Bladefs: Design and Implementation of a kernel level file system for scalable storage servers

Konstantinos Chasapis

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. *Angelos Bilas*

This work has been performed at the Foundation for Research and Technology Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Herakleion, GR-70013, Greece.

The work is supported by FORTH-ICS and the EU FP7 project IOLANES (contract number 248615).

University of Crete
Computer Science Department

Bladefs: Design and Implementation of a kernel level file system for scalable storage servers

Thesis submitted by
Konstantinos Chasapis
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: ________________________________
Konstantinos Chasapis

Committee approvals: ________________________________
Angelos Bilas, Professor, Thesis Supervisor
Computer Science Department
University of Crete

____
Kostas Magoutis, Research Scientist
Institute of Computer Science
Foundation for Research and Technology - Hellas

____
Dimitrios S. Nikolopoulos, Professor
School of Electronics, Electrical Engineering and Computer Science
Queen's University of Belfast

Departmental approval: ________________________________
Angelos Bilas, Professor
Professor, Director of Graduate Studies

Heraklion, May 2012

Abstract

The demand for storing data has been growing at a high rate. In addition, the shift towards data-centric applications that will process all stored information results in the need to improve the performance of the I/O path between application memory and physical devices in future servers. Although traditionally this path has been limited by device technology, today technologies such as solid-state disks (SSDs) can be used to increase throughput and reduce latency. However, with the advent of multicore CPUs and the increasing number of cores in servers, bottlenecks have shifted from devices to the host processor. The systems that runs on modern CPUs has not been designed for the levels of spatial parallelism that future servers will exhibit in terms of storage devices, cores, memory, and related interconnects. In addition, resource sharing between different workloads in multi-tenant setups results in increased interference as the amount of physical resources managed by the I/O path grows and applications are becoming more I/O intensive.

In this thesis we examine how partitioning of the I/O path can address both contention due to spatial parallelism as well as workload interference. We present bladefs, a kernel-level file system that supports partitioning of the I/O path. bladefs is a transparent, vfs-compliant file system that provides the minimum required functionality to handle file I/O and execute real applications and workloads. It relies on three underlying layers, a partitioned allocator, a partitioned cache and a partitioned journal for complementary functionality. We present the design of bladefs and the division of functionality across layers to build a partitioned I/O path. Our main contribution in the design of bladefs is that by enabling partitioning of the I/O path we have both contention reduce and isolation between workloads. Eliminating the contention using partitions allows us performance scaling by increasing their number.

We evaluate our approach using real-life workloads including OLAP, OLTP along with micro benchmarks. Our results show that our approach to partitioning the I/O path can isolate workloads for interfering for host-level resources (cores, storage devices and memory) in the I/O path, resulting in eliminating any performance variations in multi-tenant workloads. In addition, our approach is able to scale when increasing the number of partitions.

Περίληψη

Η συνεχώς αυξανόμενη απαίτηση για αποθήκευση δεδομένων αντανακλάται και στις data-centric εφαρμογές που επεξεργάζονται όλα τα αποθηκευμένα δεδομένα. Αυτό έχει ως συνέπεια την ανάγκη να βελτιώσουμε την απόδοση του μονοπατιού εισόδου/εξόδου μεταξύ της μνήμης της εφαρμογής και των φυσικών συσκευών αποθήκευσης σε μελλοντικούς εξυπηρετητές. Αν και παλαιότερα το μονοπάτι αυτό ήταν περιορισμένο από την τεχνολογία των συσκευών, σημερινές τεχνολογίες όπως οι solid state drives αύξησαν την ρυθμαπόδοση και μείωσαν την καθυστέρηση. Ωστόσο, με την έλευση των πολυπύρηνων επεξεργαστών και τον αυξανόμενο αριθμό πυρήνων στους εξυπηρετητές η συμφόρηση μεταφέρθηκε απο τις συσκευές στον επεξεργαστή. Το λογισμικό συστήματος που εκτελείται στους σύγχρονους επεξεργαστές δεν είναι σχεδιασμένο για να εξάγει χωρικό παραλληλισμό που θα έχουν οι μελλοντικοί εξυπηρετητές από την άποψη συσκευών αποθήκευσης, πυρήνων, μνήμης και σχετικών διασυνδεδεμένων δικτύων. Επιπρόσθετα ο διαμοιρασμός πόρων μεταξύ διαφορετικών εργασιών σε διαμοιραζόμενες πλατφόρμες έχει ως αποτέλεσμα την αύξηση της παρεμβολής καθώς αυξάνεται το πλήθος των φυσικών πόρων που διαχειρίζεται το μονοπάτι Ε/Ε και οι εφαρμογές τείνουν να εξαρτώνται περισσότερο από την λειτουργία Ε/Ε.

Σε αυτή την εργασία εξετάζουμε πως ο διαχωρισμός του μονοπατιού Ε/Ε μπορεί να συμβάλει στη μείωση της συμφόρησης που προκαλείται από τον χωρικό παραλληλισμό καθώς επίσης και από την αλληλεπιδραση των εργασιών. Παρουσιάζουμε το Bladefs, ενα σύστημα αρχείων υλοποιήμενο σε επίπεδο πυρήνα το οποίο υποστηρίζει το διαχωρισμό του μονοπατιού Ε/Ε. Το Bladefs είναι ένα διαφανές, συμβατό με το εικονικό σύστημα αρχείων (VFS) και παρέχει την ελάχιστη απαιτούμενη λειτουργικότητα για να διαχειρίζεται την λειτουργία Ε/Ε σε αρχεία και είναι ικανό να εκτελέσει πραγματικές εφαρμογές και εργασίες. Στηρίζεται πάνω σε τρία επίπεδα, ένα διαχωριζόμενο εκχωρητή μνήμης, μια διαχωριζόμενη κρυφή μνήμη και ένα διαχωρισμένο ημερολόγιο (journal) για επιπρόσθετες λειτουργίες. Παρουσιάζουμε τη σχεδίαση του Bladefs, όπως επίσης και το διαχωρισμό της λειτουργικότητας μεταξύ των επιπέδων που απαιτούνται για την δημιουργία του διαχωρισμού στο μονοπάτι Ε/Ε.

Η καινοτομία μας με το Bladefs είναι το ότι δίνοντας τη δυνατότητα να διαχωρίζουμε το μονοπάτι Ε/Ε επιτυγχάνουμε μείωση στη συμφόρηση αλλά και στην απομόνωση μεταξύ διαφορετικών εργασιών. Επίσης, κάνοντας χρήση του διαχωρισμού μας επιτρέπει την κλιμακωτή βελτίωση της απόδοσης του συστήματος καθώς αυξάνεται ο αριθμός των διαχωρισμών. Τέλος αξιολογούμε το Bladefs χρησιμοποιώντας πραγματικές εφαρμογές όπως OLAP, OLTP, αλλά και μικροεφαρμογές. Τα αποτελέσματά μας δείχνουν ότι χρησιμοποιώντας την τεχνική του διαμοιρασμού μπορεί να απομονώθει η παρεμβολή μεταξύ διαφορετικών εργασιών σε πόρους (μνήμη, συσκευές αποθήκευσης και πυρήνες) που παρέχονται σε επίπεδο κόμβου (host) ανάμεσα στο μονοπάτι Ε/Ε καταλήγοντας στην εξάλειψη της κυμαινόμενης απόδοσης σε διαμοιραζόμενες πλατφόρμες. Γενικά μπορούμε να βελτιώσουμε την απόδοση του συστήματος αυξάνοντας τον αριθμό των διαχωρισμών.

Acknowledgements

First of all I would like to express my acknowledgments to my supervisor and my academic advisor Prof. Angelos Bilas and also to the two member of my masters committee Prof. Dimitrios S. Nikolopoulos and Dr. Kostas Magoutis. AIso would like to thank all the members of ICS-FORTH CARV laboratory and especially to Dr. Manolis Marazakis and Markos Fountoulakis.

Many thanks to Stelios Mavridis for providing a stable version of the DRAM I/O cache. Also, I would like to thank Dhiraj Gulati for the implementation of the allocator module and to Yiannis Sfakianakis for the implementation of journal module. I would like also to thank CARV members Michalis Lygerakis, George Kalokairinos, Vassilis D. Papaefstathiou, Spyros Lyberis and also my colleagues Yiannis Klonatos and Maria Chalkiadaki.

Special thanks to Georgia Kydonaki. Moreover my close friends in alphabetical order Stamatis Zampetakis, Dimitris Milioris, Giannis Nikolopoulos, Marios Prasinos, Giorgos Tzenakis, Nikos Fantaoutsakis, Giorgos Fragakis that made my living in Herakleion more fun that I could imagine.

Last but not least I would like to thank my family that always supports me to fulfill my dreams.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Background . . . 4
1.2.1 File system functionality . . . 4
1.2.2 Current I/O path . . . 7
1.3 External components used by Bladefs . . . 8
1.3.1 Allocator . . . 9
1.3.2 Partitioned NUMA-Aware DRAM Cache . . . 10
1.3.3 Journal . . . 11
1.4 Related Work . . . 12
1.4.1 Shared resources in multicores and multi-processors . . . 13
1.4.2 State of the art file systems . . . 13

2 Design and Implementation **17**
2.1 Bladefs namespace management . . . 17
2.2 File structure and implementation . . . 20
2.3 Directory design . . . 23
2.4 Data and Metadata placement . . . 26
2.5 Bladefs reliability . . . 26
2.6 Partitioned allocator . . . 27
2.6.1 Allocator operations and disk layout . . . 27

3 Evaluation **29**
3.1 Benchmarks . . . 29
3.1.1 Make many files: . . . 29
3.1.2 kfsmark: . . . 29
3.1.3 TPC-H: . . . 30
3.1.4 TPC-W: . . . 30
3.2 Evaluation Platform . . . 30
3.3 Single partition Bladefs performance evaluation . . . 31
3.3.1 Make many files . . . 31
3.3.2 TPC-H . . . 31
3.3.3 TPC-W . . . 32

3.4 Isolation between Competing Workloads 34
3.5 Performance scaling with the number of partitions 36

4 Conclusions and Future Work 39

List of Figures

1.1 Future I/O server. 4
1.2 File system tree example. 5
1.3 I/O path overview. 8
1.4 Disk layout of allocators container. 9
1.5 In memory layout of free list. 9
1.6 Partitioned block-level DRAM cache. 10
1.7 Journal module operations. 12

2.1 Bladefs: one and two partitions. 18
2.2 Bladefs disk i-node layout. 19
2.3 Bladefs files disk layout. 20
2.4 Bladefs symbolic link layout. 22
2.5 Bladefs hard links design. 22
2.6 Bladefs readahead. 24
2.7 Bladefs directories structure. 24
2.8 Placement of data and metadata in partitions. 27
2.9 Allocator disk layout. 27

3.1 TPC-H queries Q[1-4] XFS and Bladefs with BIOS and ramdisk. . 32
3.2 TPC-H queries Q[1-4] XFS and Bladefs with BIOS on top of loop device. 33
3.3 TPC-H queries Q[1-4] XFS and Bladefs with cache and journal. . . 33
3.4 TPC-W throughput for XFS VS. Bladefs. 34
3.5 TPC-W average transaction time for XFS VS. Bladefs. 34
3.6 Bladefs with two partitions, one for the TPC-W, and one for the Write thread. 35
3.7 kfsmark READ / WRITE 8 SSDs request size 4KB. 36
3.8 kfsmark READ / WRITE 8 SSDs request size 64KB. 37

List of Tables

3.1 Make many files creates per second. 31
3.2 TPCW average transaction time in msec. 35
3.3 TPCW throughput transaction per second. 35

Chapter 1

Introduction

In this thesis we present Bladefs, a kernel-level file system that supports partitioning of the I/O path. Partitioning aims to improve I/O performance and scalability in multicore environments, by reducing contention between shared resources in the I/O path. Partitioning also enables performance isolation, i.e. minimizing the interference between competing workloads. In contrast to other file systems, functionality is decomposed in several layers: (1) a partition-aware allocator, (2) a partitioned DRAM cache, and (3) a journal module. Bladefs targets two important workload classes: (1) database workloads, with few but very large files, and (2) virtualization run-time environment, with large VM image files at the host.

1.1 Motivation

The amount of data stored and processed in data centres is increasing rapidly [16]. Moreover, many aspects of file system design have been thoroughly examined over the last decades. Apart from general-purpose file systems, special-purpose file systems exist for diverse types of applications and workloads. Over the past decade, systems performance has improved using multi-core architectures and trends point to an increasing number of cores. Systems with more than one CPU are being used to scale applications performance, including I/O [25]. Solid state drives (SSD's) have decreased the gap between memory and storage throughput and latency. However, performance bottlenecks on multicore architectures [33] are starting to surface in the current host-level I/O path.

Another significant aspect is the interaction between the guest operating system (guest OS) and the host OS, as well as the implications of nested file systems in virtualized environments. Virtualization provides much more flexibility to increase computing resource utilisation. Quantifying isolation among applications in different virtualization environment has been studied by [12] [21] [13] [22] [19] [26] [15] .

The above trends indicate that future storage servers will consist of a system with multicore CPUs, multiples storage controllers and many DRAM chips as

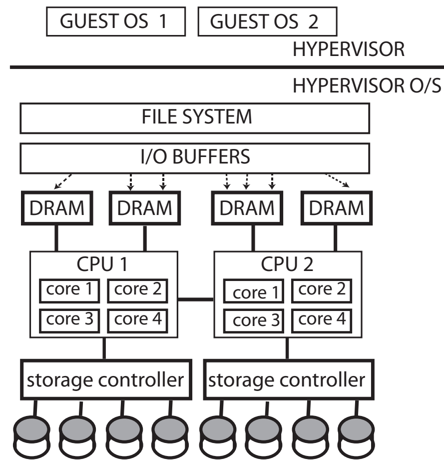


Figure 1.1: Future I/O server.

shown in Figure 1.1. The effects of performance in multi-tenant environments have been examined in [31] [4] [7] [18] and the impact of the memory subsystem and resource sharing on datacenter applications in [28].

Bladefs is designed to meet the requirements of these systems. We implement Bladefs as a kernel-level module in the Linux kernel and we evaluate it with a modified I/O stack using various workloads.

The rest of this report is organised as follows: First, we give background information related to file systems, the I/O path, and the modules that Bladefs uses. Then we discuss the design and implementation of Bladefs and the interaction with the other layers. Finally, we present the evaluation of Bladefs using various workloads and draw our conclusions.

1.2 Background

In this section we analyze the functionality that a Linux file system exports and describe common implementation methods. We describe the I/O path from user space applications down to storage devices. Finally, we present the modules that Bladefs uses for partitioning the I/O path.

1.2.1 File system functionality

A file system has to support the following functionality: namespace management, reliability, space management (block allocation), and caching (in DRAM).

Namespace management

Filenames provide a mapping between storage locations and files. Linux file systems implemented this mapping using *i-nodes* that are the low-level representation of a

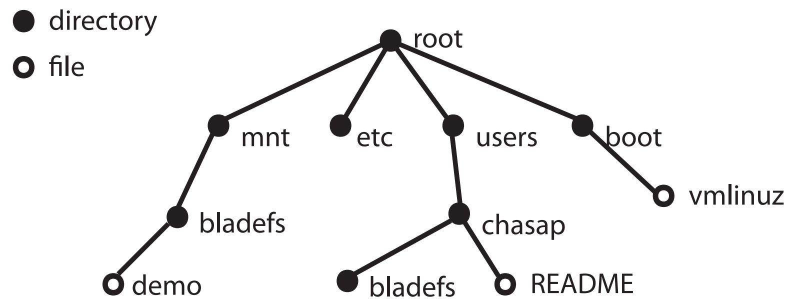


Figure 1.2: File system tree example.

file.

Directories allow users to group files. This abstraction is implemented by connecting the filename (or i-node) with a directory. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. This results to a hierarchical tree structure of file system as shown in Figure 1.2. To implement this in the Linux kernel, an object named *dentry* is used. A dentry contains the binding between a directory entry name and the directory that this entry belongs to.

Moreover, it is possible to have links to files so that the same file can be accessed from different paths (filenames). Links are either *symbolic* or *hard*. Symbolic links contain the path of the file that they point to, and can span file systems and devices. On the other hand, hard links have a direct connection to the i-node that they point to, and cannot span file systems.

Block devices, network interfaces, pipes etc. are represented as special-purpose files. Bladefs does not currently support such device files.

A significant amount of book-keeping information is typically associated with each file within a file system. Examples include the length of the data contained in a file, the number of blocks allocated for the file, the time when the file was created, last modified, last accessed, information about the type of file (e.g. block, character, socket, subdirectory, regular file, etc.). File systems control access to data by restricting access to file and directories via the use of owner, user, and group IDs. This information can be stored in the i-node or the dentry of the file; the details are left to the specific file system implementation and may affect the performance of a various operations.

Reliability

The file system maintains in-memory data structures, as well as persistent data structures on the storage media. The filesystem has to preserve the integrity of these data structures in the event of system failures, as well as in the event of abnormal application termination. From the point of view of the filesystem, critical operations are the updates to filesystem metadata (e.g. upon creation of new

directories and files), the updates of directory entries, and the handling of data that has been written by applications but has not yet been moved to the physical storage media. In the event of a failure, the filesystem may have to restore the integrity of out-of-date, or even corrupted data structures.

To provide reliability, file systems keep track of the metadata updates in a log that is stored in a reserved area of the underlying device (journal). Metadata updates are first appended to the journal and later are applied in-place on the storage media that hosts the filesystem. After a failure the file system must check the journal for pending transactions and apply them to the file system.

The journal can also be used for data blocks. However, this has a significant performance impact. Therefore, it is common practice to use the journal only for staging the updates of filesystem metadata and leave the consistency of file data to user applications (e.g. database servers). The filesystem exports operations to support user level applications in this task. A specific example is the `fsync()` system call that guarantees synchronisation between file data that are cached in-memory and the underlyng device.

An important property of the journal is that journal writes have to execute *synchronously*, i.e. writes to the journal must be successfully completed before issuing the associated write operation for the data blocks. Using low-latency storage devices (e.g. SSDs) for the journal is a common good practice. Another commonly applied technique is *delayed logging* where journal writes are grouped in-memory before being issued to the journal device as a batch.

For the majority of workloads, contention for completing writes to the journal is not a critical performance concern. However, there are metadata-intensive workloads where frequent directory and file management operations put pressure on the journal, e.g. in the case of general-purpose file servers.

Space management(block allocation)

File systems allocate space in a granular manner, usually at the granularity of multiple consecutive physical units on the device. The file system is responsible for the layout of files and directories, keeping track of which areas of the media belong to which file/directory and which are not currently in use. Filesystem allocators need to address several issues: (1) fragmentation that occurs when unused space or individual files are not contiguous, (2) scalability with concurrent requests, (3) scalability with increasing device sizes, (4) integrity in the presence of failures, and (5) data/metadata placement.

Various techniques are used to eliminate fragmentation in the allocator. Multiple block allocation is suitable for large appends to a file, and minimizes device fragmentation. Persistent preallocation where data blocks for a file are allocated up-front, ensures contiguous allocation as much as possible for a file, irrespective of when and in which order data actually gets written, and guarantees space availability for writes within the preallocated size. It is useful when an application has prior knowledge of the expected file sizes. Database workloads may benefit a lot

by using this technique, since they have large files that are gradually growing.

Another commonly used method is *delayed allocation* or *allocate-on-flush*. Disk space for the appended data is subtracted from the free-space counter, but not actually allocated in the bitmap or free list. Instead, the appended data is held in memory until it must be flushed to storage due to memory pressure when the kernel decides to flush dirty buffers, or when the application performs the Unix "sync" system call. This has the effect of batching together allocations into larger runs. Such delayed processing reduces CPU usage and tends also to reduce disk fragmentation, especially for files that grow gradually. It can also help at keeping allocations contiguous when there are several files growing at the same time [1].

Multiple allocation groups can be used to increase scalability with concurrent requests to the allocator. To scale with the size of the device, *extents* represent large, contiguously allocated space. For reliability of the allocator, the journal mechanism is used for keeping track of persistent metadata changes related to allocation.

Finally, the filesystem allocator needs to define specific policies to decide where a particular file will reside. Layout and placement policies have an impact on access performance, as they affect the number and average distance of disk seeks. Spreading directories on the disk allows files in the same directory to remain more or less contiguous as their number and/or size grows. However, there are cases where this causes excessive spreading of the data on the disk's surface, penalizing performance. The Orlov block allocator [2] tries to address this issue.

DRAM Caching

Today storage device latency is orders of magnitudes higher than DRAM memories. To increase system performance DRAM is used for temporarily caching data to improve file system performance. Bypassing the DRAM cache is usually possible using the O_DIRECT flag when opening a device, forces the traffic at any read/write to occur directly to the storage device.

1.2.2 Current I/O path

I/O operations are initiated in user space upon applications requests. Figure 1.3 illustrates the whole I/O path in the linux kernel. In most cases applications do not use directly system calls but I/O operations provided by applications libraries e.g. the C standard library. Using application libraries may result in different I/O patterns than the one application initiates. Moreover extra buffering might occur in these libraries and applications may use additionally operations, such as `fflush()`, to empty these buffers.

In kernel space there is a software layer that provides an abstraction between system call semantics and file system implementation called Virtual File System

[1] This feature closely resembles an older technique that Berkeley's UFS called "block reallocation".

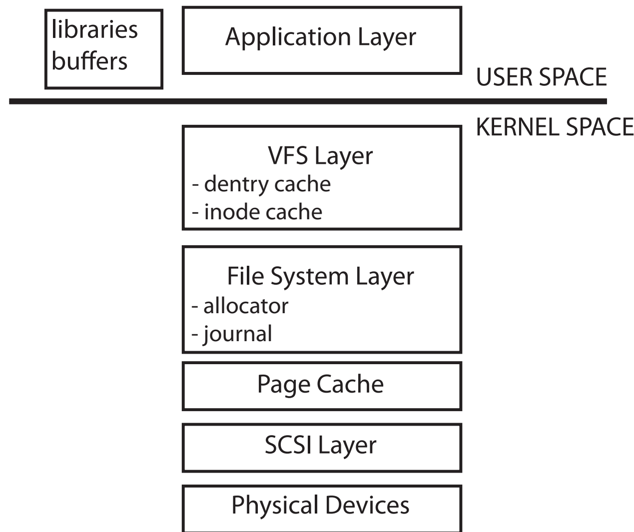


Figure 1.3: I/O path overview.

switch (VFS). To decouple file system implementation and the abstraction that VFS provides, VFS requires that file systems implemented below VFS use and fill up VFS metadata. Most commonly, VFS metadata are the superblock, i-nodes, and dentries. Superblocks of all mounted file system are kept in a linked list. Recently used metadata are kept in the special purpose caches within the VFS layer to seed up performance in metadata operations. Contention is commonly noticed at these caches that are shared among mounted file system. A lot of work was recently done in Linux community to overcome the scalability issues encountered in VFS.

Most file systems implement both block allocation and reliability (journal) mechanisms by themselves. DRAM caching for the file system data and metadata is provided using the kernel buffer cache, a generic data cache. The buffer cache is used to temporary store in DRAM a significant percentage of application data and file system metadata. Current implementations of the buffer cache do not support NUMA awareness and policy differentiation between data and metadata. Moreover, you can not control the amount of buffer cache that each application will use. Below the buffer cache is the SCSI layer that hides the complexity of physical format and export intelligent, buffered, peer to peer interface to transfer data between physical device and the upper layer.

1.3 External components used by Bladefs

Bladefs functions as a demultiplexer and mainly implements partitioned namespace management. It relies on a partitioned allocator, a partitioned DRAM cache, and a Journal to provide the rest of the functionality. In the next section we discuss the design and features of these components.

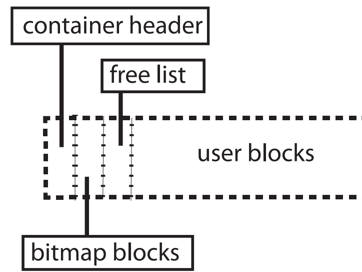


Figure 1.4: Disk layout of allocators container.

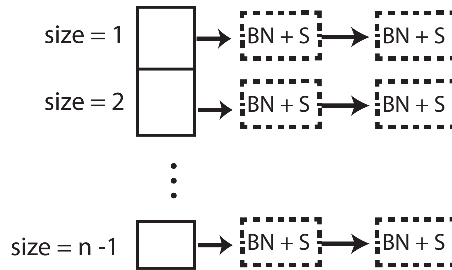


Figure 1.5: In memory layout of free list.

1.3.1 Allocator

The basic building block for the allocator is a container. Containers are physically contiguous chunks on a disk device. Containers cannot be modified after creation. A container resides only on one device and cannot span device partitions or devices. As shown in Figure 1.4 the disk layout of a container consists of four sections: i) container header ii) bitmap blocks iii) free list form container iv) and the actual data blocks.

The container header consists of container size, block size, free space and a pointer to the next container in order to link all containers. The block size is the minimum allocation unit of the container. Bitmaps are physically stored just after the container's header. An in-memory copy of the bitmap is later used for batching small updates and serving reads without I/O accesses. Regarding a container with 4KBytes blocks size, one bit represents 4 KBytes of physical storage.

Eventually, containers will suffer from fragmentation and locating the desired amount of consecutive physical space will imply parsing through the entire bitmap. To overcome this limitation we maintain an in memory list of blocks that have recently been freed. This free-list of blocks is indexed according to the size of free blocks. Such free lists do not exist in freshly created containers; they are created after a period of continuous allocations-deallocations to speed up the performance of alloc and free operations instead.

Container implement two basic operations:

- *getcblock(addr, count)*: allocate consecutive blocks. To make a new request

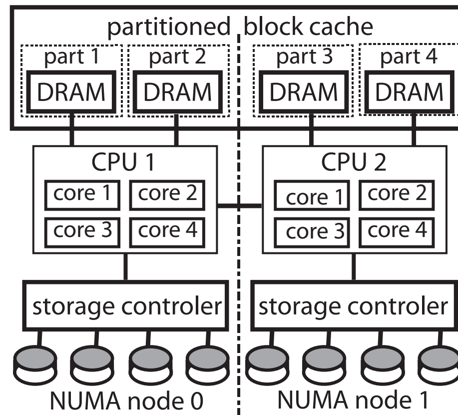


Figure 1.6: Partitioned block-level DRAM cache.

for blocks in a given container, along side of handle to the device. The allocations are made first by checking the in-memory free lists. If allocation request can not be served by the in memory free list then the container bitmap searches for free space. A container support up to 64 consecutive blocks.

- *freecblock(addr, count)* frees consecutive blocks allocated using getcblocks.

Every time a block is allocated, the in-memory copy of the bitmap is marked allocated. Updated entries in bitmaps are later on taken into account when writing FS metadata to the journal.

1.3.2 Partitioned NUMA-Aware DRAM Cache

Bladefs use the partitioned NUMA-aware block-level DRAM cache illustrated in Figure 1.6. Partitioning is being used to isolate different areas of the total address space. This reduces contention and allows scaling by using independent partition caches, while offering a single system image on top of the I/O subsystem.

Caches supports the creation of multiple partitions per cache with variable size elements mapped to disk ranges. Elements are packed transparently on the disk and unpacked by the cache when the user needs them. Accessing data through a cache is done by using the element's index and tag, and the cache manages all I/O in case of a miss.

Performance scaling while limiting contention is possible by using multiple independent partitions of caches, each of which stores and handles a disjointed set of data blocks. Blocks from a single workload or file can span caches, since there is no restriction on the mapping of blocks to caches.

This cache offers NUMA awareness in the sense that it can pin partitions to specific NUMA nodes. Each cache partition has two threads: i) a thread that handles requests I/O requests, and ii) an evictor thread that evicts elements from the cache. Both threads that are used per partition are pined to CPUs belonging

to a certain NUMA node. Memory buffer allocation per cache is done on the same NUMA node.

Cache uses an LRU approximation with time stamps instead of maintaining an exact LRU list and the associated locks. This approach trades accuracy for using only atomic operations on reference counts and timestamps rather than locks. The cache structure is implemented with a hash table of bucket lists, each of which is protected by fine grain locks. The cache packs elements in 4 KByte extents and maintains metadata per extent. The interaction of the cache with the actual I/O device occurs on the base of extents, regardless of the size of cached objects (elements).

The cache uses mostly a write back policy with a time-stamp based LRU approximation. Write-back is always used for data blocks. However, metadata blocks need to be transferred directly to the journal device to ensure correct ordering during recovery. For this reason, metadata blocks (or any block that needs to be journaled), enforce a write-through mode.

This cache is exported as a block device. The Linux block API (BIOS) requires a copy of the pages used to perform the I/O. To eliminate this, besides the standard block level API we also provided a custom *get/put* API that avoids copying. Essentially, the get/put API allows the caller to directly operate on buffers in the cache using pointers.

1.3.3 Journal

The journal used by Bladefs exports a reliable block device over a device given as input. In order to achieve recovery the journal defines the notion of atomicity-internals similar to transactions. Atomicity internals are used only for recovery purposes (not isolation).

The journal module supports three modes of operations: i) Both data and metadata are journaled, ii) only metadata are journaled, iii) pass throughout mode, where journal does nothing. Internally, the journal buffers I/O blocks to an in memory ring before sending them to the core device. A lookup structure that stores information about the location of data in the module is used to identify if the requested blocks exist in the memory buffer. Both read and write operations at the journal are explained below and illustrated in Figure 1.7.

The read operation is fairly simple. The lookup structure is used to find the location of each block separately. For the blocks that are present in the circular buffer there is no need for I/O to the core device so a copy is returned. However, for blocks that are not found in the circular buffer, the journal module issues a read operations to either the log or the storage device.

Write operation in the journal consists of three steps:

- Write to the in memory ring buffer: a metadata header for the operation is created. This header plus each page from the I/O are copied to the circular

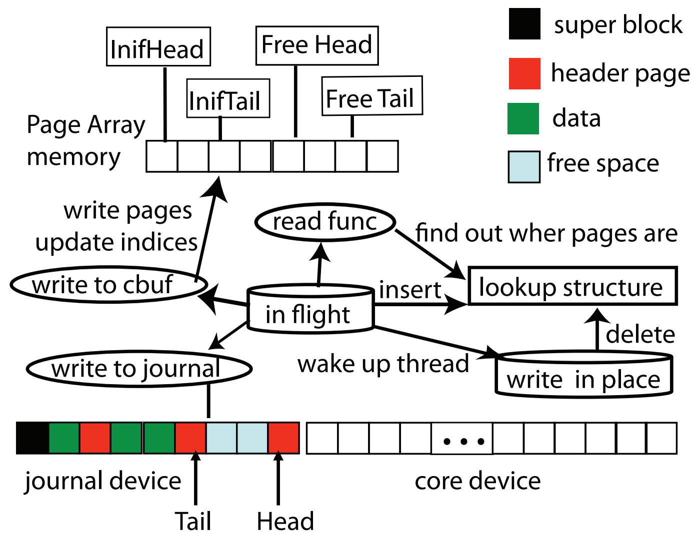


Figure 1.7: Journal module operations.

buffer in order. At this point the lookup structure is updated and the issuing thread can proceed. In case of a full buffer the copy is stalled.

- Writing to journal log: when a portion of the circular buffer is filled with data, they are transferred to the journal device. Using this method results that writing to journal device are large and sequential.
- Write in-place to the storage device: a separate thread handles in place writes. It finds all transactions that are completed, forwards them in-place to the core device and updates the lookup entry for these pages.

Recovery can only be ensured if the superblock of the journal is not corrupted. The superblock contains the head and the tail of the journal. The superblock is used to identify the portion of the log tat contains valid entries during recovery. After all complete transactions are moved in place, the log is cleaned and the journal starts accepting regular I/O requests.

Other modules can use the API of the journal to specify the begging and end of transactions. Transaction can not abort during regular operation and they may be discarded during recovery after a failure.

1.4 Related Work

Scalable performance with multicore processors is an emerging necessity for modern storage systems as they are relying more and more in them . There are several file systems that aim to have scalable performance under multicore architectures and I/O bound applications including XFS [3] , ext4 [5], and Btrfs [8] that represent the state of the art. In this section we briefly present key design concepts and features

of these file systems. Moreover, we present related work that aimed to provide performance isolation and resource management at shared server environments.

1.4.1 Shared resources in multicores and multi-processors

Gupta et al. presents the design and evaluation of a set of primitives implemented in Xen [6] to support performance isolation across virtual machines [15]. Guo el. al present a framework for native multi-tenancy application development and management in[14]. Whitaker et. al presents the Denali isolation kernel, an operating system architecture that safely multiplexes a large number of untrusted Internet services on shared hardware in [32]. Karlsson et. al propuse Triage, a software-only solution that ensures performance isolation and differentiation for storage systems in [17].

1.4.2 State of the art file systems

There are several file systems that aim to have scalable performance under multicore architectures and I/O bound applications including XFS [3] , ext4 [5], and Btrfs [8] that represent the state of the art. In this section we briefly present key design concepts and features of these file systems.

XFS

XFS [10] is a high-performance 64-bit journaling file system created by Silicon Graphics, Inc. for IRIX OS. After initial implementation, developers of XFS completed also the porting to Linux kernel. XFS is particularly efficient in parallel I/O and has extreme scalability in concurrency of I/O threads, file system performance, and files' size.

XFS uses B+ trees to keep metadata for files and directories. Files use extents to represent the file layout. XFS handles directories as special files that keep *dentries*. XFS *dentries* hold the file name and the *inode* number of the *inode* that the file is associated with. Each directory can have unlimited number of *dentries*. Files use 64-bit address space and the maximum file size is up to 8 exabytes. Furthermore, XFS supports also sparse files.

XFS file systems are internally partitioned into allocation groups, which are equally sized linear regions within the file system. Files and directories can span to multiple allocation groups. Each allocation group manages its own inodes and free space separately, providing scalability and concurrency in the presence of multiple threads and processes. This architecture helps to optimize parallel I/O performance on multiprocessor or multicore systems, as metadata updates are also parallelizable. The internal partitioning provided by the allocation groups can be beneficial when the file system spans to multiple physical devices. Partitioning enables higher utilization and extraction of higher throughput from the underlying storage components. XFS allocator support variable length extents that are used to describe data blocks within a file. Apart from delayed allocation and multiple

block allocation, striped allocation is also supported for XFS file system. Striped allocation maximized throughput on striped RAID arrays. Letting XFS be aware of the stripe unit ensures that data allocations, inode allocations and the internal log (journal) are aligned with the stripe unit.

After a system crash XFS performs recovery automatically at the file system mount time. The recovery speed is independent of the size of the file system. It its early stages, XFS was slower than the rest of the traditional file systems. However, XFS recently added a "delayed logging" technique in it's journaling mechanism and gained significant performance in metadata operations [9]. Furthermore, XFS supports journal placement on a separate physical device, with its own I/O path, that can also increase performance.

XFS supports online defragmentation, online growing, variable block size[2] and extended file attributes[3]. Although backup and restore utilities exist, there is no snapshot mechanism.

Another unique feature of the XFS file system is that it provides an API that guarantees minimum guaranteed I/O rate for the applications. To implement this XFS dynamically calculates the performance available from the underlying storage devices, and reserve bandwidth sufficient to meet the requested performance for a specified time. However this feature is not supported in Linux version of XFS.

XFS relies on the underling devices to support data encryption, de-duplication and data redundancy.

Ext4

Ext4 is a journaling file system for Linux, developed as the successor to ext3. Ext4 focuses on addressing scalability, performance, and reliability issues of ext3. Ext4 is backward compatible with ext3 and ext2, making it possible to mount ext3 and ext2 filesystems.

Ext4 file design is different compered to ext3. Ext3 implements one-to-one mapping from logical blocks to disk blocks using direct and indirect pointers. Although ext3 scheme is better for small and sparse files it does not scale for large files. Ext4 introduces *extents* to support large files efficiently. An *extent* is a single descriptor that represents a range of contiguous physical blocks. Ext4 includes *extents* in a tree base mapping and a single *extent* can represent up to 215 contiguous blocks, or 128 MB with 4 KB block size. Maximum file size in ext4 is 16TB with 4KB block size. This approach efficiently maps logical to physical blocks for large contiguous files.

Ext4 allocates inode tables statically as ext3, so the maximum number of files is defined at file system creation time. Reserving a large amount of disk blocks for

[2] File system block size represents the minimum allocation unit.

[3] Enables users to associate files with metadata not interpreted by the file system such as storing the author of a document, character encoding, checksums, cryptographic hash or digital signatures

inode tables leads to waste of storage space. This is more important for ext4 as the inode size has been increased to 256Bytes in contrast to ext3.

Ext4 improves directory scalability. In contrast to ext3 that supports 32000 subdirectories, ext4 can have an unlimited number of subdirectories. Also ext4 changes the data structure that ext3 uses to store dentries. Instead of a linked list that ext3 use, ext4 uses an HTree[4] data structure to store dentires inside a directory. This change in directories achieves a tremendous performance improvement of up to a factor of 50 to 100 for large directories with more than 10.000 entries.

Ext4 has a powerful block allocation scheme that can efficiently handle large block I/O and reduce file system fragmentation of small files under multi-threaded workloads. Ext4 uses bitmaps with extents to describe the allocated space for a device. Techniques such as persistent pre-allocation, delayed allocation and multiple block allocations are also implemented in ext4.

Ext4 implements journaling in a separate block device module. Ext4 uses checksums in the journal to improve reliability. This feature has the side benefit that it can safely avoid a disk I/O wait during journaling, improving performance. Journal checksumming was inspired by [23] with modifications within the implementation of compound transactions.

Ext4 supports online defragmentation, online resizing, variable block size, and extended file attributes. Moreover, backup and restore utilities exist but there is no snapshot mechanism.

Btrfs

Btrfs is intended to address the lack of pooling, snapshots, checksums and integral multi-device spanning in Linux file systems. The core data structure of Btrfs the copy-on-write B-tree was originally proposed in[24]. Rodeh et al. suggest the addition of reference counts and certain relaxations to the balancing algorithms of standard B-trees that would make them suitable for a high-performance object store with copy-on-write snapshots, yet maintain high concurrency.

File data are kept outside the tree in extents, which are contiguous runs of disk blocks. Extents do not have headers and contain only (possibly compressed) file data. Each extent is tracked in-tree by an extent data item. When a small part of a large extent is overwritten, then the resulting copy-on-write may create three new extents: a small one containing the overwritten data, and two large ones with unmodified data on either side of the overwrite. To avoid having to re-write unmodified data, the copy-on-write may instead create bookend extents, or extents which are simply slices of existing extents.

An extent allocation tree is used to track space usage by extents, which are zoned into block groups. Block groups are variable-sized allocation regions which alternate successively between preferring metadata extents (tree nodes) and data extents (file contents). The default ratio of data to metadata block groups is 1:2.

[4] HTree data structure is a specialized BTree-like structure using hashes.

Inode items include a reference to their current block group. The Btrfs allocator, similar to the Orlov block allocator [2] and block groups in ext3, allocates related files together and resisting fragmentation by leaving allocation gaps between groups.

Btrfs integrates multi-device management at the filesystem level. The devices can be mixed in size and speed, providing the admin more flexibility when managing large pools of storage. The long term goal is to be able to choose allocation policies to the underlying devices. Btrfs maintains both data and metadata integrity checksums and is able to detect corrupted copies of blocks and use the internal RAID code to pull up the correct data.

Btrfs supports a very limited form of transactions without ACID semantics: rollback is not possible, only one transaction may run at a time and transactions are not atomic with respect to storage. They are not analogous to databases transactions, but to the I/O "transactions" in ext3's JBD layer. An ioctl interface, however, is provided so that user processes can start transactions to make temporary reservations of disk space. Once started, all file-system I/O is then bound to the transaction until it closes, at which point the reservation is released and I/O is flushed to storage.

As of Linux 3.2, Btrfs implements all XFS features. In addition provides online volume growth and shrinking, online block device addition and removal, online balancing (movement of objects between block devices to balance load), object-level RAIDs, subvolumes, transparent compression, snapshots, file cloning (copy-on-write on individual files, or byte ranges thereof), checksums on data and metadata, in-place conversion (with rollback) from ext3/4 to Btrfs, and file system seeding (Btrfs on read-only storage used as a copy-on-write backing for a writable Btrfs).

Chapter 2

Design and Implementation

Bladefs is a minimal filesystem implemented in the Linux kernel. The main contribution of Bladefs is that it supports partitioning of the I/O path to achieve performance scaling and performance isolation on multi-core architectures.

Namespace management in Bladefs is implemented in the filesystem layer. Bladefs requires a partitioned allocator and a partitioned DRAM cache mechanism. Space management, DRAM caching, and a journal mechanism are implemented as separate modules. An example of Bladefs partitions is illustrated in Figure 2.1. Placement of files and directories into different partitions is managed at the filesystem layer. To maintain the integrity of its data structures, Bladefs distinguishes between data and metadata blocks, and delineates transactions for groups of metadata operations that need to be executed in an atomic manner.

The design of Bladefs aims to match to workloads with relatively small total number of files, but with rather large sizes such as VM's and database workloads. In the rest of this chapter we present our design decisions for Bladefs, and provide a detailed analysis of its features.

2.1 Bladefs namespace management

Three persistent structures are needed for namespace management in a Linux kernel filesystem: superblock, i-node, and dentry.

The persistent superblock contains all the information needed to mount a filesystem. This structure is initialised at filesystem creation time (using the generic `mkfs` utility program). The superblock stored at a predefined disk location, next to the the allocator's superblock. Mounting a Bladefs filesystem (using the generic `mount` utility program) triggers a read of the persistent superblock. The filesystem code then creates in-memory a standardized representation of the superblock (specified by the VFS subsystem of the Linux kernel). The persistent superblock is updated when we un-mount the filesystem.

The following fields make up the Bladefs superblock structure:

- *fs_magic*, Bladefs magic signature.

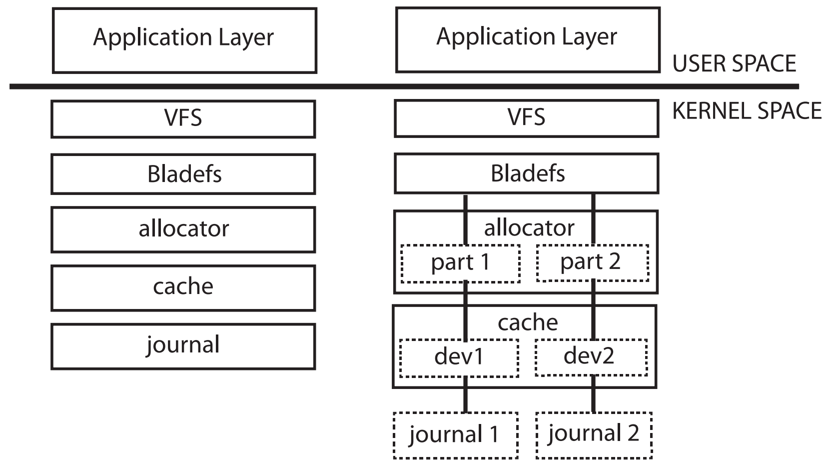


Figure 2.1: Bladefs: one and two partitions.

- *dev_total_size*, size of device filesystem created, in bytes.
- *superblock_size*, byte size of superblock data.
- *fs_block_size*, minimum block allocation size.
- *dev_block_size*, underlying device size.
- *dev_i-node_size*, size of the bladefs i-node.
- *root_i-node_addr*, byte address of root directory.
- *classes*, number of allocation unit classes.
- *allocations_units[]*, allocation units per class.
- *pointers_per_class[]*, number of pointers used for class.
- *max_file_length*, max file supported.
- *total_i-nodes_created*, number of i-nodes created.
- *total_i-nodes_deleted*, number of i-nodes deleted.
- *total_partitions*, number of partition bladefs instance has.

As in all traditional UNIX file-systems, each file or directory in Bladefs is described by an object called *i-node*. i-nodes are the basic building block for the file-system. Bladefs i-nodes are relatively large 4KB in contrast to other file-systems that are only few bytes (ext4 i-nodes are 256Bytes). Small i-node size is very critical in cases of many small files. Bladefs on the other hand can store all needed metadata for large sized files in one inode and is best suited targeted workloads. Also, Bladefs i-node contain information that most file-systems place them in denty such as name, type etc. In memory ,VFS and Bladefs i-nodes are packed together in a VFS i-node cache. The disk layout of bladefs i-node is shown in Figure 2.2.

Fields of persistent i-node structure of bladefs are the following:

- *magic*, stamp to clarify that disk block is used for i-node.
- *i-node_number*, unique number per i-node.
- *hard_link_i-node_addr*, used only to implement hard links.
- *type*, type of file (currently supported: regular file, directory, symbolic link).

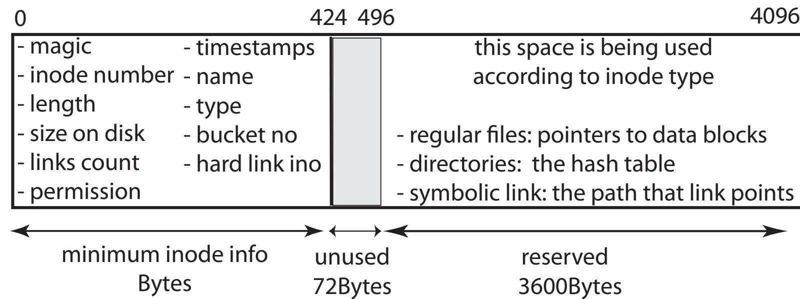


Figure 2.2: Bladefs disk i-node layout.

- *links*, how many hard links this i-node has.
- *allocated_ blocks*, how many 4K blocks we have allocated.
- *fsize*, length of file in bytes.
- *umode*, permissions modes.
- *gid*, group id of the file owner.
- *uid*, the user id of the file owner.
- *ctime*, creation time.
- *mtime*, last modifications of content of file time.
- *atime*, last access time.
- *name[]*, filename on regular files/ directory name for directories/ link name for symbolic links.
- *bucketno*, the directories hash table bucket number this i-node is stored.
- *partid*, the partition id that this i-node is created.
- *padding[]*, unused bytes reserved for possible future use.
- *pointers[]*, reserved space used according to i-node type.

Usually file-systems store in the dentries information such as the name of the entry that it points, the number, and the type of the i-node that is associated with. Keeping file name in the dentry increases the complexity of efficient storing the dentries. This happens since the file name can vary from 1 to 255 characters, that is the maximum file name length currently supported in the linux kernel. Instead of keeping the entry name we store an 8 byte hash of that name in our dentry. Apart from the hash of the entry name we also store the number of the i-node associated. For the in memory representation of the dentry we use the VFS dentry that is cached in the VFS dentry cache.

For simplicity, Bladefs does not separate metadata and data zones, i.e. both data and metadata blocks are spread out across the entire device address space. This decision does not adversely affect the type of workloads targeted by our design, where files are relatively few and therefore only need a relatively low number of metadata blocks.

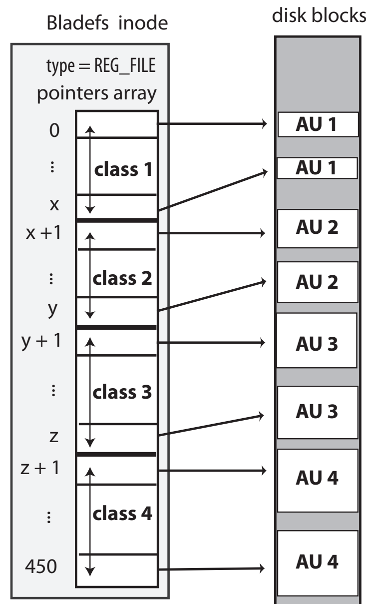


Figure 2.3: Bladefs files disk layout.

2.2 File structure and implementation

Files in Bladefs are implemented using i-nodes. These i-nodes are of type *regular file*. The reserved bytes in each i-nodes (3600 Bytes in a 4KB i-node) are used to hold an array of direct pointers that point to variable size data extents. Using variable size extents, rather than single file-system blocks, supports larger files. The various sizes of the extents are named *allocation units*. Pointers are grouped into *classes* depending on the allocation unit of the corresponding extents. These classes are stored in ascending order of allocation unit. The mapping between a user addressable byte offset in the file and the corresponding physically stored data block, is calculated by adding the allocation units of the pointer classes, until the sum exceeds the byte offset. The results of these operations are cached in a memory buffer to avoid re-calculation overheads. This design results to fast translation from byte offset to physical data block address in contrast to designs that use multiple levels of indirection, without additional I/O even for large file sizes.

During file-system creation (mkfs) you can specify the file-system block size, the maximum number of classes, the allocation unit for each class, and the number of pointers assigned to each class. The minimum bladefs block size is 4KB. Figure 2.3 shows the disk layout of a file of Bladefs using 4 classes of direct pointers.

This design of files can efficiently handle a large variety of file sizes, from very small (less than 4K) up to hundreds of GB, with all file meta-data stored inside the i-node. External fragmentation is also decreased since files grow using ever increasing allocation units. This results in some internal file fragmentation, since we pre-allocate space that might not be used. However, in the workloads we target

this is not the case since data base files tend to grow as they age.

The maximum file size depends on the number of direct pointers and the supported allocation unit sizes. Bladefs uses 64-bit pointers for blocks. Since blocks are multiples of 4KB, we "trim" the size of pointers by 12 bits. Keeping fewer bits per pointer increases the total number of direct pointers we can store in the i-node representation of a file.

Bladefs also supports *sparse files*, also known as "files with holes". For offsets that map to file "holes", we do not allocate space, unless a "hole" is within the limits of an allocation unit that is already being used.

File operations are implemented as follows:

- *open*: Opens a given file name by initiating the VFS i-node and dentry cache. Open support the following flags:
  O_APPEND: The file position pointer is set at the end of the file.
  O_SYNC: The file is opened for synchronous I/O and ensures that all writes that occur in that file will be return only when it is synchronised to the storage device. POSIX semantic of SYNC is to provide three variants of synchronised I/O flags: O_SYNC, O_DSYNC, and O_RSYNC. In our case, as most linux file-systems we do not implement fully O_SYNC semantics, which requires both data and meta-data to be written in the disk before system call returns but we synchronise only data.
  O_CREAT: Creates the file if it does not exist. It merely calls the directory i-node operation creat if the lookup operation fails for the specified file.
  O_TRUNC: in case the file exits it will be truncated to zero length.
- *read*: Tries to read the specified number of bytes from a file starting form the current position inside the file. Read increments the file position pointer according to the bytes that were read. It might not be able to return the requested number of bytes even under failure free execution. If it returns zero then we have already reached the end of file. In case the read requirers more than one file extends and accesses non contiguous extends, more than one read requests will issued by the file-system. Read modifies only the in-memory structure of the i-node as a result does not require any journal use. A readahead mechanism that is explained later is implemented to optimise this operation in case of sequential reads.
- *write*: Writes a specified count of bytes from the buffer that is passed as an argument to the file. Similar to the read operation write will also increase the file position pointer to the number of bytes that has written. In case the file is opened with O_SYNC flags, the function will return only when the written data are synchronised to the underlying storage device. When accessing more than one file extents that are not contiguously allocated, more than one writes will issue by the file-system. If a write request is not aligned and sized to the file-system block size then a read request will first be issued (read/modify/write). The allocator might be involved in a write operation, when the blocks allocated are not enough to fulfill the write request (file

append). The amount of space requested from the allocator depends on the current size of the file and the settings of class and allocation unites we have define at file system creation. Write operations modify both in-memory and persistent i-node state. The VFS layer issues write operations to the i-nodes that are cached in memory. Persistent updates pass throughout the journal mechanism to ensure recoverability. Finally, on writes we mark an in memory bitmap used to identify which extends are dirty, which allows us to improve the cost of sync operations.

- *lseek*: Moves the file position pointer according to the parameters passed. Locks the open file that the operation will be applied and does not modify any persistent structure of the file-system.
- *sync*: Synchronises all data and meta-data that are in cache. Both fsync and fsyncdata system calls use the same file operation and a flag is used to separate if a meta-data sync is required. For each data sync we scan the in memory bitmap that keeps dirty bits for file extents. For each dirty allocation unit we send a sync request to the cache for all the pages that an allocation unit contains. Metadata syncs, affect only the i-node, by sending a sync request at the page that contains the i-node.
- *release*: Drops the usage counter of the dentry object for that file. When the usage counter drops to zero the dentry object is released and the usage counter of the i-node associated with that dentry object is also decreased.

Both hard and symbolic links are supported in Bladefs. Symbolic links in Bladefs are implemented using only one i-node. The name of the symbolic link is stored in the name attribute of i-node and we use the reserved bytes to store the path that the symbolic links points to as layout at Figure 2.4. For proper support of symbolic links VFS requires to implement two extra calls.

- *readlink:* Returns the path that the link points to; and
- *follow link:* Parses the path that the link points to.

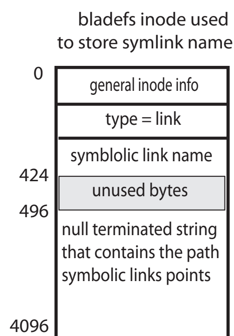


Figure 2.4: Bladefs symbolic link layout.

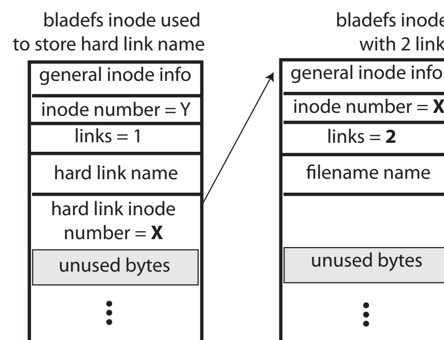


Figure 2.5: Bladefs hard links design.

Traditionally hard links in UNIX like file-systems are implemented using a new dentry that contains the link name and the i-node number of the file associated with. However, in Bladefs filenames are stored in i-nodes so apart from creating a dentry we also have to create an i-node to store filename. The links dentry points to the i-node that contains links name. The i-node used to store the file name also stores the i-node number of the linked i-node in the hard link i-node number. Finally we increase the link counter of the i-node that the new link has been created. An example of a hard link shown in Figure 2.5.

This design of hard links affects the lookup operation where we have to check if an i-node has set the hard link i-node number. If it has, then we add to the VFS cache the linked i-node information. When we delete a file we decrease the link counter of the i-node and in case this i-node has a hard link associated, we also decrease the link counter of the linked i-node. When the link counter of the i-node drops to zero, then the i-node is removed from VFS's caches and the allocated space in the device is freed.

An alternative to the above design of hard links, used by other files systems, would be to store hard links inside directory contents. However, this would increase the complexity of directory design and implementation, especially if we wanted to support unlimited number of hard links per directory.

Bladefs implements a per-file readahead technique. The last position of the file along with the number of sequential read access is held in memory for every opened file. Apart from the contiguous number of sequential reads, the total amount of data requested by the read request is also taken into account before triggering a readahead. Experimentally we find that after two sequential reads the mechanism should begin readahead. The maximum amount of readahead bytes is fixed. However, actual readahead data depends on the starting position pointer for the read within the file. Figure 2.6 illustrates three different cases in which readahead mechanism has been triggered. The difference between these cases is the amount of readahead pages, which is trimmed at the end of the allocation unit. Both the first and the second case are able to perform readahead but only in the second case the amount of readahead pages is the maximum allowed. In the third case even if the readahead mechanism is enabled we do not perform any readahead since the read operation reaches the end of the allocation unit.

2.3 Directory design

Next we discuss the design of Bladefs directories. Directory dentries are stored in a persistent hash table. i-nodes use the reserved bytes to store a hash table as shown in Figure 2.7. Each entry at the hash table points to a directory metadata page. A directory metadata page contains an array of dentries, a pointer to the next directory metadata page to implement collisions lists and a counter that keeps the number of dentries currently used inside the metadata page. Grouping dentries into pages results in all read and write requests for directory metadata being aligned to

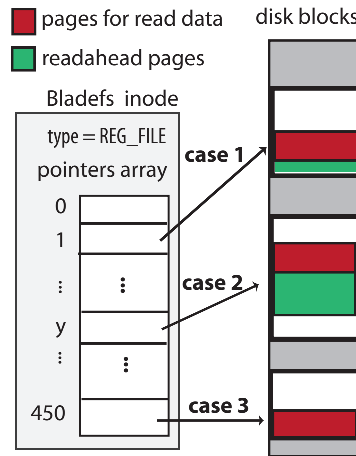


Figure 2.6: Bladefs readahead.

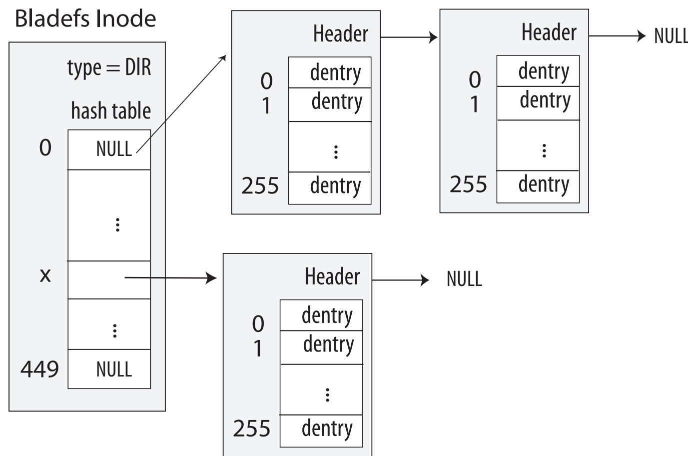


Figure 2.7: Bladefs directories structure.

pages.

Although the workloads we target have only a few entries per directory this design is able to scale well as the number of entries per directory increases. A directory metadata page we can store up to 255 entries. Each consists of 16 Bytes; 8 Bytes for the i-node number and 8 for the hash. With 4K i-nodes we have up to 450 pointers to the bucket lists. Since each directory metadata page has 255 first entries of buckets it can store up to 114,750 entries.

Directory operations are implemented as follows in Bladefs:

- *lookup*: Searches for the existence of a specified name inside a directory. VFS is responsible to tokenise relative and absolute paths and request a lookup for each token of the path. If the name exists, it fills up the dentry and i-node structures of VFS for that entry and initialise a new binding between them.

In case there is no entry in the directory with the specified name, it returns NULL.

- *create*: Creates a file with the specified protection mode. It is up to VFS to verify that there is not an entry inside the directory with the same name. File creation invokes a call to allocator to reserve space for the new i-node. The physical address of the new i-node is the i-node number. The file system has to instantiate the dentry cache with the newly created file. File names are limited by VFS to 255 chars. A dentry is also created and inserted at the directory hash table. A filesystem transaction ensures that dentry insertion at directories hash table will be performed in an atomic manner.
- *mkdir*: Creates a new directory. Operations that occur at mkdir are the same ones as for create.
- *link*: Creates hard links within the same partition of the filesystem. We discussed in the previous section.
- *symlink*: They are i-nodes that have as data the path that the link points to. The maximum path is 3600 bytes and the linked file might cross device. In case we deleted part of the path or the actual file that a symbolic link points to the symbolic link will be broken. Besides the i-node that stores the path name, a dentry is also created. Insertion of the dentry in the directory hash table is inside a transaction to ensure that it will be atomic.
- *rename*: Changes the name of file or directory. In case, where an entry with the same name as the destination exists, first we delete that entry. In our case we create a new i-node, we copy all information that the old i-node holds, and we overwrite the name with the new one. Then we insert the newly created i-node to the directory and we delete the old i-node. Creation of the i-node and the directory operations occur are within a transaction.
- *unlink*: Decreases the link counter for a file. If the links counter drops to zero then it is also removed from the directory structure. When the VFS i-node is evicted from the VFS i-node cache, we check if the links counter is equal to zero and we free the allocated space for that i-node. The deletion of the dentry from the directory is performed in a transaction.
- *rmdir*: removes a directory. The directory we want to delete must be empty, in which case it is merely unlined.
- *readdir*: Fills up a dirent object with the next avaliable directory entry. This call invoke a use of an iterator to store the current position in the process of reading the directory. Directory entries are not returned in a lexicographic order but as they are inserted in the hash table.

Besides file and directory i-node operations Bladefs implements also the following operations to handle permission and VFS required functions.

- *setattr*: Changes i-node attributes (permissions, owerships etc.).
- *getattr*: Returns i-node attributes (permission, owerships, size, length etc.).
- *mkfs*: It is a user level utility to create a file bladefs file-system over a block

device. The parameters of mkfs are: number of partitions, allocation units size, pointers per class, and the number of classes.

- *mount*: Attaches the device that has a bladefs file-system into the file-system tree. This operation invokes the allocation of an instance of the VFS i-node cache, the read of Bladefs superblock and root i-node that are also inserted to the VFS caches.
- *read i-node*: Reads a persistant i-node and initialises the in memory VFS i-node that corresponds to it.
- *write i-node*: Synchronises the in-memory i-node with the persistent structure. VFS sends asynchronous calls of this operation to synchronise the VFS i-node cache with the underlying device. This operation is used also in the sync and fsync system call.
- *delete i-node*: remove i-node from VFS i-node cache. Also de-allocate i-node disk space, called when link counter drop to zero.
- *drop i-node*: Remove i-node from VFS i-node cache, called when the usage counter of the i-node drops to zero.
- *umount*: Detaches a previously mounted bladefs instance. Drops all VFS meta-data and free the i-node cache. Write the superblock.
- *statfs*: Report statistics from the mounted file-system (used, available, total space, number of existing i-node etc.)

2.4 Data and Metadata placement

Placement of data and metadata in different partitions is determeined during allocation time and affects system performance. Policies for choosing the partition can take into account the filename, the directory name or I/O usage of each partition. We choose to implement a policy that is easy to control from the user of the file system. Figure 2.8 illustrates the implemented policy. The superblock and the root inode that are allocated during file system creation we place them at the first partitions. Directories that are one level below the root directory are placed in a different partition in a round-robin order. All the other entries are placed in the same partition as their parents. File data blocks and directory metadata blocks inherit the partition of the parent inode.

2.5 Bladefs reliability

The journal component described in the background section is used by Bladefs to ensure recovery. Transactions offered by journal module used at Bladefs to perform atomic modification in the persistent metadata. Bladefs is responsible to initiate and end transactions. Each partition of the file system operates an a separate journal module. With this design it is possible to have partitions that do not offer reliability if we do not use a journal.

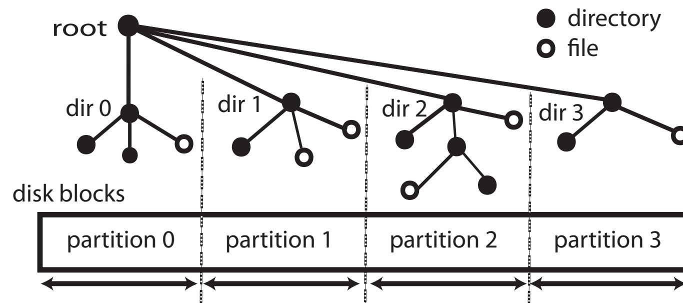


Figure 2.8: Placement of data and metadata in partitions.

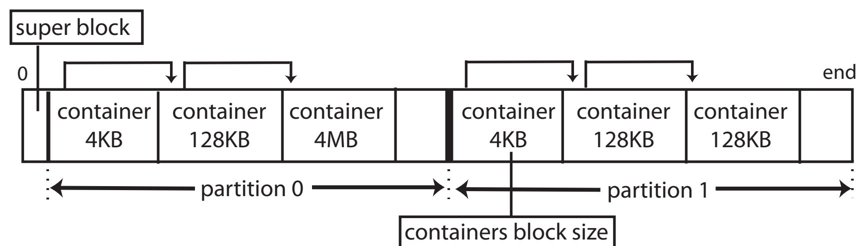


Figure 2.9: Allocator disk layout.

2.6 Partitioned allocator

Bladefs relies on a block device allocator that is implemented as a separate module. Separating the allocator from the file-system module simplifies file-system complexity and resulting in a bare minimum file-system layer. It also provides us flexibility to place the allocator in the guest or the host at VM's configurations. In order to meet Bladefs requirements allocator supports partitioning of the address space that are equally sized divisions of it. The allocator offers the flexibility to decide in which partition the allocation will be performed. A partitioned allocator provided on top of the allocator described in the background section. The disk layout of the partitioned allocator is illustrated in Figure 2.9.

2.6.1 Allocator operations and disk layout

The allocator implements the following operations:

- load(device): Checks for the allocator superblock and loads meta-data in memory.
- *init(number of partitions)*: initialises an allocator with the number of partitions requested. This call is used at file-system creation (mkfs).
- *shutdown()*: Flushes all the in-memory allocator meta-data (superblock and containers meta-data) on to the disk.

- *alloc(size, partition id)*: Allocates the number of bytes into the partition passed as a parameter. The allocator checks with the pre-existing containers to fulfil the request. If there is no existing container that can serve the requested allocation, or there is no space in containers that already exist, then a new container will be created. The block size of the new containers matches the requested allocation. Currently, the total size of the new container is predefined. A dynamic calculation based on the total device size is yet to be implemented. If the partition, passed as parameter, does not have enough consecutive free space to serve the request the allocation will fail even if another partition has free space.
- *free (first byte, size)*: Frees the number of bytes passed as a parameter starting from the first byte. The free operation does not specify the partition since this can be calculated using the address of the first byte that passed as a parameter.
- *get_stats()*: Returns the current allocated and free space of the device along with the number of blocks used for metadata.

Chapter 3

Evaluation

In this chapter we present the evaluation of Bladefs using both real-life applications such as OLAP[1] and OLTP[2] along with micro benchmarks. The evaluation is preformed in a x86-based Linux sever. XFS, that is commonly deployed in Linux kernel, is used as a comparison file system for Bladefs.

The evaluation of Bladefs is made on three aspects. First we provide a baseline single partition comparison of Bladefs with XFS. Then we present the effects of file system partitioning in competing workloads. Finally we evaluate the performance scaling of Bladefs with the number of partitions.

3.1 Benchmarks

In our evaluation we use the following benchmarks.

3.1.1 Make many files:

This single threaded micro benchmark creates a file tree with the number of requested files at the leaf directories. It is a metadata write intensive benchmark that stresses the journal module.

3.1.2 kfsmark:

A modified version of fsmark [1] that is widely used to evaluate file systems. The fsmark benchmark tests synchronous file system operations. It varies the number of directories, threads, files, file size that it uses. kfsmark reports operations per second for each system call. Using kfsmark we measure the I/O operation per second (IOPS) scalability as we increase the number of partitions.

[1]OLAP refers to online analytical processing.
[2]OLTP refers to online transaction processing.

3.1.3 TPC-H:

The TPC Benchmark H (TPC-H) [29] is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

(TPCH) generates only read operations. We execute queries Q1 to Q4 back to back and in this order. Each query generate different access patterns for the file system. Q1 and Q4 generate sequential reads at the files that store database tables. However, they differ in the request size. Q1 has a merged request of 128KB, whereas Q4 an average request size of 8 KB. The other two queries that we test generate random reads with small request sizes. In our case we use MySQL [20] database server and populated database with total size of 6.7 GB.

TPCH is used to evaluate single thread read performance of the system.

3.1.4 TPC-W:

TPC Benchmark W (TPC-W) [30] is a transactional web benchmark. The workload is performed in a controlled internet commerce environment that simulates the activities of a business oriented transactional web server. The workload exercises a breadth of system components associated with such environments. The performance metric reported by TPC-W is the number of web transactions processed per second and the average transaction time. Multiple web interactions are used to simulate the activity of a retail store. Each interaction is subject to a response time constraint.

TPC-W performs mostly fsync operations that are used to maintain the database logs file. This results to many synchronous write calls to the underling device. In our setup the populated database size we use is 2.7 GB and the database sever is postgreSQL [27].

3.2 Evaluation Platform

The components of the platform that is used to evaluate Bladefs are: a dualsocket Tyan motherboard with two 4core Intel Xeon 5520 64bit processors running at 2.26 GHz, with two hardware threads per core, 12 GB of DDR-III DRAM, and 8 32GB enterprisegrade Intel X25-E SLC NAND-Flash SSDs, connected to one LSI MegaSAS 9260 storage controllers. When we create RAID0 configuration we use the default md [11] Linux driver, with a chunk-size of 64KB. The Linux distribution installed is CentOS, v.6.0.

3.3 Single partition Bladefs performance evaluation

For the base line comparison of Bladefs with XFS we use Make many files, TPC-H and TPC-W benchmarks.

3.3.1 Make many files

Table 3.1: Make many files creates per second.

# of instances	XFS	Bladefs	Bladefs without journal
1	506	172	230
2	370	188	220

We run make many files with a tree of 27 leaf directories and 15000 files per directory. This results that a single instance of make many files creates 40500 files and two concurrent instances 810000 files. Apart from file creation we also write a 4KB buffer to each file so that the newly created i-node will be modified again and generate from I/O traffic. We have observed that XFS journal consumes all available memory before issuing write operations to the core device. For that reason we boot the system with only 3GB total memory so that we can stress XFS journal to flush operations.

Table 3.1 illustrates file creation ratio per second. The settings for the Bladefs column are: 2GB private memory for the cache module and 256MB private memory for the journal module that is used for the in memory circular buffer. We also report number without the journal device to measure the journal overhead. Note that in this case file systems metadata are also written in the device since cache policy for metadata is write through. Moreover, we see that Bladefs performance does not decrease in the scenario with two concurrent instances.

3.3.2 TPC-H

Next we present the evaluation with the TPC-H benchmark. For this benchmark we use a mode that Bladefs support to use it without the need of the cache module and instead we use the buffer cache of the system. In this mode Bladefs use BIOS calls to the underlying device. This configuration minimises the overhead of cache but we always incur the penalty of an extra copy and the creation of BIOS.

First we examine the perfromacne using a ramdisk to store the database. As shown in Figure 3.1 Bladefs performance is simmilar to XFS.

Then, we use as an underlying device a loop device that is linked with an SSD and to use the buffer cache mechanism of the linux kernel to perform I/O to the device. We choose to boot the machine with only 3 GB total memory that is less than half size of the database that we used so that evictions from the cache will be produced while running the queries. Figure 3.2 shows the comparison between XFS and Bladefs in BIOS mode on top of loop device. We also provide results

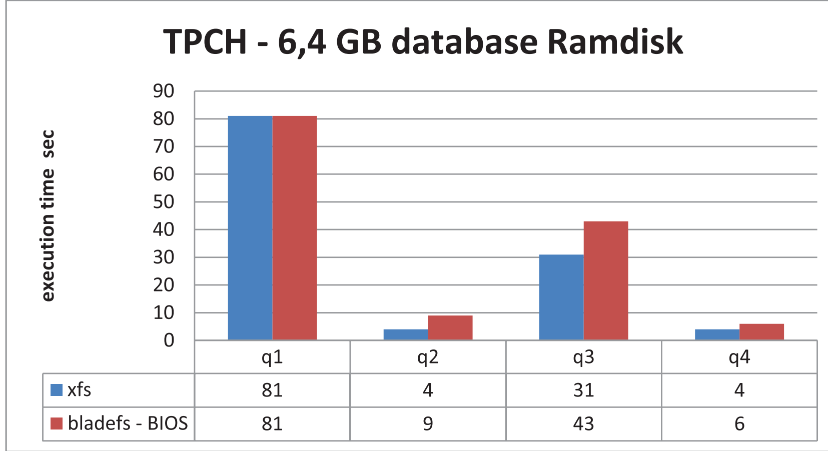


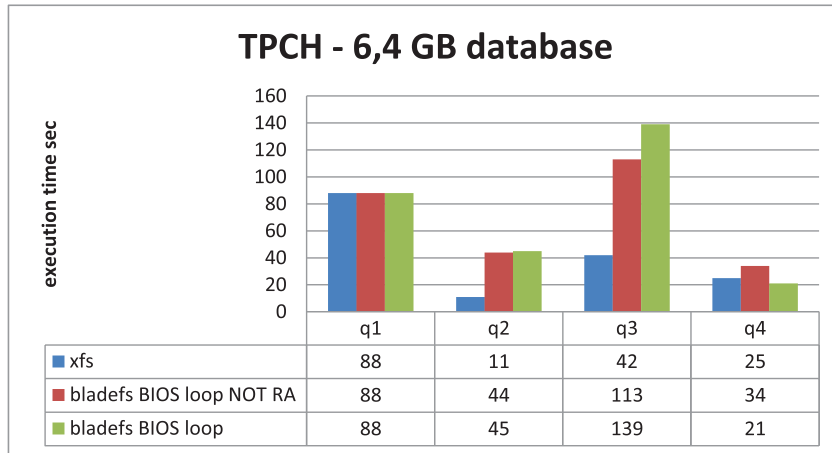
Figure 3.1: TPC-H queries Q[1-4] XFS and Bladefs with BIOS and ramdisk.

with disabled the read-ahead mechanism that Bladefs supports. We see that in Q1 which issues large sequential reads (streaming) our results are the same as XFS and that the read-ahead mechanism does not affect the performance. For Q2 which has many random and very small requests XFS outperforms Bladefs. This is due to the extra copy that Bladefs does due to block I/O interface. In Q3 that is mostly random, small reads difference between Bladefs and XFS is much higher for the same reason as in Q2. Q4 that consists of small and sequential read requests Bladefs, with read-ahead performs even better that XFS.

More realistically we evaluate Bladefs with TPC-H. We use the cache and journal modules. In this scenario the cache has 2 GB of private memory and the journal 32 MB. Similar to the previous setup we use only 3GB of memory. Figure 3.3 showns the execution times. Q1 is the same with the XFS. However, for all the other queries execution time is increasing due to cache inefficient to small requests when we use cache module. This inefficiency is due to the fact that cache module operates with its own threads and context switch between application threads, that file system use, is always needed. This context switch cost more than the actual copy of few bytes. We also observe the performance improvement when we use the readahead mechanism.

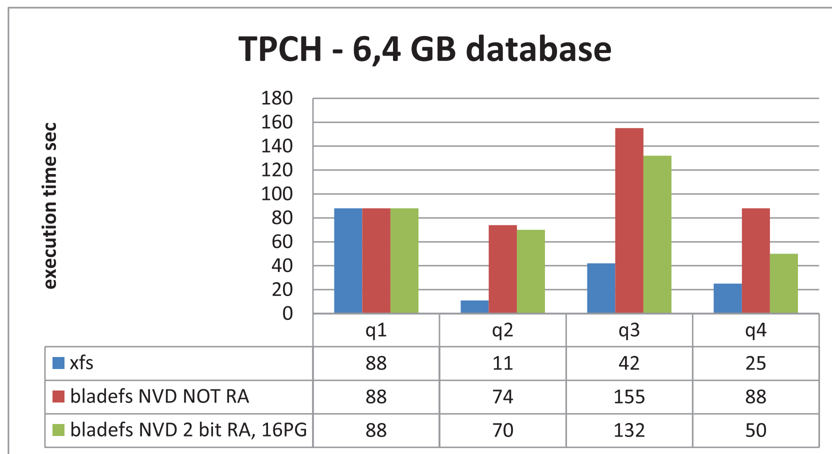
3.3.3 TPC-W

Figures 3.4 and 3.5 shows the throughput and the average transaction time respectively for loads from 8000 to 28000 clients. We store the database in a single SSD. With this benchmark we want to illustrate the synchronous write performance of Bladefs. To avoid any evictions from the cache due to limited size we use 12GB memory that are more enough for the workload that we execute. For Bladefs we provide numbers for two different configuration of the private memory assigned to the cache module, 2GB and 4GB respectively. We observe that between



	q1	q2	q3	q4
xfs	88	11	42	25
bladefs BIOS loop NOT RA	88	44	113	34
bladefs BIOS loop	88	45	139	21

Figure 3.2: TPC-H queries Q[1-4] XFS and Bladefs with BIOS on top of loop device.



	q1	q2	q3	q4
xfs	88	11	42	25
bladefs NVD NOT RA	88	74	155	88
bladefs NVD 2 bit RA, 16PG	88	70	132	50

Figure 3.3: TPC-H queries Q[1-4] XFS and Bladefs with cache and journal.

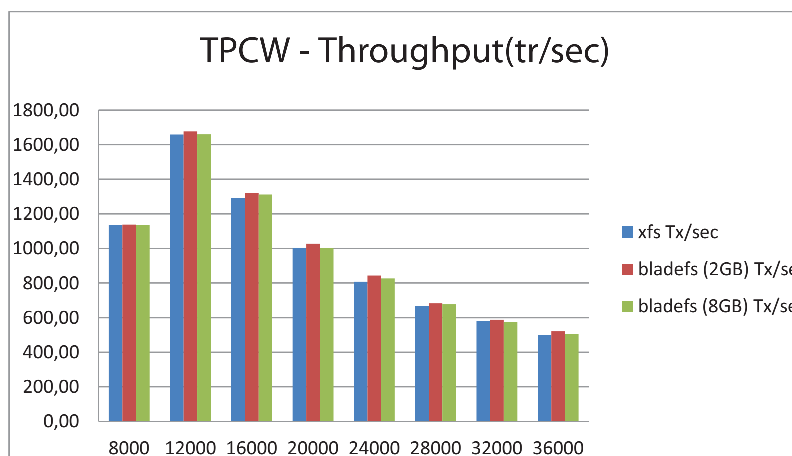


Figure 3.4: TPC-W throughput for XFS VS. Bladefs.

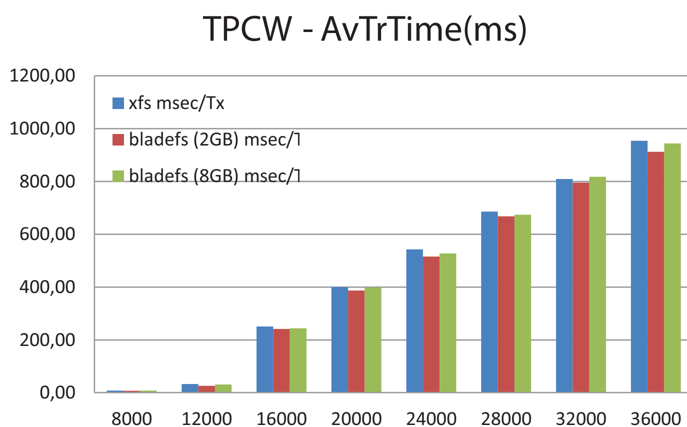


Figure 3.5: TPC-W average transaction time for XFS VS. Bladefs.

the different configurations there is not much difference. This is due to the fact that with the database workloads we use (2.7GB) TPC-W does not require more than 2GB memory to operate without cache evictions. Moreover, we observe that Bladefs throughput is slightly better than XFS and also average transaction time is slightly less than XFS.

3.4 Isolation between Competing Workloads

To demonstrate the behavior of Bladefs in competing workloads we use TPC-W and a single thread (WT) that executes sequential writes every 3 seconds. WT consumes memory that TPCW uses as cache. To evaluate this scenario we use Bladefs with two partitions and we compare with XFS. The set up of this workload

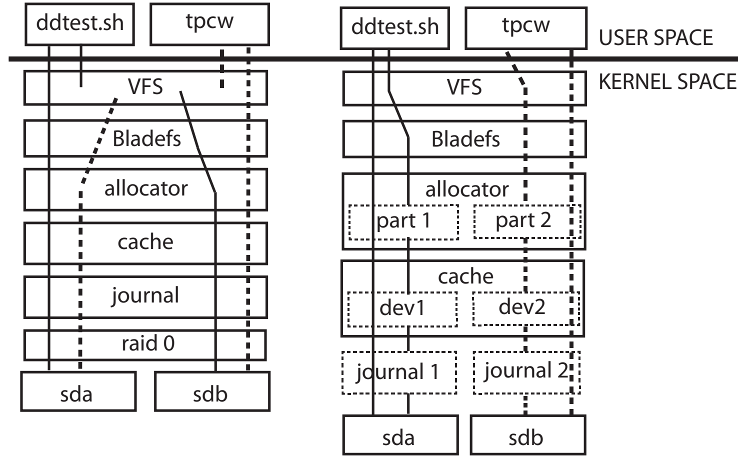


Figure 3.6: Bladefs with two partitions, one for the TPC-W, and one for the Write thread.

is illustrated in Figure 3.6. In this experiment we use a RAID0 that consist of two SSDs as the storage device. We limit the available memory of the machine to 4GB so that WT and TPCW will compete for this resource.

We run TPC-W alone both for XFS and Bladefs. In Bladefs run we assign 3GB private memory to cache layer. Then we run both workloads together. We observe that throughput and especially average transaction time of TPCW are highly affected when we run them simultaneously. Finally, we run also the same experiment with 2 Bladefs partitions, where we split the memory used by cache module to 2GB for the partition that will run TPC-W and 1GB for the other one. Moreover, in this case each partition operates over a separate device so that access to devices are also isolated. We observe that when we separate the different workload to run into isolated partitions then TPCW average transaction time is only 2X more than when it is run alone.

Table 3.2 summarizes the average transaction time in milliseconds for the scenarios described above. Table shows 3.3 the throughput of TPC-W. The average throughput of WT is 150MB/s with XFS, 360MB/s with Bladefs within a single partition and 350MB/s when we have two partitions.

Table 3.2: TPCW average transaction time in msec.

XFS + WT	Bladefs + WT	Bladefs + WT 2 Partitions	XFS	Bladefs
2644,01	22173,16	74,73	34,72	11,71

Table 3.3: TPCW throughput transaction per second.

# XFS + WT	Bladefs + WT	Bladefs + WT 2 Partitions	XFS	Bladefs
89,98	16,88	105,91	140,33	138,61

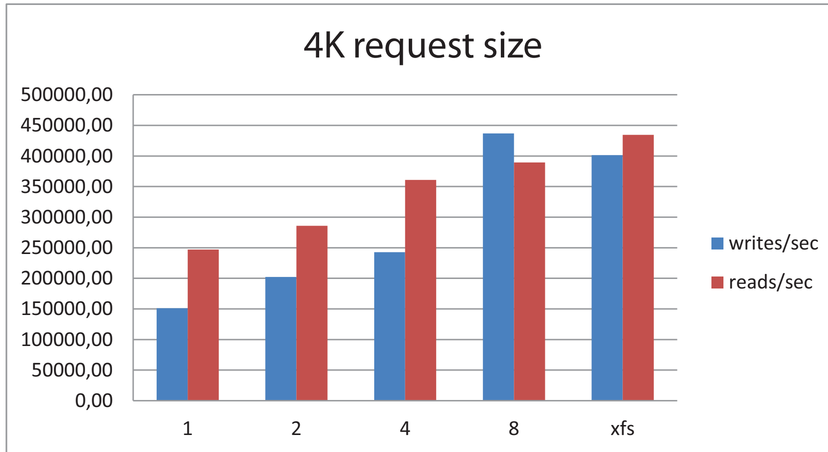


Figure 3.7: kfsmark READ / WRITE 8 SSDs request size 4KB.

3.5 Performance scaling with the number of partitions

The last aspect of Bladefs that we evaluate is the performance scaling with the number of partitions. For this case we use kfsmark benchmark configured to use multiple partitions. Each thread of kfsmark operates at a private directory. The directories that are created at the top level belong to different partitions with round robin-order. This give us the ability to isolate threads to different partitions.

To execute this benchmark we use a RAID-0 that consists of 8 SSDs as the storage device and 12GB memory. kfsmark is configured to use 32 threads. Each thread creates 64 files of 32 MB size that results to a 64GB total data I/O size that is more 5 times the memory size. For Bladefs runs we assign 8GB as private memory to the cache module and 256 MB for the journal. When we increase the number of partitions then we divide the memory assign to the cache module and journal to equal sizes portitions for each cache partition and journal. Figure 3.7 shows operations per second with a 4KB request size, with 1, 2, 4, 8 partitions and XFS. We observe that IOPS scale with the number of partitions. Figure 3.8 shows that we achieved similar performance scaling using 64KB request size for the same set up. This is due to higher device utilization that is being achieved since we minimise I/O wait time.

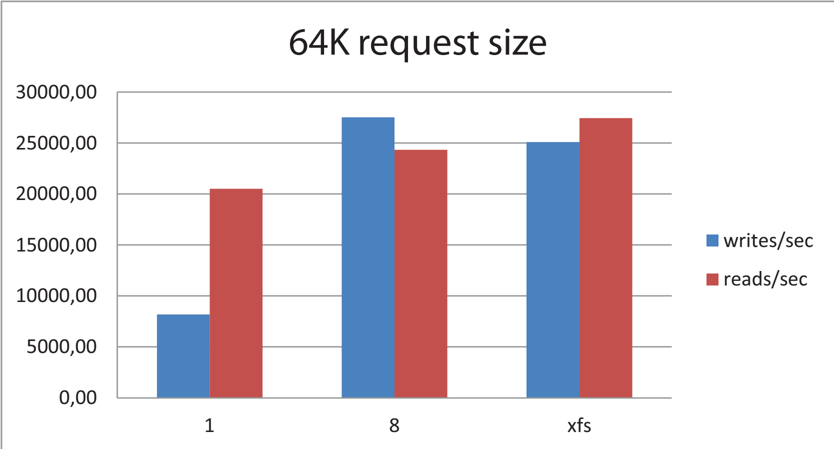


Figure 3.8: kfsmark READ / WRITE 8 SSDs request size 64KB.

Chapter 4

Conclusions and Future Work

In this thesis we present Bladefs a kernel level file system that targets multi-core and multi-socket architectures. Bladefs addresses this challenge via partitioning of the storage address space. To achieve partitioning Bladefs uses an extended partitioned cache and a partitioned block allocator. Moreover, recovery functionality of Bladefs is implemented in a separate partitioned journal layer. Partitions of Bladefs can be assigned to different cores, sockets, and hard drives to eliminate interference between competing workloads and also to provide scalable performance as the number of partitions increases. We evaluate Bladefs using real-life applications along with micro-benchmarks.

In our evaluation we use a combination of a write intensive workloads, that do not requires a significant amount of cache memory, and a complex workload that uses cache memory for reads and issues many synchronous writes with small request sizes. Our results show that when we run such workloads in parallel, the write intensive workload impacts the performance of the other workload consuming memory that is not required. Assigning workloads to separate partitions results in the performance of each workload remaining unaffected. In addison, we show that multiple partitions can improve systems scaling. We see a speed up of a 2.6X for read IOPS and 1.6X for write IOPS using 8 partitions and compared to the performance of single a partition.

Currently Bladefs partitions are fixed and equal sized. Unequal and dynamic size partitions can provide more flexible load balancing across cores.

Overall, we believe that techniques such as I/O partitioning will be used in storage servers to offer both isolation and performance scaling to applications.

Bibliography

[1] fsmark. http://sourceforge.net/projects/fsmark.

[2] Orlov block allocator.

[3] XFS. http://xfs.org.

[4] Stefan Aulbach, Michael Seibold, Dean Jacobs, and Alfons Kemper. Extensibility and data sharing in evolving multi-tenant databases. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 99–110, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Avantika Mathur, Mingming Cao, Suparna Bhattacharya. The new ext4 filesystem: current status and future plans. Linux Symposium Volume Two, 2007.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] Cor-Paul Bezemer and Andy Zaidman. Multi-tenant saas applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92, New York, NY, USA, 2010. ACM.

[8] Chris Mason. Btrfs. https://btrfs.wiki.kernel.org.

[9] Dave Chinner. XFS - Recent and Future Adventures in Filesystem Scalability. In *In proceeding of Linux conference.*

[10] Dave Chinner. XFS: The big storage file system for Linux. In *Linux Symposium 2012.*

[11] Miguel de Icaza, Ingo Molnar, and Gadi Oxman. The linux raid-1,-4,-5 code. In *LinuxExpo*, apr 1997.

[12] Duy Le, Hai Huang, Haining Wang. Understanding Performance Implications of Nested File Systems in a Virtualized Environment. In *Proc. of the 7th USENIX Conf. on File and Storage Technologies*, Berkeley, CA, USA, 2012. USENIX Association.

[13] Patrick Fabian, Julia Palmer, Justin Richardson, Mic Bowman, Paul Brett, Rob Knauerhase, Jeff Sedayao, John Vicente, Cheng-Chee Koh, and Sanjay Rungta. Virtualization in the enterprise. 10(3):227–242, August 2006.

[14] Changjie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. In *CEC/EEE*, pages 551–558, 2007.

[15] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[16] J. F. Gantz, D. Reinsel, C. Chute, W. Schlichting, J. Mcarthur, S. Minton, I. Xheneti, A. Toncheva, A. Manfrediz. IDC, 2007.

[17] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance isolation and differentiation for storage systems. In *In International Workshop on Quality of Service (IWQoS*, pages 67–74, 2004.

[18] Thomas Kwok and Ajay Mohindra. Resource calculations with constraints, and placement of tenants and instances for multi-tenant saas applications. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 633–648, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Jeanna Neefe Matthews, Wenjin Hu, Madhujith Hapuarachchi, Todd Deshane, Demetrios Dimatos, Gary Hamilton, Michael McCabe, and James Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[20] MySQL AB. MySQL: The World's Most Popular Open Source Database, 2005.

[21] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, New York, NY, USA, 2008. ACM.

[22] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical report, 2007.

[23] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[24] Rodeh, Ohad. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):2:1–2:27, February 2008.

[25] Shoaib Akram, Manolis Marazkis, and Angelos Bilas. NUMA Implications for Storage I/O Throughput in Modern Servers. In *In 3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12) , In conjunction with the 7th International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC'12).*

[26] Gaurav Somani and Sanjay Chaudhary. Application performance isolation in virtualization. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09, pages 41–48, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Stonebraker, Michael and Rowe, Lawrence A. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM.

[28] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. *SIGARCH Comput. Archit. News*, 39(3):283–294, June 2011.

[29] Transaction Processing Performance Council. TPC-H: an ad-hoc, decision support benchmark. http://www.tpc.org/tpch/.

[30] Transaction Processing Performance Council. TPC-W: Benchmarking An Ecommerce Solution. http://www.tpc.org/tpcw/.

[31] Zhi Hu Wang, Chang Jie Guo, Bo Gao, Wei Sun, Zhen Zhang, and Wen Hao An. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In *Proceedings of the 2008 IEEE International Conference on e-Business Engineering*, ICEBE '08, pages 94–101, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[33] Yannis Klonatos, Manolis Marazakis, and Angelos Bilas. A Scaling Analysis of Linux I/O Performance. In *Poster at EuroSys 2011.*