	SLAB Allocator	69
Bibliography		70

ACKNOWLEDGMENTS

I would like to thank my advisor Jim Deverick for his invaluable guidance on this project. Without his advice, support, and encouragement this thesis would not have been possible. I would also like to thank Linus Torvalds, Christoph Lameter, Jeff Bonwick, Matt Ahrens, and the thousands of other developers whose work has built the Linux and OpenZFS projects. Science is built on the shoulders of giants and nowhere is that more visible than in computer science.

LIST OF FIGURES

1.1	Memory Hierarchy	1
2.1	Direct and Indirect Blocks. Directly pointing to a data block makes it a level 0 block, while pointing to one indirect block before the data block makes it level 1, two indirect blocks makes it level 2, and so on.	4
2.2	Copy-on-write filesystems avoid changing any of the existing data on disk, and instead write all changes to new locations on disk[2].	6
3.1	A ZFS block pointer is a complex structure powering many of ZFS's most useful features[2].	8
3.2	Block Pointers create a Merkel Tree allowing each layer to validate the blocks below it [7]. As long as Block 0 is known to be valid, the checksums within it ensure that blocks 1 and 2 can be proven to be valid, and also every other block, as 1 and 2 contain checksums of 3, 4, 5, and 6, which themselves contain the user's data.	9
3.3	On disk a ZFS pool consists of a set of object sets[16]. The first layer is the Meta-Object Set, consisting of the filesystems, snapshots, and other components that make up the storage pool, along with essential object sets like the master object set for metadata and the space map for storing what parts of the disks have and have not been already filled with data.	11

3.4	A sample ZFS pool, with two filesystems and three snapshots [16]. When the snapshot and the filesystem are identical they point to the exact same object set, while once they diverge the snapshot continues pointing to the same object set, which might still point to many of the same objects if those files are unchanged between the snapshot and the filesystem. In this example filesystem A and its two snapshots are all different from each other, so each points to its own object set. Filesystem B, on the other hand, is the same as its one snapshot, so both point to the same object set.	12
3.5	A very simple VDEV configuration with four disks and two mirrors, each mirror with two disks.	13
3.6	ZFS consists of many components working together to write and read data from disks. This diagram depicts ZFS from the perspective of a file read operation, including the VFS layer above it that actually calls into ZFS to start a read.	14
3.7	ARC Headers, as depicted in comments in the ZFS ARC code [6, module/zfs/arc.c].	21
4.1	Uniform and Non-Uniform Memory Access	24
4.2	Local and Remote Memory Accesses	24
5.1	A flowchart of how automatic testing worked.	29
5.2	Reading a file from a different node than the one its ARC data is stored on causes a 10-15% loss of performance, proving that ZFS, specifically its ARC, is affected by NUMA.	31
6.1	Repeating the previous testing with NUMA Balancing enabled does not improve performance when accessing the ARC from a different node.	34

- 6.2 Repeatedly reading data from the ARC does not trigger NUMA Balancing, likely because it does not consider kernel memory like the ARC. If NUMA Balancing was occurring the different node latency would converge towards the same node latency. Note that the same node latency is only from the initial test run on the same node after priming the ARC to ensure that it would not interfere with NUMA Balancing by adding more reads from the same node. It is extended across the graph to provide a point of reference. 35
- 7.1 Each process has a specific set of processors it is allowed to run on and NUMA nodes that it is allowed to allocate memory from. 37
- 7.2 A visual representation of what calling `set_cpus_allowed_ptr(curthread, cpumask_of_node(1))` does to the allowed processors of the current process. 38
- 7.3 A visual representation of what calling `set_mems_allowed(nodemask_of_node(1))` does to the allowed memory of the current process. 40
- 8.1 Comparing ZFS with a modified version using task migration, a 10-15% improvement is visible with task migration for files 1 GB and larger when the ARC data being accessed is on another NUMA node. 44
- 8.2 Task migration with the ARC on a different node compared to starting the process on the same node. The results are practically identical, showing that task migration can significantly improve performance without substantial overhead in this scenario. 45
- 8.3 Bandwidth improves almost as much as latency when accessing ARC data from a remote node with task migration. Note that unlike most other graphs in this work this uses 2 MB reads, but the results are consistent with 1 MB reads tested with previous versions of my prototype. 46
- 8.4 Bandwidth, as with latency, is identical when on a different node with my task migration prototype as compared on reads from the same NUMA node. 47

8.5	The Effect of Process Size on the Percentage Improvement in Runtime. Measuring using the simple read program sized to various process sizes, with a read size of 1M.	49
8.6	Impact of Read Size	50

Chapter 1

Introduction

Data access is a critical component of modern computing and often the bottleneck of performance for many applications, especially as processors have continued to get faster and faster over time. Because data storage that is the quickest to access is also the most expensive, all the data in a system cannot be available at one time equally quickly. Instead a hierarchy of data access was created, where the higher levels can store less data but are quicker to access and the lower levels can store more data but are slower to access. This fundamental principle of computing was a major consideration in the design of computers as long ago as 1946[9].

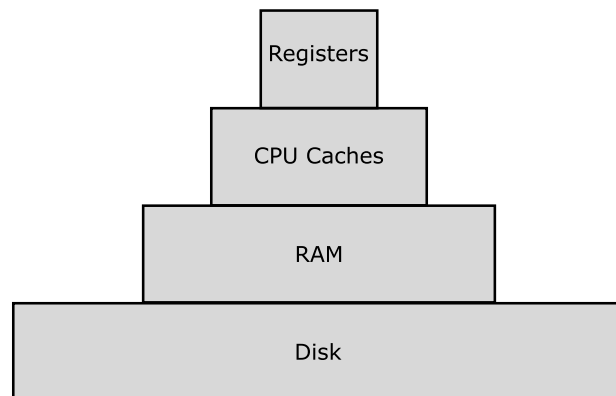


Figure 1.1: Memory Hierarchy

At the top of this hierarchy are registers directly stored on the processor, and at the bottom are disks, slow but vast in size. The organization of this lowest level of the memory hierarchy, and efficient use of the higher levels that have become more complex over time, is the subject of

this thesis. Specifically the filesystem, which organizes the data stored on disks, has become more powerful over time, allowing data to be spread out among multiple disks and ensuring it remains exactly as it was stored, despite the limitations of every layer of the system. One of these modern filesystems is ZFS, which is the focus of this work. ZFS handles its own interactions with the higher levels of the memory hierarchy, specifically RAM, unlike most other filesystems on Linux, an operating system kernel in common use for the most powerful computers today.

The RAM layer of the memory hierarchy has become more complex over time, as processors have become more and more powerful. Thus ZFS's methods for improving its own performance with RAM-based caching are no longer optimal on some systems. This work investigates methods for optimizing them on these new systems and one particularly interesting new method showed promising, though limited, results.

Chapter 2

Filesystems

Filesystems are a critical part of modern computing, providing long term storage of data. Many different filesystems exist today, including FAT¹, NTFS, Ext4, HFS+², BTRFS, and ZFS³. Some filesystems are much simpler, working with only one disk and providing only a journaling system to recover from system crashes or other inconsistencies. Others, like BTRFS and ZFS, are much more complex, allowing users to store vast amounts of data over multiple disks.

Journaling is a method used by filesystems to recover from inconsistencies found on disk. They record what operations they intend to perform on the disk in a separate area before actually making any changes. Thus if they are interrupted when writing out data, the journal can be used to complete that operation when the system boots up again. Working with multiple disks can be done with any filesystem through the use of a volume manager, but there are distinct advantages to having native support for multiple disks, as discussed in Section 2.2.

2.1 Blocks

Blocks are the fundamental unit of a filesystem. They are typically fixed-sized units on a disk referred to by their offset from the start of the disk. There are two types of blocks, which filesystems

¹A filesystem that is still ubiquitous today, despite lacking any of the features that are discussed in this paper.

²NTFS, Ext4, and HFS+ are filesystems known for having journaling capabilities, and are the default filesystems for Windows, Linux and MacOS respectively. NTFS also allowed for a major increase in the maximum size of files, which was previously limited to 4 GB on FAT.

³BTRFS and ZFS are newer modern filesystems that both support generally the same set of features, though ZFS is well known for being more stable and is the focus of this work

combine to allow them to store variably-sized, non-contiguous structures, which are most commonly files. These two types are direct blocks, those that simply store the user's data, and indirect blocks, which store pointers to other blocks, either indirect or direct. The root block of a file is then always a form of indirect block, which can then point to either some direct blocks if the file is very small, or more likely a set of indirect blocks that themselves point to more indirect blocks and eventually direct blocks, with the number of intermediate layers depending on the size of the file.

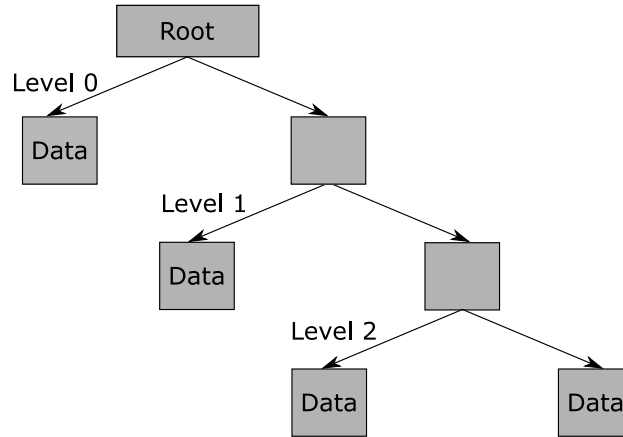


Figure 2.1: Direct and Indirect Blocks. Directly pointing to a data block makes it a level 0 block, while pointing to one indirect block before the data block makes it level 1, two indirect blocks makes it level 2, and so on.

Using blocks allows filesystems to have variably-sized non-contiguous data structures, which is critical when storing structures like files that can be widely varying in size.

2.2 Volume Managers

Typical filesystems require a volume manager when working with multiple disks. This is because the typical filesystem can only work with one disk at a time. Volume managers solve this problem by acting as an intermediate layer between the filesystem and a set of disks, making them appear as one logical disk to the filesystem. They can be organized in a variety of ways to provide redundancy, as discussed in the next chapter. They can also subdivide one disk, or even one disk partition into smaller units that can be extended as needed.

Modern filesystems, like ZFS and BTRFS, no longer require this intermediate layer, instead building in support for multiple disks and disk sharing between filesystems directly. This allows for more fine-grained control of where data will be stored, at the expense of each filesystem having to design its own method for spreading its data across multiple disks in various configurations, such as mirroring and RAID.

2.3 Mirrors and RAID

When a filesystem is made up of more than one disk, data can be arranged in different strategies to prevent any one disk failure from losing all of your data. For simpler filesystems without multiple disk support this is all handled by the volume layer, but some more advanced filesystems handle this themselves. The most conservative of these strategies is mirroring, or RAID-1, where all data is duplicated exactly on multiple disks, with the filesystem only having access to one disk's worth of storage[18]. This solution dramatically reduces the amount of storage available but ensures all data is easily recoverable in the event of a disk failure. Beyond this are the levels of RAID, that trade off various amounts of disk storage for redundancy, adding parity bits and other tricks in order to ensure data recovery if only some number of disks are lost. Some RAID configurations have parity information all on one disk or a set of disks, allowing any one disk, or some number of disks, but not all the parity disks, to fail while still recovering all data. Others spread the parity information out among all the disks, reducing the redundancy but allowing for many writes at once, not being constrained by having to always write to the parity disk.

When this is handled by the filesystem, it can use what it knows about the redundancy of the data to transparently repair the filesystem, rewriting data if it detects that one of the redundant copies has become corrupted or was incorrectly written to disk. This requires some kind of corruption detection mechanism, but such a mechanism is typically a standard part of a RAID configuration.

2.4 State Consistency

One goal of every filesystem is to never be in an unrecoverable state on disk [2, 16]. Most filesystems handle this situation through a journal system and the filesystem check command, known as `fsck`, that can restore the filesystem to a working state, even if a write is interrupted. Other filesystems, such as ZFS and BTRFS, try to categorically eliminate this source of failure through a Copy on Write model. Copy on Write means that any changes to a file will result in a write to an entirely different part of the disk, instead of overwriting the previous contents of the file. This means that even if a write fails, the filesystem will simply appear to be in exactly the same state as before, because only overwriting the root block, or *superblock*, of the filesystem actually changes its state in a destructive way, linking up all the new, and usually some old, blocks into the filesystem all at once.

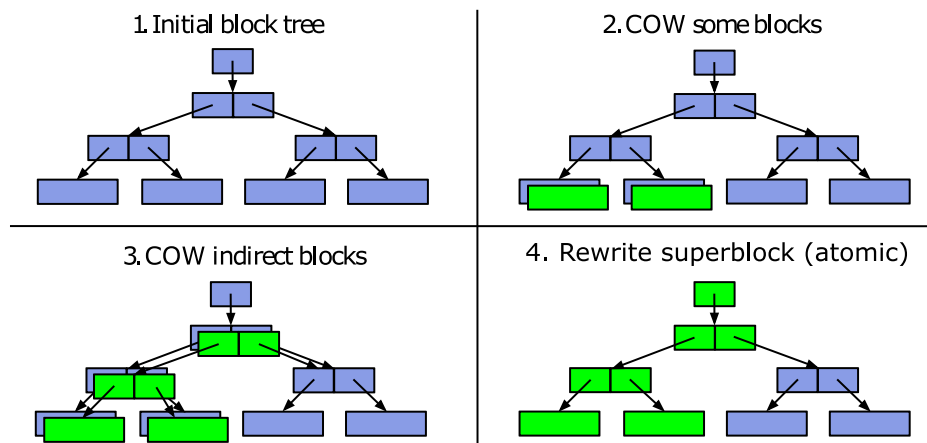


Figure 2.2: Copy-on-write filesystems avoid changing any of the existing data on disk, and instead write all changes to new locations on disk[2].

Chapter 3

ZFS

ZFS, formerly known as the Zettabyte Filesystem, is a filesystem developed originally for the Solaris operating system by Jeff Bonwick and Matt Ahrens [1]. Its most innovative feature is combining the roles of volume manager and filesystem, which allows ZFS to pool storage in a way that works much like allocating memory in a program[16]. A storage pool can consist of multiple disks in various configurations, and all of the available storage of those disks can be used by any of the filesystems that are present in the pool. ZFS is also Copy on Write, as discussed above, which both ensures that the filesystem can never be in an inconsistent state on disk and allows for snapshots, read-only copies of a filesystem from a given point in time, as old versions can be kept around with ease by simply not deleting old blocks. Snapshots also only take up the extra space of their difference between the current state of the filesystem, as the blocks used by both are not duplicated, but instead simply pointed to by both the snapshot and live filesystem.

3.1 Block Pointers

Block pointers are the fundamental structure on which ZFS's most powerful features are built. They are 128-byte structures that uniquely identify a block in a storage pool [1, 2, 16]. They can contain up to three different addresses for three different copies on different virtual devices, or VDEVs, that represent logical, not physical, disks. VDEVs can be just one physical disk, or they can be a mirror, where multiple disks contain exactly the same data, or a RAIDZ pool, where data are striped across multiple disks and a parity block is appended so that data can be recovered in

the event of a disk failure. These addresses are Data Virtual Addresses, and they contain a VDEV number, a size, as blocks can be variably-sized in ZFS, with a maximum size of 128 KB, and an offset, the distance in bytes from the start of the logical disk to the start of the block. By default, metadata, that is data about a particular filesystem, snapshot or other object in the storage pool, are stored twice, and pool-wide metadata, that is data about all of the filesystems and other objects in the storage pool, are stored three times. However, there can only be at most two copies if the pool is encrypted, as encryption key information is stored in the third DVA in the block pointer. Pool-wide metadata are never encrypted and thus are not affected by this limitation, even though they are stored three times in the pool.

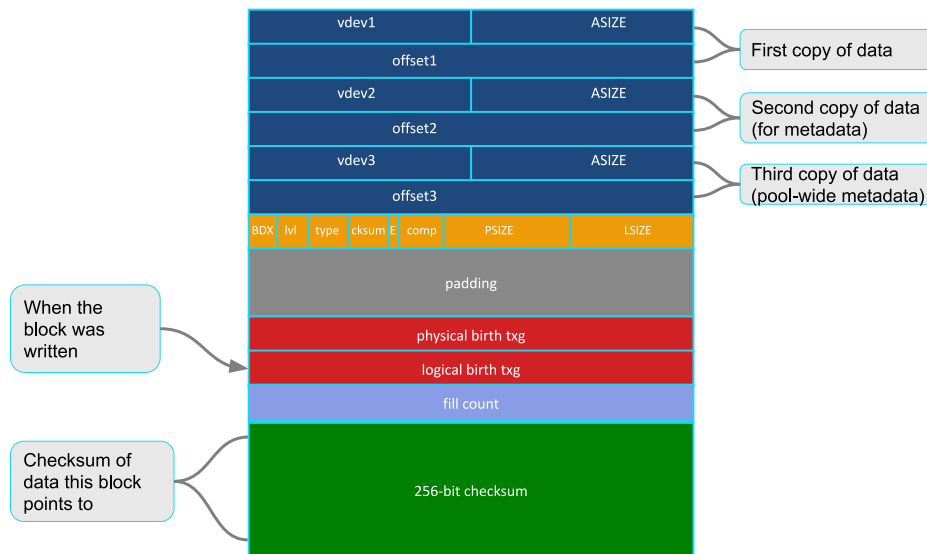


Figure 3.1: A ZFS block pointer is a complex structure powering many of ZFS’s most useful features[2].

The most important feature of block pointers is that they contain a 256-bit checksum of the data they point to. These checksums protect against failures of the initial write and allow detecting any corruption that might occur later when reading the data back from the disk. They also allow for repairing data on one side of a mirror, if the other side is uncorrupted, as its correct data can then be written from the known-good side of the mirror and the corrupted data on the first side can be deleted. Because block pointers are addresses, but also contain a checksum, they form a hash tree, also known as a Merkle tree, of addresses and checksums (Figure 3.2). Each checksum is

also validating the checksum of the blocks below it in the tree as each block contains the checksum of its child blocks. Thus each layer of the tree can validate all of the data below it.

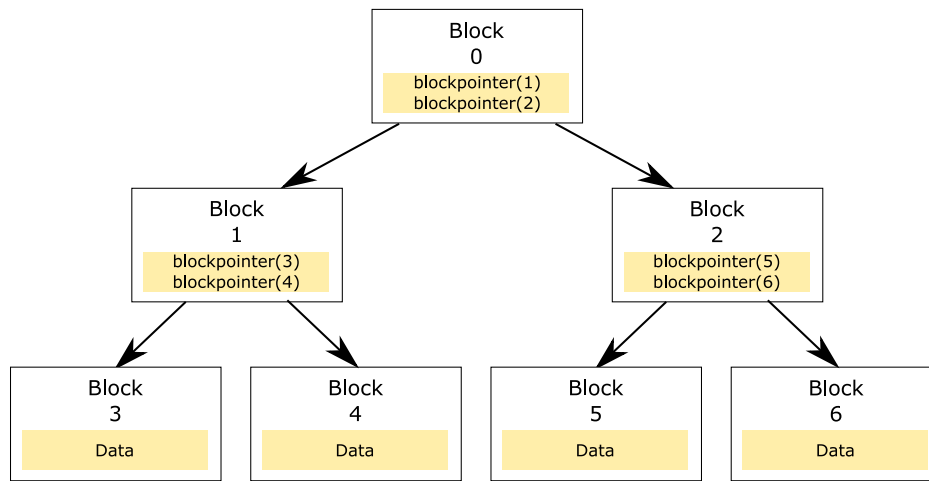


Figure 3.2: Block Pointers create a Merkle Tree allowing each layer to validate the blocks below it [7]. As long as Block 0 is known to be valid, the checksums within it ensure that blocks 1 and 2 can be proven to be valid, and also every other block, as 1 and 2 contain checksums of 3, 4, 5, and 6, which themselves contain the user’s data.

3.2 ZIL

The ZFS Intent Log is the ZFS’s journal [1]. It functions like a journal in other filesystems, ensuring synchronous writes without flushing everything to disk and requiring rewriting the entire tree of blocks every time `fsync` is called. These `fsync` operations are intended to force all data to be written to disk, but instead on ZFS only the ZIL is synced to disk. This allows these operations to be fast without actually having to write all the new direct and indirect blocks to the disks, but ensures that all data can be recovered by ZFS in the event of sudden power loss or crash before the next checkpoint is written to disk.

The ZIL is a linked list of changes made to the filesystem. It works around the problem of pointers in a copy-on-write filesystem, which would typically require changing the previous block to have a pointer to the next block, by adding a new empty block to the end of list any time it adds more data. New data are then written to the first completely invalid block found in the ZIL, under the assumption that it is the final empty block. This allows ZFS to not have to change the

pointer address of previous blocks and keep its copy-on-write guarantees, never rewriting the data of already written blocks in place.

When ZFS restores from power loss, it gets the last good state of the current uberblock, the root block of the storage pool, and then updates write by write to the last write that was synced to the ZIL [16].

3.3 Checkpoints and Snapshots

After a certain amount of writes to the pool accumulate, either 4 GB or 25% of memory, whichever is lower [6, `module/zfs/arc.c`], ZFS writes them out to the disks of the pool all at once as a checkpoint, or transaction group [1, 16]. It must first collect all updates that have happened since the last checkpoint from the ZIL and then write them out to unused locations in the pool before finally writing a new uberblock for the entire pool. Uberblocks are used in a rotation as checkpoints happen, with a set of either 128 or 256, depending on the physical block size of the disks in the pool.

Because ZFS works on a system of checkpoints and copy-on-write storage, snapshots are trivial for the filesystem to keep around. It must simply save the root block of the filesystem at a given point in time. Deleting them, however, is much trickier, because it is difficult to find the blocks that only exist in that snapshot when it is deleted. This is critical to do, as a block can only be freed if everything referencing it no longer needs it. Birth time is used to figure out if any snapshots need a particular block when ZFS is deleting a snapshot. The original copy-on-write filesystems used garbage collection, but this was very slow. ZFS instead uses a dead list of what blocks the snapshot no longer cares about relative to the snapshot right before it.

3.4 On-Disk Structure

ZFS's on-disk structure starts with an uberblock, the root block for the entire pool [1, 16]. The uberblock points to the Meta-Object-Set, or MOS, which is a set that keeps track of everything in the pool such as filesystems or snapshots, as well as clones, copies of a filesystem that are themselves

filesystems, and ZVOLs, large files stored in ZFS outside of a filesystem that are intended to be used as disks for virtual machines. ZFS treats all of these on-disk objects as sets of sets, and they are implemented through direct and indirect blocks, which allows these sets to be arbitrarily large (Section 2.1). The first object in the Meta-Object-Set is a master object set that contains information about the pool itself. The last is the space map, which keeps track of which blocks are free and which are being used over the entirety of every disk in the ZFS storage pool. Each object in the MOS is itself an object set that describes the objects within it. For example, filesystem objects are a set of files, directories, and symlinks, files that simply reference another file and are treated by most programs as exactly the same as the files they link to. Objects are also implemented in the same way as object sets, pointing to indirect blocks which eventually point to direct blocks which contain the data that makes up the object.

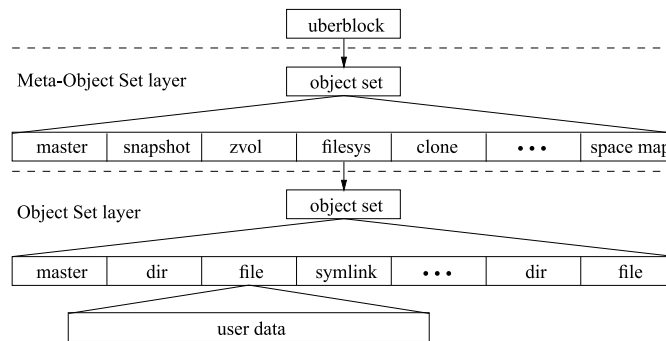


Figure 3.3: On disk a ZFS pool consists of a set of object sets[16]. The first layer is the Meta-Object Set, consisting of the filesystems, snapshots, and other components that make up the storage pool, along with essential object sets like the master object set for metadata and the space map for storing what parts of the disks have and have not been already filled with data.

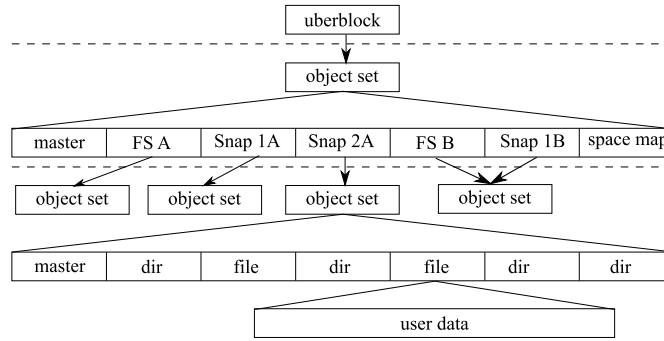


Figure 3.4: A sample ZFS pool, with two filesystems and three snapshots [16]. When the snapshot and the filesystem are identical they point to the exact same object set, while once they diverge the snapshot continues pointing to the same object set, which might still point to many of the same objects if those files are unchanged between the snapshot and the filesystem. In this example filesystem A and its two snapshots are all different from each other, so each points to its own object set. Filesystem B, on the other hand, is the same as its one snapshot, so both point to the same object set.

3.5 VDEVs

ZFS implements mirrors and RAID-like configurations, known as RAIDZ, through Virtual Devices, or VDEVs. Storage in ZFS is implemented as a set of disks organized into a tree, with one root whose leaves are either physical devices, mirrors, or RAIDZ devices of various levels of redundancy. These mirror or RAIDZ devices have child devices which are the actual physical disks. Physical disks, on the other hand, are always the leaf nodes of the tree as they are what actually stores the data.

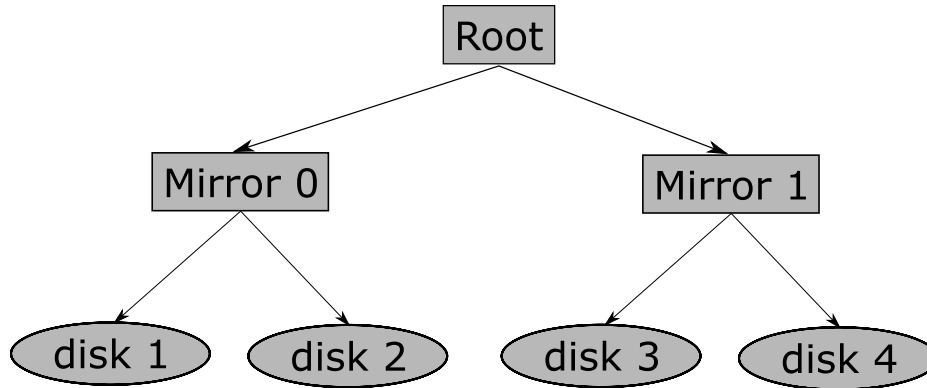


Figure 3.5: A very simple VDEV configuration with four disks and two mirrors, each mirror with two disks.

3.6 SPL

The SPL, or Solaris Portability Layer, is a major component of ZFS that allows it to work in Linux, by wrapping Linux kernel interfaces to be more Solaris-like [6]. This even includes redefining Linux kernel functions with entirely different interfaces in its headers. Most important low-level operations performed by ZFS are handled through the SPL, such as allocating memory or creating new threads. There are two major exceptions, which are disk operations and allocating ARC Data Buffers (Section 3.9.1).

It was written by Brian Behlendorf at the Lawrence Livermore National Laboratory as part of the Laboratory's port of ZFS to Linux, which eventually became OpenZFS. It is a GPL-licensed kernel module, under the same license as the Linux kernel. This is in order to use Linux functions that are only exported to modules licensed under the GPL. These functions cannot be used from ZFS directly as it is licensed under the CDDL.

3.7 The Many Components of ZFS

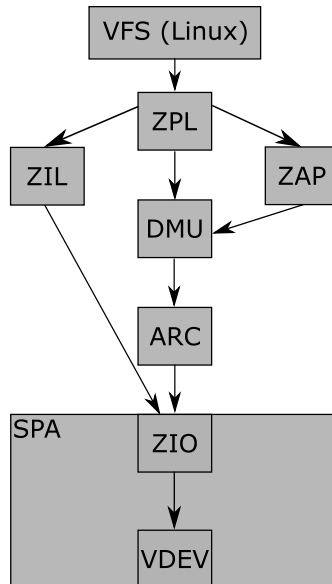


Figure 3.6: ZFS consists of many components working together to write and read data from disks. This diagram depicts ZFS from the perspective of a file read operation, including the VFS layer above it that actually calls into ZFS to start a read.

3.7.1 ZPL

The ZPL is the ZFS Posix Layer [10]. This component interfaces with the VFS, or Virtual Filesystem, layer above it, which handles filesystem operations to local or network filesystems. within the Linux kernel. All file operations within ZFS are actually organized as operations on objects, of which files are just one special type. Thus the ZPL must translate requests for file reads and writes into operations on specific objects in specific object sets. It is the initial entry point for reads and writes to files in ZFS.

3.7.2 ZAP

The ZAP is the ZFS Attribute Processor [10]. It is a key-value store that is used primarily for directories, and it stores a mapping of names to objects in the object set. In other words, it primarily maps the names of files in a directory to the objects that hold the contents of those file.

3.7.3 DMU

The DMU is the Data Management Unit [1]. Its primary job is to bundle operations into transactions, and hold the first layer of caching for the filesystem as a set of DBUF objects. These DBUF objects are just metadata that points to data stored in the ARC, to avoid even having to ask the ARC for frequently accessed data.

3.7.4 ARC

The Adaptive Replacement Cache is ZFS's large in-memory cache [1, 2]. This cache consists of two internal dynamically-sized caches with different policies, a least recently used cache for data accessed only once and a least frequently used cache for data accessed multiple times. The ARC is the most important part of ZFS for the purposes of this work and is covered in more detail in Section 3.9.

3.7.5 ZIO

ZIO is the ZFS I/O Pipeline and it is the primary component of the SPA, or Storage Pool Allocator [1]. All operations performed on disks go through it. It also handles compression, checksumming (and checksum verification), and encryption.

There are many different ZIO queues, contained within a two-dimensional array in the per-pool spa object [6, module/zfs/spa.c]. There are six types of ZIO: read, write, free, claim, ioctl, trim, each with 4 queues of different priorities, issue, issue high, interrupt, and interrupt high. The ZIO scheduler considers five classes of I/O, prioritized in this order: synchronous read, synchronous write, asynchronous read, asynchronous write, and scrub or resilver.

3.8 A Read Operation in ZFS

A read operation to a filesystem in ZFS goes through almost every part of the filesystem and gives a good overview of how ZFS works. It is also the operation that this work focuses on improving. Describing the process as it currently exists is critical to showing how it may be improved. Each

snapshot, filesystem, or clone has its own ZPL and DMU, but the ARC and SPA are shared per storage pool [1].

3.8.1 ZPL

The ZFS POSIX Layer is entrypoint from the Linux VFS, the Virtual FileSystem layer into ZFS [1, 6]. The VFS is the common interface between Linux and all of its filesystems. It asks the ZPL for some amount of data from a specific file at a specific offset and gives it a buffer to fill. The ZPL then converts this information into a ZFS object via the ZAP and requests that from the DMU.

3.8.2 DMU

The Data Management Unit takes the object request from the ZPL and first looks to see if that object is cached in its internal cache [1, 6]. This cache consists of DBUFs, or DMU buffers, which represent one logical block, one block of one file of one filesystem. Their actual content is just a reference to an ARC buffer (Section 3.9). If the DMU has all the DBUFs that are needed to satisfy this particular read, then it sends the data back to the ZPL. This is the quickest path through the read code available within ZFS, and is the best of best cases.

Otherwise, the DMU will make a new DBUF and locate the block pointers for the data requested. To do this it has to look for the indirect block parents that point to the blocks or the parents of those blocks, or the parents of the parents of those blocks, continuing up the tree of blocks until it locates something in the DMU cache. The dnode of the file, which contains the root indirect block pointer of the file, will always be cached because the file reference passed by the VFS is a vnode that points to it. Once an indirect block is located, the chain of block pointers must then be fetched to eventually find the block pointer for the actual data. These block pointers are then passed to the ARC to retrieve the data they point to. The ARC will eventually give back a pointer to an ABD, which is then associated with the new DBUF. The prefetch engine is informed about all reads as they happen, in order to detect patterns and preemptively request data that it believes will be needed soon (Section 3.8.3).

The DMU also creates an empty ZIO root and passes it along to the ARC. This allows it to wait

on this root which will be made the root of any disk operations that the ARC needs to perform in order to retrieve the data. By waiting on the root it will wait on all of its children, which requires waiting on their children, which ensures that all required I/O has finished. This guarantees that once the wait is over all the data the DMU requested has been populated into the correct buffers and it can simply return that data to the user.

3.8.3 Prefetch

The prefetch engine looks for patterns in the requests that the DMU receives and preemptively requests the data that it thinks will be needed[1]. It tracks many streams of reads at once to find those that are sequential in order to fetch the next blocks in those sequences. When it detects a read that is sequential to a previous read, it will request that the next few blocks in the sequence be loaded into the ARC. How many blocks it asks for depends on how successful prefetching has been previously, fetching more and more data up to 8 MB at a time if prefetching continues to correctly guess the data that will be accessed next[6, module/zfs/dmu.c module/zfs/dbuf.c]. This ensures that the data the process will be asking for is already in the ARC before it even needs it, which can speed up file accesses.

3.8.4 ARC

The Adaptive Replacement Cache is primarily a hash map of blocks indexed by block pointer, so the first thing it does is check this hash map for the passed block pointer [1, 6]. If it is already in the ARC, then it simply returns that cached data to the DMU. Otherwise, a new entry in the ARC, called an ARC header, is created. The ARC then invokes the ZIO pipeline to retrieve the data from the disks.

3.8.5 The ZIO Pipeline

ZIOs form a dependency graph, which lets lots of I/O be pending at once [1]. Due to the format of ZFS on disk, one small read might depend on reading a large number of blocks, depending on

the size of the file. The indirect blocks that allow files to grow as large as needed also mean that more reads are required in order to retrieve the data stored in the direct block from a disk.

This system is especially beneficial to writes, as each indirect block can be added to the tree with a dependency on all the data blocks below it, and once those are complete that indirect block can be written, regardless of how many other data blocks might still need to be written.

The goal of this complex dependency system is to keep all disks busy at all times whenever ZFS has to read or write data. As disks are typically the bottleneck, being the slowest component of a typical modern system, it is important to maximize their throughput for the best performance. It also allows for easy representation of more complex data layouts, like mirroring or RAIDZ, where the logical operation of writing a single block depends on multiple physical writes to disks.

ZIOs also verify the checksums of the blocks they read, which allows them to try and recover when those checksums show that data to be invalid. It can use the block pointer to find the locations of other copies of the data, if others were created, and create new children ZIOs to get that data instead. Thanks to the checksum in the block pointer, it can then create another child ZIO to transparently write a new block containing the correct data to the corrupted side of the mirror, once it has found a valid copy.

If the data being read is compressed or encrypted on disk, then a secondary buffer will be allocated for the on-disk data, and a decompression or decryption step will be added to the process via pushing a the required transformation to the ZIO's transformation stack. Compression is almost always used in real deployments of ZFS, as it improves performance by allowing for fewer disk operations.

This process begins with a logical ZIO associated with a block pointer, but these are transformed into child physical ZIOs for a specific VDEV and offset, based on the DVAs present in the block pointer, typically just picking the first DVA found in the block pointer. These physical ZIOs pass their data on to the logical ZIO, which figures out if the read succeeded or if it needs to try a different DVA.

A DVA might point to a non-leaf VDEV, that is a mirror or RAIDZ VDEV instead of a physical disk, in which case more physical child ZIOs will be created to retrieve the data for the original

physical ZIO.

Eventually, there will be a ZIO associated with a leaf VDEV, at which point it is, finally, queued for being read from disk. Since disks can perform far fewer operations at once, ZIOs are prioritized and sorted by disk offset to avoid having to unnecessarily seek the disk head when possible. The queue also aggregates small reads into larger reads when it finds that smaller reads currently queued are physically sequential on disk.

The pipeline is given the next ZIO to execute and runs the relevant operation on disk, which eventually calls Linux's `submit_bio` function. In the case of a synchronous operation, it will call `zio_wait` and execute ZIOs from the queue, in parallel with `zio` worker threads, until the passed ZIO, the root ZIO of all the required operations, is complete.

3.8.6 Completion

The ZIO pipeline will, after completing many, many ZIOs, populate an ARC buffer and call the ARC read done callback passed by the DMU, which associates the passed ARC buffer with the relevant DBUF [1, 6]. Once all the necessary DBUFs are in the DMU, then the data is copied into the buffer passed by the ZPL, which ultimately came from VFS layer and completes the read operation.

3.9 Adaptive Replacement Cache

ZFS relies heavily on memory to ensure performance while retaining its always-consistent property [1]. The primary way in which this is done is the ARC, or the Adaptive Replacement Cache. This cache works like a page cache in other filesystems, storing the contents of files that have been accessed in order to ensure quicker access to those files in the future and storing the contents of writes that have yet to be propagated to disk.

The ARC consists of two dynamically-sized caches, one that follows a least recently used eviction policy and another which follows a least frequently used eviction policy. The first contains data that has been accessed only once and the second contains data accessed twice or more [17]. A least recently used cache policy will discard that data which has been accessed least recently when filled

and needing to add more data, as the name would suggest, A least frequency used cache policy will discard instead that data which has been accessed least frequently, that is the fewest number of times. These caches are hash tables of block pointers, so the ARC has no understanding of files, delegating that to the abstraction layers above it.

The relative size of these two caches is determined by a ghost cache. The ghost cache contains the headers, but not the data, of all the recently evicted blocks along with which part of the cache they were stored in. When a cache miss occurs in the ARC, it checks the ghost cache to see if the data requested was recently stored. If it was, then the ARC will make the component of the cache that data was previously stored in larger and the other smaller, under the assumption that this would have helped the ARC perform better and have fewer cache misses.

This combination of multiple cache types gives the ARC the important advantage of being scan-resistant, that is reading a very large file one time sequentially will not substantially degrade the performance of the cache. In simpler caching systems, such as a single LRU, reading data that is larger than the size of the cache will replace the entire contents of the cache. This is a problem, as one large file read can then destroy the performance of the cache on most systems, where often certain files are accessed quite frequently and should always been cached if memory allows.

The ARC has also been extended in ZFS with fast disk storage, known as the L2ARC, while the in-memory ARC is now referred to within ZFS as the L1ARC. When configured, the ARC will use a faster disk, such as an SSD, as a secondary larger cache for speeding up access to your data. L2ARC is of course still slower than L1ARC, but it can be much larger than the L1ARC, because of the vast differences in size that generally exist between disks and RAM.

Internally the ARC is implemented as a hash table of headers, metadata structures that point to the actual contents of the blocks stored in the ARC. These header structures point to a set of buffer structures, which themselves point to ABD, ARC Buffer Data, structures containing the actual data. The first and most important of these buffers is the physical buffer, the ABD that contains the contents of the physical block on disk. Every header that is in the L1ARC will have at least this buffer, though it is freed when the header moves to the ghost cache.

Every time a block is requested from the ARC by a new consumer, a new entry is added to the

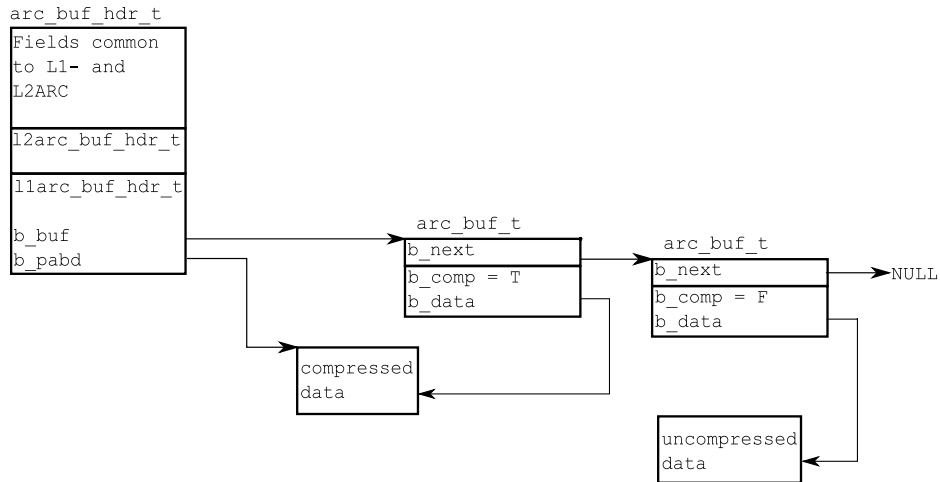


Figure 3.7: ARC Headers, as depicted in comments in the ZFS ARC code [6, module/zfs/arc.c].

consumer list, called the buf list within the ARC, with its own data pointer to an ABD containing the requested data. A consumer is a higher level component of ZFS, such as a particular filesystem’s DMU, that needs to retrieve the data pointed to by a block pointer. The buffer attached to this consumer buf structure is what the consumer receives back from the ARC when its data is available.

What the ABD that the buf structure points to looks like depends on what the block’s data actually looks like on disk. If the block requested is compressed or encrypted, than the ARC will need to allocate a new buffer to store the uncompressed data, which will then to pointed to by the new buf structure before it is returned to the consumer. If the data on disk are not compressed or encrypted, then the buf structure will just point to the physical ABD of the header, as no changes are needed in order for it to be used by the consumer.

3.9.1 ARC Buffer Data

ARC data is stored in an `abd_t`, an ARC Buffer Data structure, which allows the creation of both large contiguous, or linear, buffers and buffers that might consist internally of a series of equally-sized buffers allocated in different locations, known as a scattered buffer [6, module/zfs/abd.c]. This allows ZFS to avoid having to find large contiguous chunks of memory for ARC buffers when the system is running low on memory and avoids having to map the entire buffer into the kernel’s memory space when only part of it is accessed. Both types can be accessed using the same set of

functions, allowing the use of scattered buffers instead of linear buffers wherever it appears useful, without having to change any consumers of those buffers.

ABDs are allocated in terms of pages, attempting to allocate first from the local memory node, but the allocation process prioritizes keeping as much of the buffer as possible on the same node, which might not be the local node if its memory is very full. Unlike every other part of ZFS, it uses Linux's `alloc_pages_node` to directly allocate a set of pages from a specific area of memory (a specific NUMA node, explained in Chapter 4). It does this to attempt to ensure the series of allocations that it makes are as close together as possible, regardless of whether they are close to the current process. This behavior is clever, but undesirable for the purposes of this project, as it means that larger allocations can be spread out over multiple areas, depending on the available memory.

3.9.2 Memory and the ARC

The ARC requires large amounts of memory, but gives ZFS comparable or even faster performance than other filesystems in some circumstances. However, this also means that ZFS is significantly affected by memory latency, more so than other filesystems, because they don't typically read your files directly from memory.

One development that has recently become quite important in improving high-performance systems, is in the inclusion of a memory controller within each physical CPU[13]. Now systems with multiple CPU sockets have multiple memory controllers, meaning that the access latency to different parts of memory is no longer uniform, thus these new architectures came to be referred to as Non-Uniform Memory Access, or NUMA, architectures.

Chapter 4

What is NUMA?

The traditional models of computing, the Turing Machine and the von Neumann architecture, assume that all the memory a computer has is accessible in the same amount of time. Programmers have followed this assumption, and have generally not cared about where exactly in memory their program's data are stored, or on what processors their programs run.

However, as newer systems have gotten faster and faster, with more and more cores and memory, this fundamental assumption could no longer hold [13]. As processors have become faster and more numerous in systems, the absolute distance between a processor and memory began to matter for latency. In order for memory to keep up, new system architectures had to be devised to allow memory to be closer and thus keep pace with faster processors. This was done, as outlined above, by integrating the memory controller directly into the CPU package, allowing memory to be directly attached to the processor (Figure 4.1). While this allows for much lower latency when accessing that memory from a processor in that socket, it makes access to memory attached to any other socket very slow, as it first has to pass through a memory interconnect between sockets (Figure 4.2). This property of these architectures is known as NUMA, or *Non-Uniform Memory Access*. A processor and its attached memory are together known as a node.

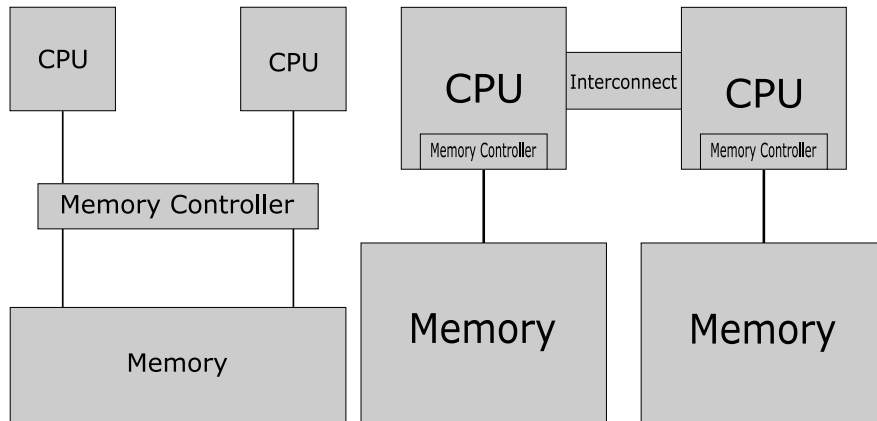


Figure 4.1: Uniform and Non-Uniform Memory Access

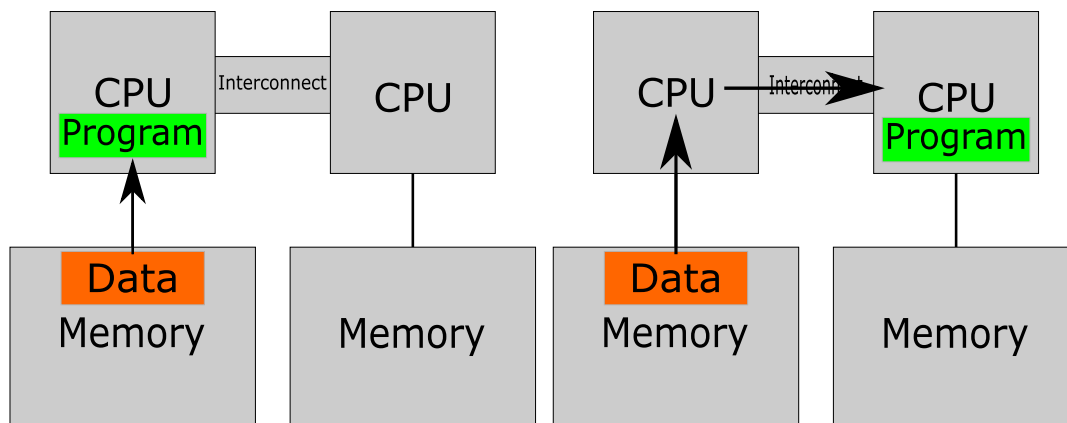


Figure 4.2: Local and Remote Memory Accesses

4.1 Measuring the Impact of NUMA

Tools such as the STREAM benchmark can be used to measure the bandwidth of memory based on the throughput of various operations performed on memory [8]. While originally used to measure CPU performance, “On modern hardware, each of these tests achieve similar bandwidth, as memory is the primary constraint, not floating-point execution” [8]. STREAM showed a 33% reduction in bandwidth when accessing memory remotely, from a program running on one node to memory located on another node. This went up to 39% when running with multiple threads, which of course caused contention over the interconnect, as it is shared between the multiple processors on the same CPU socket.

Another tool used to measure the bandwidth and latency of machines with a NUMA architecture is the Intel Memory Latency Checker. The Intel MLC is likely to have a more reliable result as it is directly intended to measure NUMA latency and bandwidth, while STREAM only incidentally does so. Using this tool in my test environment, I found that there was about a 63% increase in latency (66.8 nanoseconds locally versus 108.8 remotely) and about a 50% reduction in bandwidth (18405.1 MB/s locally versus 9284.0 remotely). Thus, there is a significant performance impact from NUMA, to both memory bandwidth and memory latency.

4.2 Linux and NUMA

The most important place for NUMA support is of course the operating system, and it has been supported in Linux since version 2.5, with the current system of APIs and scheduler support added in version 2.6 [11] [4, Documentation/admin-guide/mm/numa_memory_policy.rst]. There are two main ways in which the operating system has to support NUMA. First, in memory allocation, so that a process can ask for memory from a specific node, and in the scheduler, so that a process can be kept on the socket closest to the memory that it uses.

4.2.1 Memory Policy

Linux uses memory policy to determine where an allocation should take place [4, Documentation/admin-guide/mm/numa_memory_policy.rst]. Typically, a process has a local memory policy, such that it prefers memory from the NUMA node closest to where it is currently running when possible. Other configurations are possible for better performance in some scenarios, including preferring a specific node, or binding all memory allocated to a specific node, such that the process would rather an allocation fail if it was unable to allocate memory from a specific node. Memory in Linux is allocated on a first touch policy, that is the process or thread that first accesses the memory determines the policy used to decide where that particular allocation will be located. As an example, if two threads had a local memory policy and one thread running on node 0 were to allocate some memory, but another thread running on a node 1 were to be the first to actually write to that memory, that allocation would be placed closest to the second thread on node 1.

4.3 Scheduling

Linux's scheduler is also aware of NUMA [4, Documentation/scheduler/sched-domains.txt]. This allows it to attempt to keep processes running on the same NUMA node. It uses a hierarchy of scheduling domains, which the scheduler tries to keep a process within while still balancing all running processes across all available processors. These include not only the same NUMA node, but also the same physical CPU core or even the same virtual processor when possible.

4.4 Memory Reclamation

Linux generally will allocate all available memory for caching, as files are read from disks [13]. Only when all system memory is running low does it release these memory caches, either those not in use or those least likely to be used, freeing them for use by the rest of the system. When memory runs low on a particular node, Linux usually just allocates on another node, unless the system has 4 or more nodes, at which point it frees memory from the local node when that node is low on memory. Unlike every other filesystem on Linux, ZFS does not utilize Linux's page cache, with the exception of when files are directly mapped into memory by a process. Thus it does not take advantage of this caching mechanism, instead using its own internal cache, the ARC (Section 3.9). This allows ZFS to have much more control of how it caches files and use a very clever caching policy, an Adaptive Replacement Cache.

4.5 How Does NUMA Affect Programs?

NUMA architectures mean that operating systems and programs need to be cognizant of where they allocate memory from in order to achieve maximum performance. Poor memory and CPU location can lead to substantially increased latency in most cases [13].

Chapter 5

Mitigating the Effects of NUMA on ZFS

5.1 Test Environment

All experiments were performed on a Dell PowerEdge 610 with two Xeon X5570 CPUs and 48 GB of RAM split evenly between them. This system has two NUMA Nodes, each with 4 processors, with each processor capable of hyperthreading and thus running two processes at once. The NUMA nodes are connected directly via Intel's Quick Path Interconnect, their first NUMA interconnect system [12]. Each node has 24 GB of DDR3-1066 RAM. The system has three disks, two root disks, each 67.77 GiB in size, mirrored by the BIOS (one of which failed during testing), that were used to store the operating system, the testing scripts, and the test results, and one ZFS disk, 68.38 GiB in size, used only to store the files used in testing.

Every data point shown is the average of three distinct tests, in order to ensure the effects that are visible in the data are not the result of any particular test run, but instead are consistent and repeatable.

5.1.1 Automatic Testing

To guarantee a clean test environment, the machine was rebooted between each test. However, the machine used for testing took around 10 minutes to reboot, so manually running multiple tests in a

row quickly became very tedious. In order to run many tests in a row without human intervention, I wrote a collection of scripts in Python 3 and Bash to measure the differences between file accesses on different nodes. Each test run was started by an `rc.local` script that would run on each boot. This script would run `autotest.py`, which would then look for new tests to be run based on the file names within the test-run folder. The name of each test run would inform the script which version of ZFS needed to be loaded, either a modified version, or a standard release version of ZFS to compare against, which test to run, and what file to test against.

The script automatically rebooted the machine at the end of each test. However, care was taken to ensure that it would not reboot if there were no more test runs needed, so that the machine would not enter a boot loop and become inaccessible.

A ZFS filesystem was created with ZFS version 0.8.5 and populated with a set of files full of randomly generated data of various sizes, which were then used to test the performance of reading files from ZFS.

The `numactl` program was used to control which processors the test program ran on, as it allows you to set what nodes a program is permitted to run on before executing it.

Either `fio`[3] or `simpleread` (Listing 5.1) was used as the test program, with `fio` measuring itself and `simpleread` timed by the `time` command.

`fio` is the Flexible I/O Tester, written by Jen Axboe[3]. It is a powerful tool for benchmarking the performance of every possible kind of read and write operation on Linux. I simply used its sequential read capabilities with known files on my test ZFS filesystem to get more accurate measures of latency and bandwidth, as `fio` accurately times each individual read operation in order to calculate its statistics. Through these measurements it can calculate various percentiles of latency and also read bandwidth. These metrics were invaluable for detecting differences in read operations between my prototype and normal ZFS.

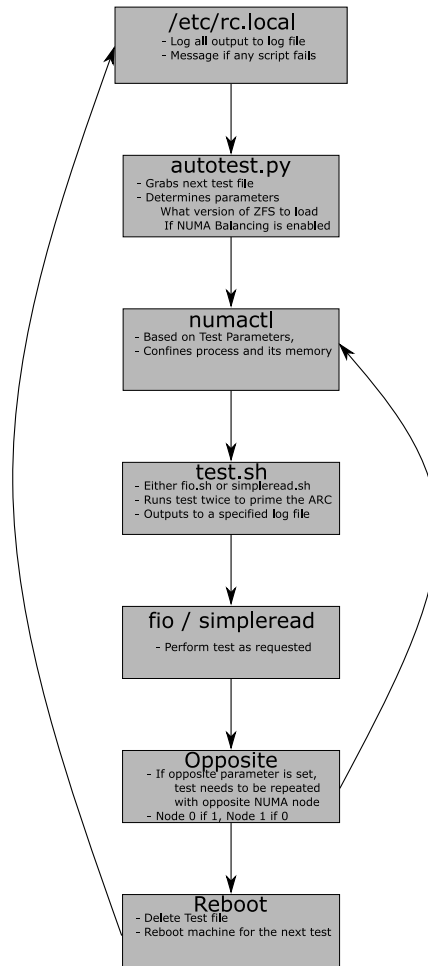


Figure 5.1: A flowchart of how automatic testing worked.

5.2 Demonstrating the Effects

As a direct demonstration of the effects of NUMA on ZFS, consider this simple file-reading program (Listing 5.1). It takes a filename and a read size, how many bytes to read at once, and reads through the entire file sequentially before exiting.

```

1 #include <errno.h>
2 #include <fcntl.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>

```

```

7 #include <unistd.h>
8
9 #ifdef HUGE
10 // Borrowed from Stack Overflow
11 // https://stackoverflow.com/questions/43520681/increase-binary-executable-size
12 char huge[HUGE] = {'a'};
13 #endif
14
15 int main(int argc, char **argv) {
16     if (argc < 3) {
17         fputs("simpleread <filename> <readsize (in bytes)>\n", stderr);
18         return 1;
19     }
20
21     char *filename = argv[1];
22     int readsize = atoi(argv[2]);
23
24     int fd = open(filename, 0);
25     if (fd < 0) {
26         fprintf(stderr, "Open file \"%s\"\n", filename);
27         perror("simpleread open file");
28         return 2;
29     }
30
31     /* char is always 1, so we can skip sizeof */
32     char *buf = malloc(readsize);
33     if (buf == NULL) {
34         perror("simpleread buf malloc");
35         return 3;
36     }
37
38 #ifdef HUGE
39     for (int i = 0; i < HUGE; i++) huge[i] = i % 255;
40 #endif
41
42     size_t bytes = 1;
43     while (bytes > 0) {
44         bytes = read(fd, buf, readsize);
45     }
46
47     if (errno != 0) {
48         printf("%d\n", errno);
49         perror("simpleread read");
50         return 4;
51     }
52     return 0;
53 }

```

Listing 5.1: Simple Read Program

In order to measure the effects of memory latency on ZFS, I generated files full of random data

on a ZFS filesystem. To get a sense of how file size affected the results of my tests, I generated files ranging in size from 128 KB to 16 GB, doubling in size each time. However, the tests presented typically show results from 250 MB and up, as this is where the effects of NUMA become most visible. Tests involved reading these files with various programs, one file per boot to make sure the ARC and other kernel caches were fully flushed between tests. I ensured the ARC data being read was located on the right node by taking advantage of the fact that ZFS will initially load data into the ARC on the same node as the program that first requests it. Thus I simply ran this program once, bound to the correct node, before taking any measurements in order to ensure a consistent state of the ARC.

Using run time as a coarse measurement of latency, we find that this program takes less time if run on the same node as the ARC data it is consuming, when that ARC data is 1 GB or larger. It improves from about 10% less time for a 1 GB file up to a maximum of 15% less time for a 16 GB file (Figure 5.2).

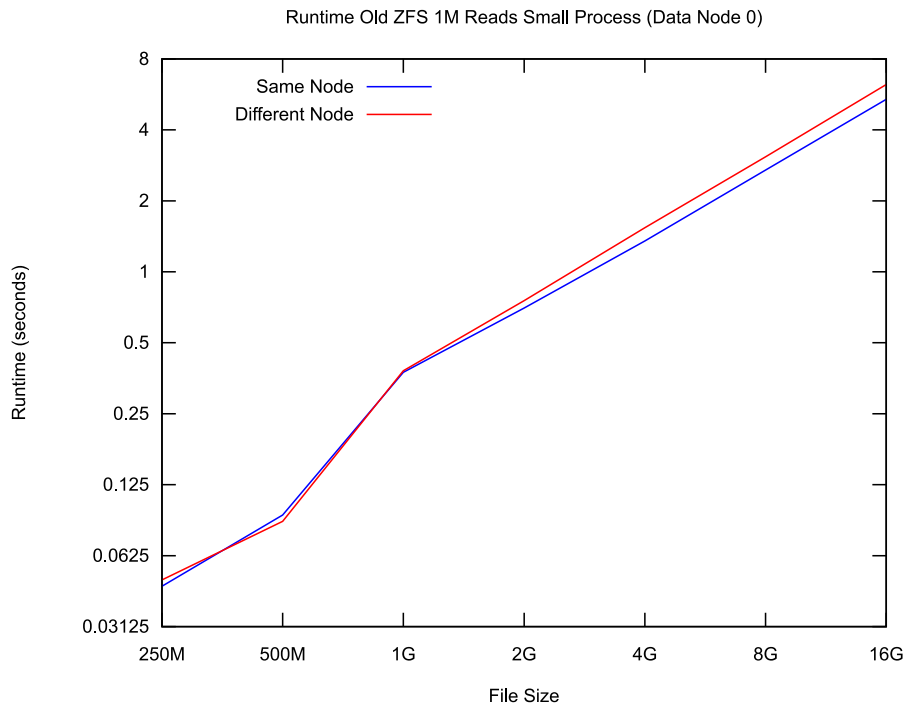


Figure 5.2: Reading a file from a different node than the one its ARC data is stored on causes a 10-15% loss of performance, proving that ZFS, specifically its ARC, is affected by NUMA.

5.3 How to Mitigate the Effects

Fundamentally, the performance loss observed comes from the need to pass data across the interconnect between NUMA nodes. The data needs to be close to the process that is reading it in order to get the best performance.

When we find a process in this situation, reading data from another node, there are two ways we can fix it. Either all the data the process needs can be moved closer to the process, or we can move the process closer to its data. Regardless, some data will need to move across the interconnect, as the process's other memory allocations should continue to be local to it, and not have to be accessed via the interconnect, as otherwise this will incur the same performance penalty we are trying to avoid.

Chapter 6

NUMA Balancing

Linux attempts to prevent this kind of performance loss by tracking remote node memory accesses, and automatically moving memory to the node that uses it most [4, Documentation/sysctl/kernel.txt]. It tracks this by periodically marking pages inaccessible in the page table and trapping the eventual page fault that occurs when a process tries to access it. It then keeps statistics on which NUMA node the process is running on when it accesses memory. A periodic scanning thread looks at these statistics for each process and decides if a page needs to be moved based on how often it was accessed from a different node. However, this operates only at the page level, meaning that by the time it figures out that a page needs to be moved, it is often too late to help speed up the memory accesses of a particular process. It also only scans once every second by default, so if your program takes less time than that the memory it accesses is not even considered for migration[4, kernel/sched/fair.c].

Enabling NUMA Balancing and using the same simple read program as before, at best it manages to regain 3% of the lost performance (Figure 6.1). This is likely within the margin of error for these tests, as due to the way that the ARC allocates memory (Section 3.9), it is given physical pages by the Linux kernel and thus the kernel is unable to move them around as it would be if the ARC asked for virtual memory.

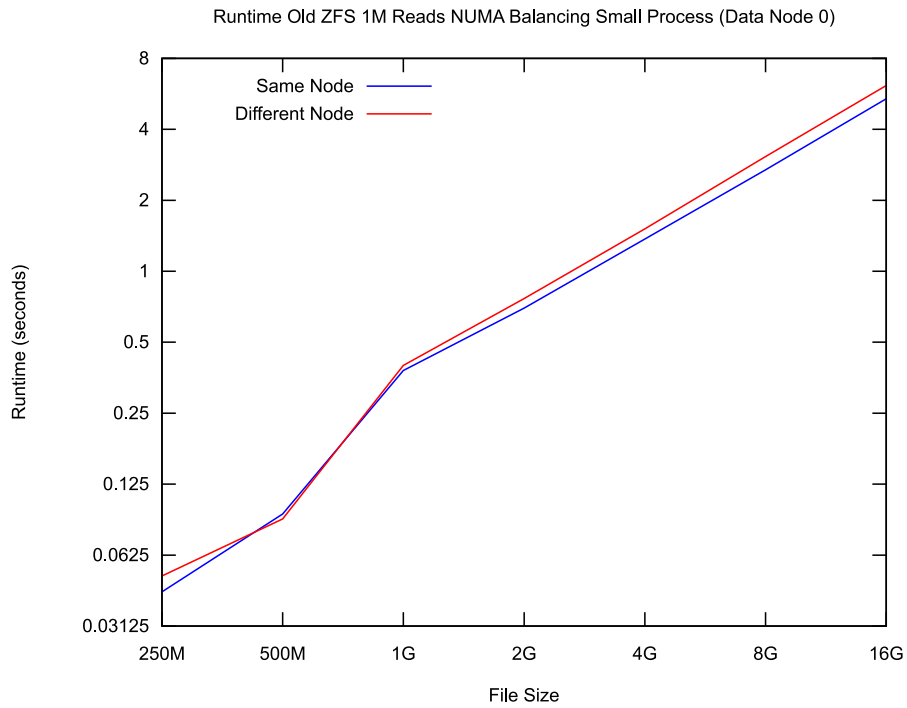
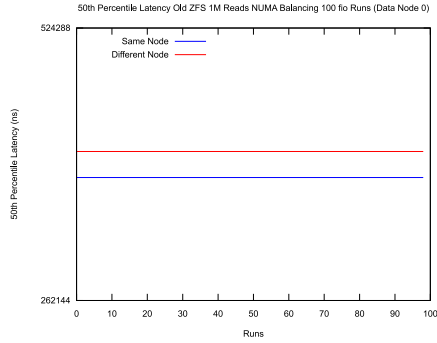
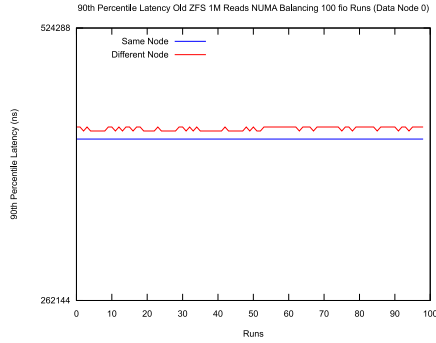


Figure 6.1: Repeating the previous testing with NUMA Balancing enabled does not improve performance when accessing the ARC from a different node.

Other testing repeatedly reading a file from a different node with NUMA Balancing enabled confirms this, as the read latency never improves and is always higher than a read from the same node as the ARC data.



(a) 50th percentile latency sees no change at all over 100 runs with NUMA Balancing enabled.



(b) 90th percentile latency sees no significant change at all over 100 runs with NUMA Balancing enabled.

Figure 6.2: Repeatedly reading data from the ARC does not trigger NUMA Balancing, likely because it does not consider kernel memory like the ARC. If NUMA Balancing was occurring the different node latency would converge towards the same node latency. Note that the same node latency is only from the initial test run on the same node after priming the ARC to ensure that it would not interfere with NUMA Balancing by adding more reads from the same node. It is extended across the graph to provide a point of reference.

Investigating the Linux NUMA balancing code confirms what this graph show by experimentation, that ARC memory is not considered. Only memory that is part of a specific process is considered and even then only after that process has been running for one full second [4, kernel/sched/fair.c].

Chapter 7

Task Migration

Task migration is moving a process automatically closer to the data that it is reading, if we find that the process is on a different node than that data. This requires storing the current node number in the ARC header structure, and moving the current task and all of its memory to that same node if it is not running there already. This will invalidate the CPU caching that has built up over the course of the process's current run time, but will guarantee that the process will be local to the chapter of the ARC currently being accessed.

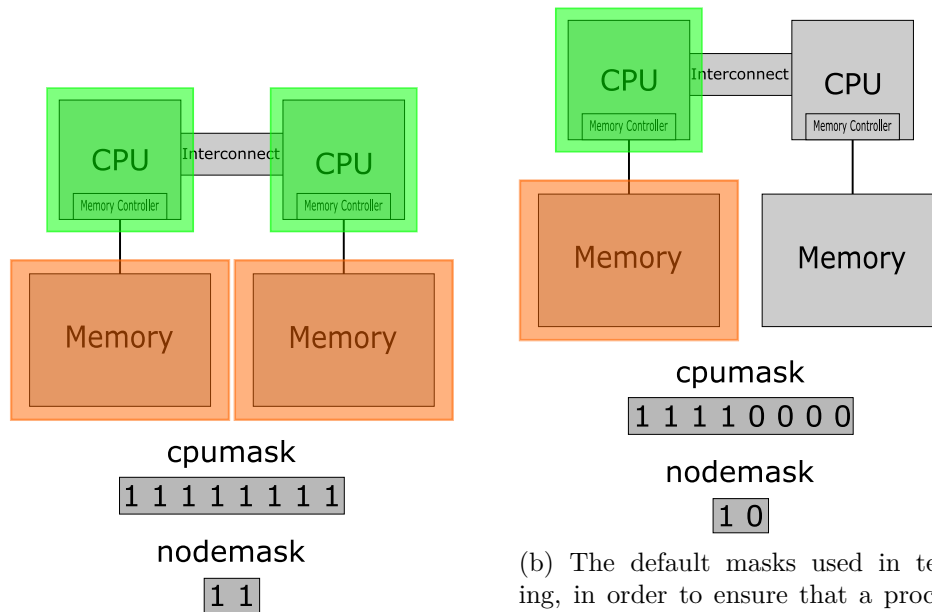
My expectation was that above a certain threshold of file size, the overhead from remote node accesses would be high enough that a significant performance improvement would be visible in the results. Ideally, this would mean a restoration of all lost performance from these remote node accesses. Assuming the overhead from moving the process and its memory is not too high, then a performance improvement should occur, as future memory accesses to that same part of the ARC become local node accesses instead of remote node accesses.

7.1 Implementing Task Migration

In order to implement task migration, I had to modify both the Linux Kernel and OpenZFS. The initial step was to add code allowing ZFS to migrate the current process to another node, as such functionality is not currently available to kernel modules like ZFS in Linux. Linux is rather conservative about the functionality it exposes to kernel modules, and this allows one to take fairly direct control over the location of any process's memory, which is more than any kernel module

would typically need. Something like this would usually be implemented in the scheduler or the page cache, both of which are part of Linux itself, and thus would not need this functionality exported. This required three steps, first moving the process to the correct NUMA node, then ensuring it will allocate new memory only from that new node, and finally moving all of the process's current memory allocations to the new node.

Every process in the Linux kernel has a task structure, which contains various pieces of information about that task. By modifying this structure inside Linux or one of its modules, like ZFS, one can control how the process is scheduled, and where its memory is located. This structure for the current process, in whose context this code is executing, can always be referred to by the `current` macro within the Linux kernel, or, within the SPL or ZFS as `curthread`, because that is what the structure was called on Solaris.



(a) The default processors and memory a process is allowed to use is typically all of them available on the system.

(b) The default masks used in testing, in order to ensure that a process was running in a known location. This configuration was also mirrored to the other node to ensure that there were not differences between nodes, but they performed identically throughout testing.

Figure 7.1: Each process has a specific set of processors it is allowed to run on and NUMA nodes that it is allowed to allocate memory from.

The first step, changing where a process is running, was quite straightforward. Linux provides

a convenient interface to inform the scheduler where a process is allowed to run. By default, a program is allowed to run anywhere (Figure 7.1a), but for testing purposes programs were typically restricted to one NUMA node at the start, in order to simplify data collection (Figure 7.1b). By restricting this set of processors to only those in a specific NUMA node, we can ensure that the scheduler runs a process on that specific node. This interface is a bitmask, a set of bits, one per each processor in the system, where each bit that is set to 1 is a processor that is allowed, and each bit that set to 0 is a processor that is not allowed. Each NUMA node typically has multiple processors, in the case of my test system each node has four processors. Linux also provides a convenience function, `cpumask_of_node`, that can convert a simple node number to a bitmask of processors, referred to internally in the Linux kernel as a cpumask. This cpumask can then be passed, along with the task structure of the current process, to the `set_cpus_allowed_ptr` function, which will correctly set the cpumask of process whose task structure is passed to it (Listing 7.1).

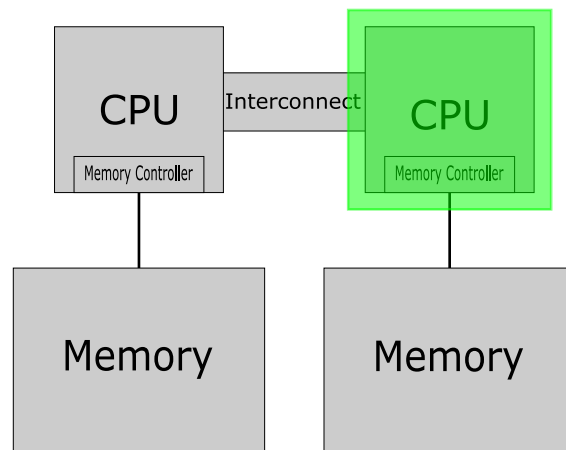


Figure 7.2: A visual representation of what calling `set_cpus_allowed_ptr(curthread, cpumask_of_node(1))` does to the allowed processors of the current process.

```

1 // include/linux/sched.h
2 struct task_struct {
3 // ...
4     int         nr_cpus_allowed;
5 // ...
6     cpumask_t   cpus_mask;
7 // ...

```

```

8 }
9 // kernel/sched/core.c
10 // set_cpus_allowed_ptr calls __set_cpus_allowed_ptr(p, new_mask, 0)
11 /* Change a given task's CPU affinity. Migrate the thread to a
12 * proper CPU and schedule it away if the CPU it's executing on
13 * is removed from the allowed bitmask.
14 */
15 static int __set_cpus_allowed_ptr(struct task_struct *p,
16                                 const struct cpumask *new_mask, bool check)
17 {
18     const struct cpumask *cpu_valid_mask = cpu_active_mask;
19     unsigned int dest_cpu; struct rq_flags rf; struct rq *rq; int ret = 0;
20     rq = task_rq_lock(p, &rf); update_rq_clock(rq);
21     // ...
22     do_set_cpus_allowed(p, new_mask); // See Listing 3, new_mask is from cpumask_of_node
23     // ...
24     /* Can the task run on the task's current CPU? If so, we're done */
25     if (cpumask_test_cpu(task_cpu(p), new_mask)) goto out;
26     if (task_running(rq, p) || p->state == TASK_WAKING) {
27         struct migration_arg arg = { p, dest_cpu };
28         /* Need help from migration thread: drop lock and wait. */
29         task_rq_unlock(rq, p, &rf);
30         stop_one_cpu(cpu_of(rq), migration_cpu_stop, &arg);
31         return 0;
32     } else if (task_on_rq_queued(p)) { // dropping lock immediately anyways
33         rq = move_queued_task(rq, &rf, p, dest_cpu);
34     }
35 out:
36     task_rq_unlock(rq, p, &rf);
37     return ret;
38 }

```

Listing 7.1: The relevant chapters of Linux’s scheduler code that handles a change in the cpumask indicating where a process is allowed to execute

```

1 // do_set_cpus_allowed always calls this function
2 // A structure of function pointers is involved,
3 // but for the normal scheduler it simply points to this function
4 void set_cpus_allowed_common(struct task_struct *p, const struct cpumask *new_mask)
5 {
6     cpumask_copy(&p->cpus_mask, new_mask);
7     p->nr_cpus_allowed = cpumask_weight(new_mask);
8 }

```

Listing 7.2: The relevant chapters of Linux’s scheduler code that set a process’s cpumask

The second step, changing where the process was allowed to allocate new memory from, was quite similar. As with the processors allowed, the task object also contains a bitmask indicating which memory it is allowed to use, which also defaults to all nodes (Figure 7.1a), but was set to one

node for testing purposes (Figure 7.1b). This takes a slightly different kind of bitmask, a nodemask that indicates which nodes the process is allowed to allocate memory from, and it also only operates on the current process.

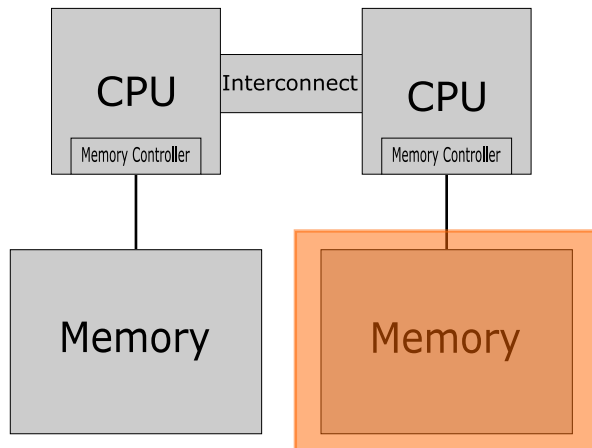


Figure 7.3: A visual representation of what calling `set_mems_allowed(nodemask_of_node(1))` does to the allowed memory of the current process.

```

1 // include/linux/sched.h
2 struct task_struct {
3 // ...
4     /* Protection against (de-)allocation: mm, files, fs, tty, keyrings, mems_allowed,
5        mempolicy: */
6     spinlock_t    alloc_lock;
7 // ...
8     nodemask_t    mems_allowed;
9     /* Sequence number to catch updates: */
10    seqcount_t     mems_allowed_seq;
11 // ...
12 }
13 // include/linux/cpuset.h
14 static inline void set_mems_allowed(nodemask_t nodemask)
15 {
16     unsigned long flags;
17
18     task_lock(current);
19     local_irq_save(flags);
20     write_seqcount_begin(&current->mems_allowed_seq);
21     current->mems_allowed = nodemask;
22     write_seqcount_end(&current->mems_allowed_seq);
23     local_irq_restore(flags);

```

```
23     task_unlock(current);
24 }
```

Listing 7.3: The relevant chapters of Linux’s scheduler code that handle the nodemask indicating where a process is allowed to allocate memory

The third step, however, was quite a bit more involved. Linux does have a system call, `migrate_pages`, that allows a user to move the memory of a program from one node to another. However, system calls like `migrate_pages` cannot be called directly from kernel modules like ZFS for a variety of reasons, but knowing that this functionality already existed enabled me to look into where in the Linux kernel this system call was implemented.

It turns out that this system call is just a wrapper around the `kernel_migrate_pages` function, but this did not help as it still took arguments in the same way as `migrate_pages`, so it could not be called from ZFS. However, within this function was the `do_migrate_pages` function, which does all of the actual work to migrate all of a process’s memory from one node to another. This structure of function calling `kernel_function` calling `do_function` is a common construct in the Linux kernel. This function could actually be called from ZFS, but was not exported to kernel modules, as it had no security checks, these all having been handled by `kernel_migrate_pages`.

```
1 diff --git a/mm/mempolicy.c b/mm/mempolicy.c
2 index d0d24abd1733..8d5df0246151 100644
3 --- a/mm/mempolicy.c
4 +++ b/mm/mempolicy.c
5 @@ -1154,6 +1154,7 @@ int do_migrate_pages(struct mm_struct *mm, const nodemask_t *from,
6     return busy;
7
8 }
9 +EXPORT_SYMBOL_GPL(do_migrate_pages);
10
11 /*
12  * Allocate a new page for page migration based on vma policy.
```

Listing 7.4: The entirety of my prototype patch to the Linux kernel, enabling the SPL to migrate a process’s memory from one node to another. An actual patch to export this functionality could not just export this function. It would require another function with error checking to prevent misuse and would likely still not be accepted, as kernel modules are generally not allowed to modify the scheduler or move around the memory of a user’s process in this way.

Once I could access this function from within ZFS, I added two new functions to the SPL, the Solaris Portability Layer that is the interface between the Linux kernel and ZFS. The first

was `spl_migrate_pages`, which simply did the work of the `kernel_migrate_pages` function, taking actual node numbers and a task structure and converting them to the arguments that `do_migrate_pages` actually takes, that is a common structure in Linux kernel interfaces.

```
1 void
2 spl_migrate(int node)
3 {
4     if (node >= nr_node_ids || node < 0) {
5         pr_warn("SPL: Can't migrate to node %d!\n", node);
6         return;
7     }
8     set_cpus_allowed_ptr(curthread, cpumask_of_node(node));
9     set_mems_allowed(nodemask_of_node(node));
10    spl_migrate_pages(curthread, node);
11    if (curnode != node) {
12        pr_err(KERN_ERR "SPL: Failed to migrate task %s!\n", curthread->comm);
13        dump_stack();
14    }
15 }
16 EXPORT_SYMBOL(spl_migrate);
```

Listing 7.5: The `spl_migrate` function, used to move the current process to another NUMA node

Now that the migration function was in place, it needed to be called from within OpenZFS. I first added a node number to the ARC header, the metadata structure for each block stored in the ARC discussed above (Section 3.9). Then, I modified the allocation method for ARC Buffer Data, to always prefer the current node when allocating memory (Section 3.9.1). When the allocating large ABDs, ZFS attempts to allocate them in large chunks, as big as the kernel will allow to fit all the data required. However, if the data required is larger than the kernel has available in contiguous physical pages in memory then it will split up the buffer into a series of chunks, each of which points to the next, but can be used as one big buffer within ZFS thanks to some abstractions built on top of this structure.

When allocating these chunks, ZFS will allocate from wherever is available, defaulting to the local node but detecting when Linux is unable to get enough memory from the local node and instead allocating from whatever remote node succeeded at allocating the previous large chunk. This could result in ABDs not located on the same node as the current process at all, and in the worst case could result in different parts of a buffer allocated on entirely different nodes, which

would be extremely difficult to handle. To simplify this I removed this allowance, ensuring that this code would always try to allocate buffers from the current node. This would only have come into effect if one NUMA node ran out of memory. Finally, I modified the `arc_read` function, which is called by the DMU if it cannot find the data it is looking for in its local cache. In the code path for returning data already in the ARC, I added code to check if the node number of this found ARC header was the same as the current node number that the process was running on, and move the process if they were different For the specifics of this patch see Appendix A Listing A.8.

Chapter 8

Results

With task migration enabled in ZFS as implemented in the above prototype, we see significant improvements in runtime and latency for sequential file-access in many cases. Repeating the same tests used above to show a difference in local and remote node ARC access, we see a 10% improvement in runtime for files larger than 1 GB with a 1 MB read size that grows to 15% as file size increases (Figure 8.1).

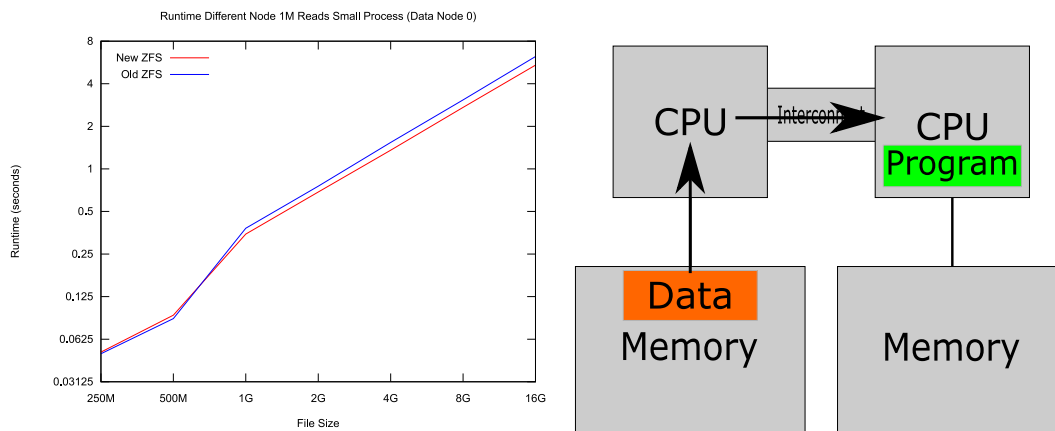


Figure 8.1: Comparing ZFS with a modified version using task migration, a 10-15% improvement is visible with task migration for files 1 GB and larger when the ARC data being accessed is on another NUMA node.

Most importantly, when comparing the results of remote node ARC access with task migration enabled to the optimal results running on the same node, we see that the task migration has managed to regain all of the performance that would otherwise be lost when accessing memory

from a different node (Figure 8.2).

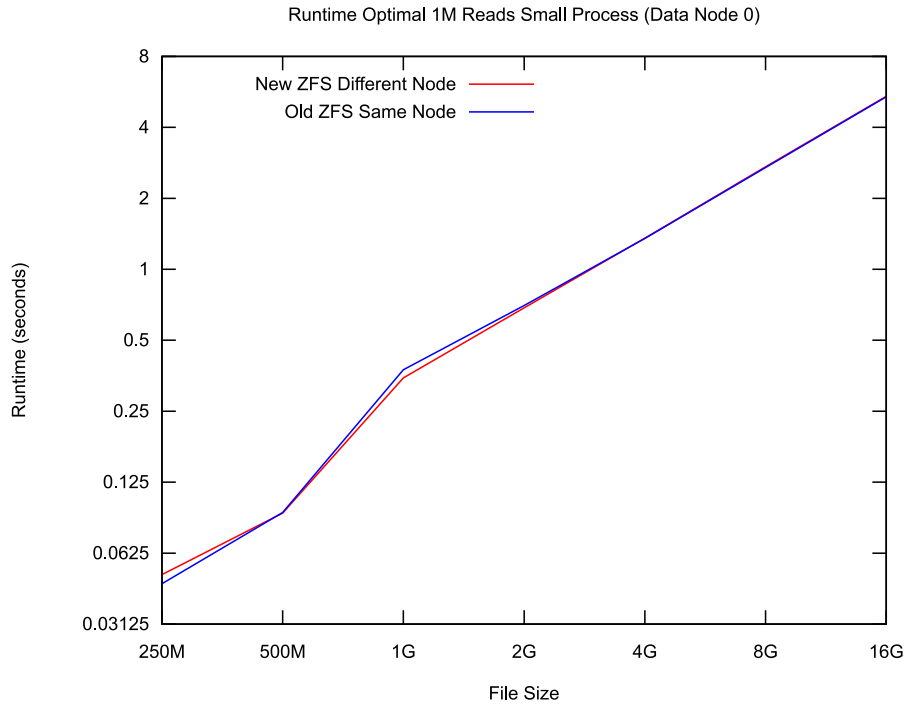


Figure 8.2: Task migration with the ARC on a different node compared to starting the process on the same node. The results are practically identical, showing that task migration can significantly improve performance without substantial overhead in this scenario.

The bandwidth of the reads, as measured with `fiio`, also substantially improves (Figure 8.3). Given that the memory and interconnect bandwidth on the test system are essentially the same, 31.99 GB/s for the 48GB of DDR3 1066 RAM and 25.60 GB/s for the QPI, one would not expect use of the interconnect to affect bandwidth very much, with only about a 6% difference between the two[12]. However, testing shows that bandwidth is affected in a similar way as latency, with a fairly consistent 10% improvement.

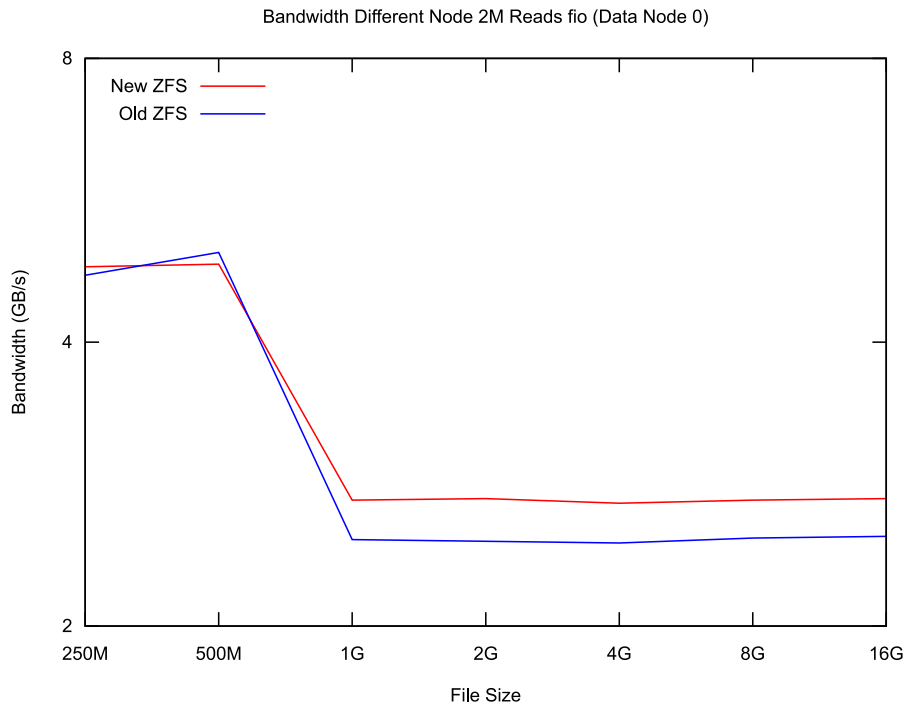


Figure 8.3: Bandwidth improves almost as much as latency when accessing ARC data from a remote node with task migration. Note that unlike most other graphs in this work this uses 2 MB reads, but the results are consistent with 1 MB reads tested with previous versions of my prototype.

This is a surprising result, as the theoretical memory bandwidth differences would indicate a much smaller improvement. However these numbers are consistent with other research into NUMA systems, as the full bandwidth of the interconnect is shared between the nodes and thus highly variable depending on what other process running at the same time happen to require[8, 15, 19]. These bandwidth results are also consistent with other tests performed on this machine, such as a single-threaded STREAM benchmark or the Intel Memory Latency Checker. Both show a significant reductions reduction in bandwidth for remote node memory access, as discussed previously (Section 4.1). Bandwidth, like latency, is restored to same level as if the ARC data and the program were originally located on the exact same NUMA node (Figure 8.4).

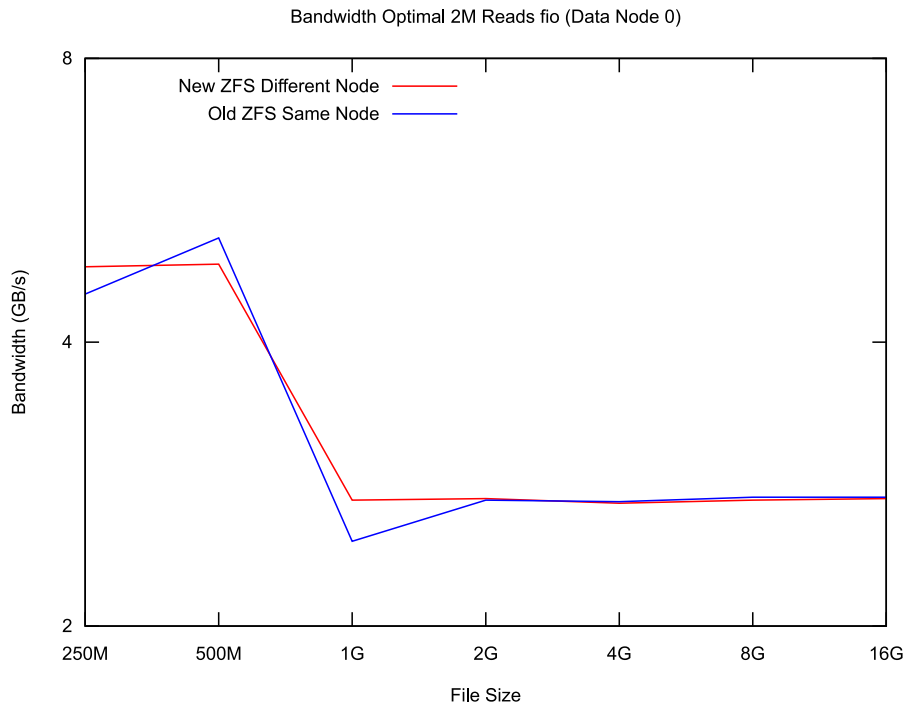


Figure 8.4: Bandwidth, as with latency, is identical when on a different node with my task migration prototype as compared on reads from the same NUMA node.

8.1 Impact of Process Size

The improvements seen with task migration enabled are not universal, however, and the actual improvement visible to user programs is highly dependent on the size of the program. This is because when a process is moved to a different NUMA node all of its other memory allocations must be moved with it in order to see the full benefits of local memory access. Otherwise, the process will continue to cross the interconnect in order to access any memory that it allocated previously, and continue to incur the latency and bandwidth penalties associated accessing memory from a different NUMA node. By creating variants of my simple read program of various sizes, I was able to measure the difference in runtime improvement that resulted from differences in program memory size.

In order to test different process sizes, a large array was added to the program, that was then

filled with data so that it would not be optimized away by the compiler. The size and existence of this array was determined by the `HUGE` macro, defined on the `gcc` command line with `-D`, so that different sized programs could be built from the exact same source code.

Given that accessing ARC data for a file 1 GB or larger across the interconnect causes an approximately 10% increase in runtime, which improves even more as the file size does, as discussed previously, transferring a larger process closer to that same size will likely have a similar impact, reducing the improvement possible from task migration. With that knowledge, I can attempt a theoretical model of the performance improvement depending on the size of the process and the file.

I = Performance Improvement

M = Maximum Possible Improvement

P = Process Size in Memory

F = File Size

R = Reduction Factor

Given these variables, I would expect performance improvement to be determined by something like the following:

$$I = M - \left(\frac{P}{F} \times R \right)$$

From this theoretical model I would expect no improvement if the process and the file are the same size, and to see the improvement slowly reduced as the size of the process in memory increases. Fundamentally, what causes the performance improvement visible in these results is moving less data across the NUMA interconnect. When the process is larger, the amount of data that needs to be moved increases, reducing the possible improvement by some reduction factor R .

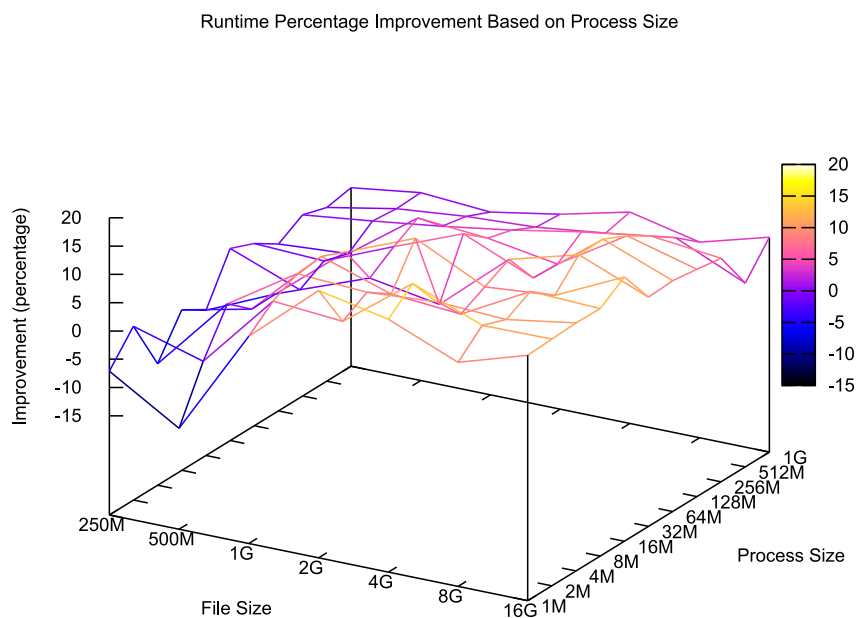


Figure 8.5: The Effect of Process Size on the Percentage Improvement in Runtime. Measuring using the simple read program sized to various process sizes, with a read size of 1M.

Reductions in runtime improvement are immediately visible with larger processes, and by a much larger reduction factor than I expected (Figure 8.5). At 1G in size, runtime improvement is reduced to 2-5% for files larger than 1 GB and no improvement is visible with a 1 GB file and a process of the same size. Regardless of where it runs, a program that has 1 GB in memory and reads a 1 GB file stored on another NUMA node must still move 1 GB of data across the interconnect regardless of whether Task Migration is enabled. This aspect of the model was correct, but the prediction for smaller processes was not. Smaller process sizes do slowly regain the performance improvement that was visible in the earlier tests, as they get smaller, but do so much more slowly than expected, indicating that the reduction factor is quite high. (Figure 8.1). This performance improvement is also highly variable across test runs, resulting in a graph that is difficult to make definitive conclusions about. This is likely due to the loop over the large memory area that had to be added to the program in order to ensure it would not remove it as unnecessary when compiled (Listing A.5, line 39).

8.2 Impact of Read Size

The size of the process is not the only factor in determining how the runtime of the process will be affected by task migration. Another important factor is the size of the individual file reads that the process uses to sequentially read through it.

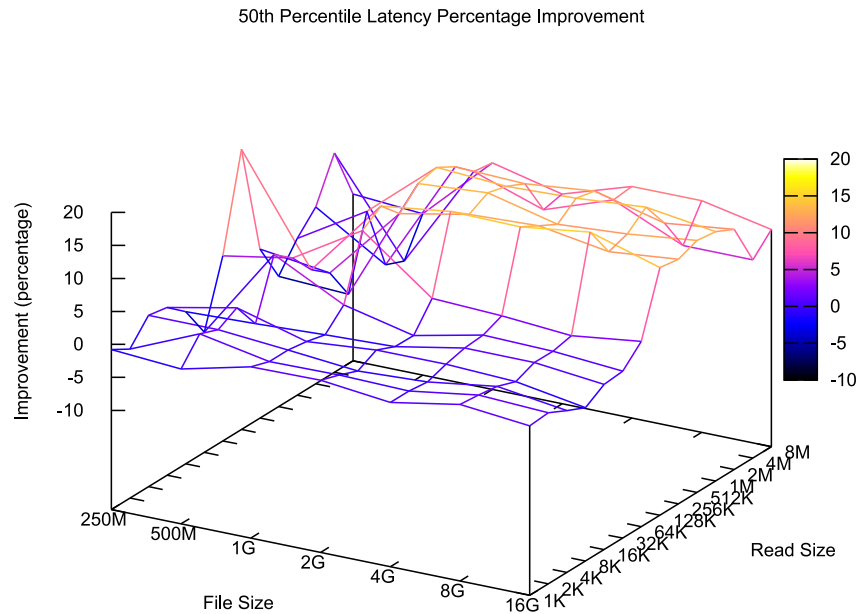


Figure 8.6: Impact of Read Size

Performance improvement from this prototype only occurs when a program reads 128 KB or more at a time (Figure 8.6). This seems to indicate that the ARC or some layer below it may be handling read sizes of less than one block differently than those reading one block or larger. Once at this read size the performance improvement increases to the 10-15% threshold which is the maximum possible as shown by previous testing (Figure 5.2).

When reading 1-2 MB at a time, there appears to be a peak where significantly more performance is regained (Figure 8.6). This ideal case seems to be caused by prefetching within ZFS, which does not consider any reads that are above 1 MB [6, module/zfs/dmu_zfetch.c]. Within the DMU, all reads smaller than this are tracked and organized into sequential streams in order

to predict what reads will happen next and ensure that the next blocks in a sequential read are already available (Section 3.8.3). This prefetching code seems to be a bottleneck in the ZFS code for sequential reads, as it does a significant amount of accounting around every read operation in order to detect sequences. However, it only calls `arc_read` from the context of the original read on blocks it detects will be next in the sequence in order to load them into the ARC. Thus, this is simply a property of ZFS itself and not related to my changes, though it is an interesting result.

Chapter 9

Conclusion

As this work has shown, Non-Uniform Memory Access has a significant impact on ZFS, a filesystem that has significant in memory caches to speed up operations. While reading files from memory is still significantly faster than reading them from disk, ensuring that programs and files end up on the same NUMA node provides significant improvements to read latency and bandwidth. In an attempt to improve ZFS's performance in these situations, I have prototyped one way to reduce accesses to ARC data stored on a different node. This solution requires invalidating CPU caches by moving a process to a CPU on a different NUMA node, but it does provide a performance improvement when the files being read are very large, over 1 GB, and they are being read in large amounts at a time, over 128K. Because a process is often smaller than the files that it is reading, moving the process instead of its memory can reduce the amount of memory moving across the interconnect, limiting the latency impact.

While NUMA balancing is a known approach already implemented in the Linux kernel to mitigate this problem for a process's own memory, it does not consider external caches that the program accesses such as the ARC in ZFS. Task migration is an alternative solution that may provide some benefit in very specific situations, especially when the amount of memory being accessed is very large and the process's other memory allocations are very small. Future research will determine if these two approaches can work together to improve the performance of a system. NUMA balancing, if applied to the ARC as well, would likely show very similar results to those presented in this work.

These results should not be taken as proof that task migration is always the better approach. For situations like the ones tested in this work, with large in-memory structures, it may be valuable for operating systems to consider the relative size of a process and the memory it believes that process will access from another node before deciding which to move when it finds a program accessing large amount of memory from a different NUMA node.

Another important contribution of this work is to show that there is much room for improvement in large in-memory caches such as the ARC when it comes to NUMA. While ZFS does take NUMA into account when first placing data into the ARC, it is clear that there is still much more that could be done for later processes reading this data who might not be on the same node, as my performance measurements of ZFS have shown. As the large systems that have necessitated a NUMA architecture are also the kinds of systems where a filesystem like ZFS is often used to store large amounts of data, this improvement would be beneficial for many of its users.

9.1 Future Work

9.1.1 Read Size Limitations

The impact of read size (Figure 8.6) is an unfortunate wrinkle in otherwise very promising results. The impact of NUMA on smaller read sizes seems more limited than with larger sizes, so this may in fact be a reflection of the limited gains that can be made with smaller read sizes. As discussed previously, the inflection point at 128 KB would suggest that the ARC is handling reads smaller than one block differently, as 128K is the default maximum block size (Section 8.2). Also the 1 MB inflection point for best improvement overlapping with the point at which the prefetcher is disabled indicates that its behavior needs to be taken into account and likely modified. Like the ARC, the prefetcher likely needs to be aware of which NUMA node the program accessing a particular file is running on and work to allocate ARC memory closest to that program.

9.1.2 Impact on Arbitrary Programs on Long-Running Systems

The testing environment for these improvements has been, by necessity, rather academic in order to eliminate the influence of other factors. However, no one reboots these machines every time

they need to run a program, and so testing this against arbitrary programs with an ARC of a long-running system would be useful future work to prove or disprove whether it can improve real-world application run times without impacting other programs on the system too much. It may be that micro-managing the scheduler in this way leads to unbalanced systems over time, as programs that access the same files are all forced to run on the same node.

The impact of this management style may also be positive, as the scheduler has no knowledge of NUMA memory accesses, and might schedule applications poorly when they are frequently requesting data from another node. As Megiddo and Modha write in their original paper on the ARC, “real-life workloads possess a great deal of richness and variation, and do not admit a one-size-fits-all characterization” [17].

9.1.3 Interactions with NUMA Balancing and Swapping

The current prototype implementation assumes blocks don’t move in memory, which may not be the case when NUMA balancing is enabled, or when swapping occurs. Retrieving the exact location of the physical buffer before checking it against the current node would partially relieve these issues, but it does not solve the problem of potentially splitting a buffer across multiple nodes, which would require handling each buffer of an ABD separately when considering whether to move a process.

This could be partially handled by locating the page of ARC header’s physical buffer at read time, but doing in the standard read code path is likely to be a very expensive operation that could reduce the amount of possible improvement by taking significantly more time just to determine this information. This approach would allow NUMA balancing to be re-enabled and work with task migration, though more changes would also need to be made to allow NUMA balancing to consider ARC pages as well.

9.1.4 More Complex NUMA Topologies

Due to only having access to one system with Non-Uniform Memory Access, it is difficult to make statements about how these changes might affect systems with more NUMA nodes and different interconnect latencies. Systems that have higher interconnect latencies between their nodes will

probably see larger improvements from these changes, while systems with lower interconnect latency will probably see less improvement. I would expect this general principle to hold no matter how many NUMA nodes a system has, but actually testing it may show different results. For example, there may be more contention on the interconnect with more NUMA nodes, which could affect latency in measurable ways. Thus comparing the results I saw in my testing with results from systems with more complex topologies could provide interesting insight on the impact of this added complexity on the system's memory latency.

9.1.5 Applicability to More General File Caching Systems

The general idea of an in-memory cache is a common feature of filesystems, with most on Linux using the page cache. The Linux page cache uses two CLOCK lists that approximate two LRU caches, one for pages accessed once, and another for pages accessed multiple times. While this approach appears superficially similar to the ARC, it does not have the dynamically resizing capabilities of the ARC that allow it to learn from past cache misses through a ghost cache (Section 3.9). While the ARC is a smarter caching system, the page cache operates with the same general principles, and so the ideas presented here may be just as applicable to the page cache.

9.1.6 Global View of the ARC and of Files

The current implementation is greedy, considering only one block at a time and believing every block to be equally important, an equal indicator that a process is going to read an entire file. It also does not consider the size of the file when deciding to move a process, which results have shown is a critical factor in whether moving a process is actually helpful.

With input from the prefetching code, it could be made smarter, only moving a process after it shows itself to be likely to read most of a file. However, waiting for a certain amount of access also guarantees that those first few accesses are going to be high latency if from a remote node, and might reduce the impact of this improvement on overall average latency and runtime.

The ARC is specifically designed to be concerned only about blocks, and leaves the higher level concepts of files to the layers above it, specifically the DMU and ZPL. Thus the process migration

decision really should be happening at one of these higher levels, probably the DMU as that is where prefetching happens. However, ensuring that the DMU knows where the data it's getting from the ARC is would likely be difficult, as the DMU is designed to know very little about how the ARC operates, with good reason as it is a clear abstraction on top of a complex caching layer.

Appendix A

Source Code

A.1 Automatic Testing Code

```
1  #!/usr/bin/python3
2
3  import os
4  import subprocess
5  import sys
6  import time
7
8
9  def meminfo(what):
10     print(what)
11     for n in ["0", "1"]:
12         with open("/sys/devices/system/node/node"+n+"/meminfo") as f:
13             print(f.read())
14
15  def runtest(folder, node, script, datafile, nums, blocksize):
16     t = subprocess.run(["numactl", "-N", node, "-m", node,
17                        folder+script+".sh", "/tank/"+datafile, nums, blocksize],
18                       stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
19
20     if t.returncode != 0:
21         message(''ERROR
22 Return Code: {}
23 Node: {}
24 Script: {}
25 Datafile: {}
26 Nums: {}
27 Blocksize: {}
28 Error:
29 {}'.format(t.returncode, node, script, datafile, nums, blocksize, str(t.stdout)))
30     sys.exit(t.returncode);
```

```

31
32 def message(text):
33     print(text)
34     subprocess.run(["sendmessage.sh", text], check=False)
35
36 os.chdir("/home/jaadams/arctest/autoarctest/")
37
38 testruns = os.listdir("test-runs")
39
40 runsleft = len(testruns)
41 if runsleft == 0:
42     message("Test Run Complete!")
43     sys.exit(0)
44 elif runsleft % 5 == 0:
45     message("{} Tests Remaining!".format(runsleft))
46
47 testfile = testruns[0]
48 print(testfile)
49 meminfo("Begin")
50
51 test = testfile.split(".")
52 bound = test[0]
53 node = test[1]
54 script = test[2]
55 datafile = test[3]
56 nums = ""
57 opposite = False
58 blocksize = ""
59
60 if len(test) > 4:
61     nums = test[4]
62 if len(test) > 5 and test[5] == "0":
63     opposite = True
64 if len(test) > 6:
65     blocksize = test[6]
66
67 os.chdir("newdata")
68
69 # Unload modules to ensure consistent state for test
70 subprocess.run(["./scripts/zfs.sh", "-u"], cwd="/home/jaadams/zfs", check=True)
71
72 if bound == "B" or bound == "T":
73     subprocess.run(["./scripts/zfs.sh"], cwd="/home/jaadams/zfs", check=True)
74 else:
75     subprocess.run(["modprobe", "zfs"], check=True)
76
77 if bound == "M":
78     subprocess.run(["sysctl", "kernel.numa_balancing=1"], check=True)
79 else:
80     subprocess.run(["sysctl", "kernel.numa_balancing=0"], check=True)
81
82 meminfo("Load ZFS")

```

```

83
84 subprocess.run(["zpool", "import", "tank"], check=True)
85 meminfo("Import tank ZPool")
86
87 try:
88     os.mkdir(bound+node)
89 except FileExistsError:
90     pass
91
92 os.chdir(bound+node)
93
94 runtest(".././.././", node, script, datafile, nums, blocksize)
95
96 meminfo("After Test")
97
98 if opposite:
99     try:
100         os.mkdir("0")
101     except FileExistsError:
102         pass
103     os.chdir("0")
104     onode = "1" if node == "0" else "0"
105     runtest(".././.././", onode, script, datafile, nums, blocksize)
106     meminfo("After Opposite Node")
107     os.chdir("../")
108
109 os.remove(".././test-runs/"+testfile)
110 print(testfile, "Complete!", flush=True)
111 subprocess.run(["reboot"])

```

Listing A.1: Automatic Testing Python Script

```

1  #!/bin/sh
2
3  LOGDIR=/var/log/autoarctest
4  mkdir -p "$LOGDIR"
5  sleep 30
6  LOGFILE="$LOGDIR/$(date --iso-8601=seconds).log"
7  /home/jaadams/arctest/autoarctest/autotest.py > "$LOGFILE" 2>&1 \
8  || sendmessage.sh "$(printf "Test Run Error:\n%s" "$(tail "$LOGFILE")")"

```

Listing A.2: Automatic Testing rc.local

```

1  #!/bin/bash
2
3  if [ -z $1 ]
4  then
5      echo "Requires a Test File!"
6      exit 1
7  fi
8

```



```

 9 DIR="$(dirname "$0")"
10 NAME="$(basename "$1")"
11 BLOCKSIZE=4096
12 if [ ! -z $3 ]
13 then
14     BLOCKSIZE="{3/M/000000}"
15 fi
16
17 EXE="$(basename "$0")"
18 EXE="{EXE/.sh/}"
19
20 for p in pre post
21 do
22     OUT="$3$EXE.$NAME.$p$2"
23     if [ -f "$OUT" ]
24     then
25         echo "$OUT already exists!"
26         exit 1
27     fi
28     {
29         time "$DIR/simpleread/$EXE" "$1" "$BLOCKSIZE"
30     } > "$OUT" 2>&1
31 done

```

Listing A.3: Simple Read Test Script

```

 1 #!/bin/sh
 2
 3 if [ -z $1 ]
 4 then
 5     echo "Requires a Test File!"
 6     exit 41
 7 fi
 8
 9 NAME="$(basename "$1")"
10 BLOCKSIZE="4096"
11
12 if [ ! -z $3 ]
13 then
14     BLOCKSIZE="$3"
15 fi
16
17 for p in pre post
18 do
19     FILENAME="$3fio.$NAME.$p$2"
20
21     if [ -f "$FILENAME" ]
22     then
23         echo "ERROR: $FILENAME exists!"
24         exit 42
25     fi
26     fio \

```

```

27     --readonly \
28     --invalidate=0 \
29     --ioengine=psync \
30     --rw=read \
31     --name "$NAME" \
32     --filename "$1" \
33     --output-format=json \
34     "--bs=$BLOCKSIZE" \
35     --output "$FILENAME"
36 done

```

Listing A.4: Fio Test Script

```

1  #include <errno.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #ifdef HUGE
10 // Borrowed from Stack Overflow
11 // https://stackoverflow.com/questions/43520681/increase-binary-executable-size
12 char huge[HUGE] = {'a'};
13 #endif
14
15 int main(int argc, char **argv) {
16     if (argc < 3) {
17         fputs("simpleread <filename> <readsize (in bytes)>\n", stderr);
18         return 1;
19     }
20
21     char *filename = argv[1];
22     int readsize = atoi(argv[2]);
23
24     int fd = open(filename, 0);
25     if (fd < 0) {
26         fprintf(stderr, "Open file \"%s\"\n", filename);
27         perror("simpleread open file");
28         return 2;
29     }
30
31     /* char is always 1, so we can skip sizeof */
32     char *buf = malloc(readsize);
33     if (buf == NULL) {
34         perror("simpleread buf malloc");
35         return 3;
36     }
37
38 #ifdef HUGE
39     for (int i = 0; i < HUGE; i++) huge[i] = i % 255;

```

```

40 #endif
41
42     size_t bytes = 1;
43     while (bytes > 0) {
44         bytes = read(fd, buf, readsize);
45     }
46
47     if (errno != 0) {
48         printf("%d\n", errno);
49         perror("simpleread read");
50         return 4;
51     }
52     return 0;
53 }

```

Listing A.5: Simple Read Program

A.2 Linux Changes

```

1 diff --git a/mm/mempolicy.c b/mm/mempolicy.c
2 index d0d24abd1733..8d5df0246151 100644
3 --- a/mm/mempolicy.c
4 +++ b/mm/mempolicy.c
5 @@ -1154,6 +1154,7 @@ int do_migrate_pages(struct mm_struct *mm, const nodemask_t *from,
6     return busy;
7
8 }
9 +EXPORT_SYMBOL_GPL(do_migrate_pages);
10
11 /*
12  * Allocate a new page for page migration based on vma policy.

```

Listing A.6: Linux Kernel Patch to export do_migrate_pages, tested against version 5.4.65

A.3 ZFS Changes

```

1 /*
2  * Copyright (C) 1991-2002 Linus Torvalds
3  * Copyright (C) 2003,2004 Andi Kleen, SuSE Labs.
4  * Copyright (C) 2005 Christoph Lameter, Silicon Graphics, Inc.
5  * Copyright (C) 2021 Jacob Adams
6  *
7  * This file is part of the SPL, Solaris Porting Layer.
8  * For details, see <http://zfsonlinux.org/>.
9  *
10 * The SPL is free software; you can redistribute it and/or modify it
11 * under the terms of the GNU General Public License as published by the
12 * Free Software Foundation; either version 2 of the License, or (at your

```

```

13 * option) any later version.
14 *
15 * The SPL is distributed in the hope that it will be useful, but WITHOUT
16 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
17 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
18 * for more details.
19 *
20 * You should have received a copy of the GNU General Public License along
21 * with the SPL. If not, see <http://www.gnu.org/licenses/>.
22 *
23 * Solaris Porting Layer (SPL) Process Migration
24 */
25
26 #include <sys/thread.h>
27 #include <sys/migrate.h>
28
29 /*
30 * Ripped straight from kernel_migrates_pages in linux/mm/mempolicy.c
31 * Requires do_migrate_pages to be exported
32 * Skips a bunch of security checks, because those functions
33 * aren't exported
34 */
35 static inline int
36 spl_migrate_pages(struct task_struct *task, int node)
37 {
38     struct mm_struct *mm = NULL;
39     int err;
40     nodemask_t old = nodemask_of_node(curnode);
41     nodemask_t new = nodemask_of_node(node);
42
43     /* Find the mm_struct */
44     rcu_read_lock();
45     if (!task) {
46         rcu_read_unlock();
47         err = -ESRCH;
48         goto out;
49     }
50     get_task_struct(task);
51
52     err = -EINVAL;
53
54     rcu_read_unlock();
55
56     mm = get_task_mm(task);
57     put_task_struct(task);
58
59     if (!mm) {
60         err = -EINVAL;
61         goto out;
62     }
63
64     err = do_migrate_pages(mm, &old, &new, MPOL_MF_MOVE);

```

```

65
66     mmput(mm);
67
68 out:
69     return err;
70 }
71
72 void
73 spl_migrate(int node)
74 {
75     if (node >= nr_node_ids || node < 0) {
76         pr_warn("SPL: Can't migrate to node %d!\n", node);
77         return;
78     }
79     set_cpus_allowed_ptr(curthread, cpumask_of_node(node));
80     set_mems_allowed(nodemask_of_node(node));
81     spl_migrate_pages(curthread, node);
82     if (curnode != node) {
83         pr_err(KERN_ERR "SPL: Failed to migrate task %s!\n", curthread->comm);
84         dump_stack();
85     }
86 }
87 EXPORT_SYMBOL(spl_migrate);

```

Listing A.7: SPL Migrate, module/spl/spl-migrate.c, for ZFS 0.8.5

```

1 diff --git a/module/zfs/abd.c b/module/zfs/abd.c
2 index 8b2514404..8a3bee3c2 100644
3 --- a/module/zfs/abd.c
4 +++ b/module/zfs/abd.c
5 @@ -269,9 +269,9 @@ abd_alloc_pages(abd_t *abd, size_t size)
6     gfp_t gfp_comp = (gfp | __GFP_NORETRY | __GFP_COMP) & ~__GFP_RECLAIM;
7     int max_order = MIN(zfs_abd_scatter_max_order, MAX_ORDER - 1);
8     int nr_pages = abd_chunkcnt_for_bytes(size);
9 - int chunks = 0, zones = 0;
10 + int chunks = 0;
11     size_t remaining_size;
12 - int nid = NUMA_NO_NODE;
13 + int nid = curnode;
14     int alloc_pages = 0;
15
16     INIT_LIST_HEAD(&pages);
17 @@ -296,10 +296,6 @@ abd_alloc_pages(abd_t *abd, size_t size)
18
19     list_add_tail(&page->lru, &pages);
20
21 -     if ((nid != NUMA_NO_NODE) && (page_to_nid(page) != nid))
22 -         zones++;
23 -
24 -     nid = page_to_nid(page);
25     ABDSTAT_BUMP(abdstat_scatter_orders[order]);
26     chunks++;

```

```

27     alloc_pages += chunk_pages;
28 @@ -362,11 +358,6 @@ abd_alloc_pages(abd_t *abd, size_t size)
29     ABDSTAT_BUMP(abdstat_scatter_page_multi_chunk);
30     abd->abd_flags |= ABD_FLAG_MULTI_CHUNK;
31
32 -     if (zones) {
33 -         ABDSTAT_BUMP(abdstat_scatter_page_multi_zone);
34 -         abd->abd_flags |= ABD_FLAG_MULTI_ZONE;
35 -     }
36 -
37     ABD_SCATTER(abd).abd_sgl = table.sgl;
38     ABD_SCATTER(abd).abd_nents = table.nents;
39 }
40 diff --git a/module/zfs/arc.c b/module/zfs/arc.c
41 index ebbc89030..413f4c26b 100644
42 --- a/module/zfs/arc.c
43 +++ b/module/zfs/arc.c
44 @@ -292,6 +292,7 @@
45 #include <sys/zil.h>
46 #include <sys/fm/fs/zfs.h>
47 #ifdef _KERNEL
48 +#include <sys/migrate.h>
49 #include <sys/shrinker.h>
50 #include <sys/vmsystem.h>
51 #include <sys/zpl.h>
52 @@ -3406,6 +3407,9 @@ arc_hdr_alloc(uint64_t spa, int32_t psize, int32_t lsize,
53     hdr->b_l1hdr.b_arc_access = 0;
54     hdr->b_l1hdr.b_bufcnt = 0;
55     hdr->b_l1hdr.b_buf = NULL;
56 +#if defined(_KERNEL)
57 +     hdr->b_l1hdr.b_node = NUMA_NO_NODE;
58 +#endif
59
60     /*
61      * Allocate the hdr's buffer. This will contain either
62 @@ -6291,6 +6295,13 @@ top:
63
64     ASSERT(!embedded_bp || !BP_IS_HOLE(bp));
65
66 +#if defined(_KERNEL)
67 +     if (hdr->b_l1hdr.b_node == NUMA_NO_NODE) {
68 +         hdr->b_l1hdr.b_node = curnode;
69 +     } else if (curnode != hdr->b_l1hdr.b_node) {
70 +         spl_migrate(hdr->b_l1hdr.b_node);
71 +     }
72 +#endif
73
74     /* Get a buf with the desired data in it. */
75     rc = arc_buf_alloc_impl(hdr, spa, zb, private,
76         encrypted_read, compressed_read, noauth_read,

```

Listing A.8: ARC Patch for ZFS 0.8.5

```

1 diff --git a/include/spl/sys/migrate.h b/include/spl/sys/migrate.h
2 new file mode 100644
3 index 000000000..8d1301f17
4 --- /dev/null
5 +++ b/include/spl/sys/migrate.h
6 @@ -0,0 +1,32 @@
7 +/*
8 + * Copyright (C) 2021 Jacob Adams
9 + *
10 + * This file is part of the SPL, Solaris Porting Layer.
11 + * For details, see <http://zfsonlinux.org/>.
12 + *
13 + * The SPL is free software; you can redistribute it and/or modify it
14 + * under the terms of the GNU General Public License as published by the
15 + * Free Software Foundation; either version 2 of the License, or (at your
16 + * option) any later version.
17 + *
18 + * The SPL is distributed in the hope that it will be useful, but WITHOUT
19 + * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
20 + * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
21 + * for more details.
22 + *
23 + * You should have received a copy of the GNU General Public License along
24 + * with the SPL. If not, see <http://www.gnu.org/licenses/>.
25 + */
26 +
27 +#ifndef _SPL_MIGRATE_H
28 +#define _SPL_MIGRATE_H
29 +
30 +#include <linux/cpuset.h>
31 +#include <linux/mempolicy.h>
32 +#include <linux/sched.h>
33 +#include <linux/sched/mm.h>
34 +#include <linux/security.h>
35 +
36 +extern void spl_migrate(int);
37 +
38 +#endif /* _SPL_MIGRATE_H */
39 diff --git a/include/spl/sys/thread.h b/include/spl/sys/thread.h
40 index 3762717da..aa53f9657 100644
41 --- a/include/spl/sys/thread.h
42 +++ b/include/spl/sys/thread.h
43 @@ -54,6 +54,7 @@ typedef void (*thread_func_t)(void *);
44 #define thread_exit()          __thread_exit()
45 #define thread_join(t)         VERIFY(0)
46 #define curthread              current
47 +#define curnode                cpu_to_node(current->cpu)
48 #define getcomm()              current->comm
49 #define getpid()               current->pid
50
51 diff --git a/include/sys/arc_impl.h b/include/sys/arc_impl.h

```

```

52 index c8f551db7..9e24fe731 100644
53 --- a/include/sys/arc_impl.h
54 +++ b/include/sys/arc_impl.h
55 @@ -154,6 +154,8 @@ typedef struct llarc_buf_hdr {
56     kcondvar_t    b_cv;
57     uint8_t       b_byteswap;
58
59 + /* What node the ARC header/buffer was allocated on */
60 + int             b_node;
61
62     /* protected by arc state mutex */
63     arc_state_t   *b_state;
64 diff --git a/module/spl/Makefile.in b/module/spl/Makefile.in
65 index e16666aa9..b6c664921 100644
66 --- a/module/spl/Makefile.in
67 +++ b/module/spl/Makefile.in
68 @@ -16,6 +16,7 @@ $(MODULE)-objs += spl-kmem.o
69  $(MODULE)-objs += spl-kmem-cache.o
70  $(MODULE)-objs += spl-kobj.o
71  $(MODULE)-objs += spl-kstat.o
72 +$(MODULE)-objs += spl-migrate.o
73  $(MODULE)-objs += spl-proc.o
74  $(MODULE)-objs += spl-procfs-list.o
75  $(MODULE)-objs += spl-taskq.o

```

Listing A.9: ZFS Header and Makefile Changes, for ZFS 0.8.5

Appendix B

Low-Level Memory Allocation in the SPL

My initial research into ZFS led me to dive into the very low level details of how memory allocation works in ZFS, as it is actually rather different from how memory allocation works in other Linux kernel modules. My attempts to make these memory allocations more NUMA-aware had a substantially negative effect on the performance of ZFS. This is likely because I had to substantially constrain the scheduling of the memory allocation threads. I ended up taking my project in an entirely different direction after this, and thus these details of how this part of ZFS works are mostly irrelevant to my work, so I have confined it to this appendix.

B.0.1 kmem

The SPL contains a wrapper on kmem alloc and free, which attempts to emulate a more Solaris-like allocation process for small allocations, preventing them from failing, and instead retrying repeatedly until it succeeds [6, module/spl/spl-kmem.c]. It also includes debug functions to track memory allocations and find memory leaks. These functions call the per-NUMA node `kmalloc_node`, but with no NUMA node preference, and no such option exposed to the ZFS module to control its allocations. However, this function respects the calling thread's memory policy, which defaults to preferring allocations from the local NUMA node.

For larger allocations, ZFS uses `vmem_alloc` to allocate chunks of virtual memory, which uses

Linux's `__vmalloc` pretty much directly, and again uses a loop to prevent failure, which the code claims is unlikely with `__vmalloc` [6, module/spl/spl-kmem.c].

B.0.2 SLAB Allocator

The SPL also implements its own slab allocator for `kmem_cache`, instead of just providing a wrapper as it does for other kinds of memory allocation [6, module/spl/spl-kmem-cache.c]. This allows it to diverge significantly from the Linux implementation. Unlike the Linux implementation, which, according to the comments in `spl-kmem-cache` was inspired by Solaris' version, it supports destructors for cache objects (which have been removed from Linux), and bases its allocations on virtual addresses, removing the need for large chunks of contiguous memory when large allocations are requested (when objects smaller than 16KB are requested, it falls back to the Linux slab allocator). It also implements cache expiration, returning objects that have not been accessed in a few seconds back to the cache (Linux used to only do so in the case of a memory shortage, though it now has expiration after 2 seconds [14]). Finally, the SPL slab allocator implements a optimization known as a "cache magazine." These retain a per-cpu cache of recently-freed objects, which can be reused without locking.

Internally, the slab allocator supports using Linux's `__get_free_pages` with some specific flags, but does not use it except in deadlock situations, instead preferring `__vmalloc` to allocate objects in virtual memory. It also uses an SPL task queue for handling memory allocation and cache expiration.

While `kmalloc_node` can be used to enforce memory node binding, there is no direct way to influence `__get_free_pages` or `__vmalloc`. However, both functions respect the current thread's memory policy. `__vmalloc` technically has a similar function, `__vmalloc_node`, but this symbol is not exported to modules [4, mm/vmalloc.c]. It seems that ZFS is using this function in an unusual way, allocating memory from the `HIGHMEM` zone, while the expected use of the function, per the more standard (and exported) `vmalloc` and `vmalloc_node` functions, allocates from the `NORMAL` zone, and the comment above the function states "Any use of `gfp` flags outside of [the `NORMAL` zone] should be consulted with [memory management] people." [4, mm/vmalloc.c]. In

newer versions of Linux `__vmalloc_node` is exported but only with a specific testing configuration that regular kernels are unlikely to use [5].

Due to the nature of a slab allocator, which subdivides large memory allocations for smaller objects, relying on memory policy is not enough. Linux's slab allocator was made NUMA-aware by having per-node and per-cpu lists, though this leads to a large increase in memory usage by this

Bibliography

- [1] Matt Ahrens. *Lecture on OpenZFS read and write code paths*. Mar. 3, 2016. URL: <https://www.youtube.com/watch?v=ptY6-K78McY> (visited on 08/29/2020).
- [2] Matt Ahrens and George Wilson. *OpenZFS Basics*. May 14, 2018. URL: <https://www.youtube.com/watch?v=MsY-BafQgj4> (visited on 08/29/2020).
- [3] Jens Axboe et al. *Flexible I/O Tester*. Version 3.25. Dec. 4, 2020. URL: <https://github.com/axboe/fio>.
- [4] Linus Torvalds et al. *Linux*. Version 5.4.65. Sept. 12, 2020. URL: <https://www.kernel.org/>.
- [5] Linus Torvalds et al. *Linux*. Version 5.10.4. Dec. 30, 2020. URL: <https://www.kernel.org/>.
- [6] Matt Ahrens et al. *OpenZFS*. Version 0.8.5. Oct. 6, 2020. URL: <https://github.com/openzfs/zfs/tree/zfs-0.8.5>.
- [7] Azaghal. *Diagram of a binary hash tree*. Jan. 25, 2012. URL: https://commons.wikimedia.org/wiki/File:Hash_Tree.svg (visited on 04/21/2021).
- [8] Lars Bergstrom. "Measuring NUMA effects with the STREAM benchmark". In: *CoRR* abs/1103.3225 (2011). arXiv: 1103.3225. URL: <http://arxiv.org/abs/1103.3225>.
- [9] Arthur W. Burks, Herman Heine Goldstine, and John von Neumann. *Preliminary discussion of the logical design of an electronic computer instrument*. June 28, 1946. URL: <http://hdl.handle.net/2027.42/3972> (visited on 04/19/2021).
- [10] Pawe Dawidek. *A closer look at the ZFS file system*. BSDCan. May 16, 2008. URL: https://www.bsdcn.org/2008/schedule/attachments/58_BSDCan2008-ZFSInternals.pdf (visited on 12/20/2020).

- [11] Matthew Dobson et al. “Linux Support for NUMA Hardware”. In: *Proceedings of the Ottawa Linux Symposium* (July 26, 2003). URL: <https://www.kernel.org/doc/ols/2003/ols2003-pages-169-184.pdf>.
- [12] Garima Kochhar and Jacob Liberman. *Optimal BIOS Settings for High Performance Computing with PowerEdge 11G Servers*. Dell Product Group, July 13, 2009, p. 28. URL: <https://i.dell.com/sites/content/business/solutions/whitepapers/en/Documents/11g-optimal-bios-settings-poweredge.pdf>.
- [13] Christoph Lameter. “NUMA (Non-Uniform Memory Access): An Overview”. In: *Queue* 11.7 (July 1, 2013), p. 40. ISSN: 15427730. DOI: 10.1145/2508834.2513149. URL: <http://dl.acm.org/citation.cfm?doid=2508834.2513149> (visited on 06/10/2020).
- [14] Christoph Lameter. “Slab allocators in the Linux Kernel”. Oct. 3, 2014. URL: <https://events.static.linuxfound.org/sites/events/files/slides/slballocators.pdf>.
- [15] T. Li et al. “Characterization of Input/Output Bandwidth Performance Models in NUMA Architecture for Data Intensive Applications”. In: *2013 42nd International Conference on Parallel Processing*. 2013 42nd International Conference on Parallel Processing. ISSN: 2332-5690. Oct. 2013, pp. 369–378. DOI: 10.1109/ICPP.2013.46.
- [16] Kirk McKusick. *ZFS Internals Overview*. Oct. 21, 2015. URL: https://www.youtube.com/watch?v=IQp_FglfzUQ (visited on 09/01/2020).
- [17] Nimrod Megiddo and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache”. In: *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, Mar. 2003. URL: <https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>.
- [18] David A. Patterson, Garth Gibson, and Randy H. Katz. “A case for redundant arrays of inexpensive disks (RAID)”. In: *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*. SIGMOD ’88. New York, NY, USA: Association for Computing Machinery, June 1, 1988, pp. 109–116. ISBN: 978-0-89791-268-6. DOI: 10.1145/50202.50214. URL: <https://doi.org/10.1145/50202.50214> (visited on 04/18/2021).

- [19] Wonjun Song et al. “Evaluation of Performance Unfairness in NUMA System Architecture”.
In: *IEEE Computer Architecture Letters* 16.1 (Jan. 2017). Conference Name: IEEE Computer
Architecture Letters, pp. 26–29. ISSN: 1556-6064. DOI: 10.1109/LCA.2016.2602876.