pFS: A partitioned filesystem targeting Virtual Machine images

Anastasios Papagiannis[*,1],
Yannis Sfakianakis[*,1],
Stelios Mavridis[*†,1],
Manolis Marazakis[*,1],
Angelos Bilas[*†,1]

[*] *Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), 100 N. Plastira Av., Vassilika Vouton, Herakleion, GR–70013, Greece*
[†] *Department of Computer Science, University of Crete*

ABSTRACT

In this paper we present a new filesystem for storing virtual machine images. In the current filesystem design all I/O calls pass through a singe path inside the Linux kernel, resulting in contention on shared resources and interference along independant virtual machine images. We propose a partitioned I/O path through Linux kernel to minimize the contention and inteference. This partitioned scheme contains a filesystem and an allocator. The other parts are a partitioned DRAM cache and partition journal meachanism which are beyond the scope of this work.

KEYWORDS: I/O, filesystem, virtual machine, interference, storage

1 Introduction

Filesystems have been examined for a lot of years in many aspects. Recent technology evolution of multicore servers and SDD drives gives new opportunities for more optimization in filesystems. In [KMB] the authors show that current filesystem design does not scale. In this work we present an application specific filesytem targeting virtual machine images. The resources of modern servers are capable of managing large number of virtual machines. Although there has been a lot of work in virtual machine managers like KVM, Xen and VMWare, there is still space for optimization in the filesystem, which is responsible for storing the virtual machine images. Keeping this in mind we can make several assumptions for this type of files. VM images are generally large in size with limited need for resizing. Each virtual machine image is independant from the others. Using these assumptions we can provide several optimizations in the design of I/O path. Because of the fact that virtual machine images are generaly large and resizing is very rare we can apply several optimizations for more effiecient read/write path. Also the block allocator can be less sophisticated thus resulting in less overheads during block allocation and deallocation. Another very important

[1]E-mail: {apapag, jsfakian, mavridis, maraz, bilas}@ics.forth.gr

part of managing virtual machine images is the need of isolation. VM images have to share a single I/O path with the current linux configuration, which results in significant performance penalty [KMHK12]. There are currently Linux mechanisms like cgroups to provide isolation but this is not proven to work effieciently.

In this work we present the design and the implementation of a filesystem and an allocator for managing virtual machine images. We refer to these partitions as I/O slices. Our new I/O path consists of a filesystem which is responsible for namespace managment and a low overhead allocator. The filesystem is responsibe for dispatching each I/O call to the correct I/O slice. We also provide a partitioned DRAM cache and partitioned journal mechanism to support reliability. The last of them are beyond the scope of this work.

The rest of this paper is organized as follows: Section 2 provides the background of the current I/O path and filesystem design. In Section 3 we show the design of a partitioned block allocator, while in Section 4 we describe the design of the partitioned filesystem.

2 Background

A typical filesystem implements the following functionality:

1. **Namespace management:** provides a mapping between a filename and the content of the file. Hierarchical directories are also provided to group files. Namespace management requires extensive use of metadata. The most important elements of filesystem metadata are:

    - *inodes* are the low level representation of a file. They contain a unique numerical identifier, pointers to allocated blocks, disk usage of the file, file length, file permissions, and timestamps (creation/last-access/last-modification).

    - *dentries* contain a binding between a filename and the directory that this file belongs.

    - The *superblock* contains general information about the filesystem such as: filesystem block size, address of root inode, how many files have been created in the file system etc.

2. **Reliability:** Filesystems provide a mechanism to keep a consistent state of the machine even in presence of power failures or system crashes.

3. **Block Allocation:** Filesystem operations may have to allocate or free blocks, so there is a need to keep track of unused blocks on the block storage device underlying the filesystem, which requires additional metadata.

4. **DRAM caching:** Storage devices throughput is orders of magnitude slower than the CPU, making it imperative to rely on caching recently accessed blocks in non-persistent DRAM to increase filesystem performance.

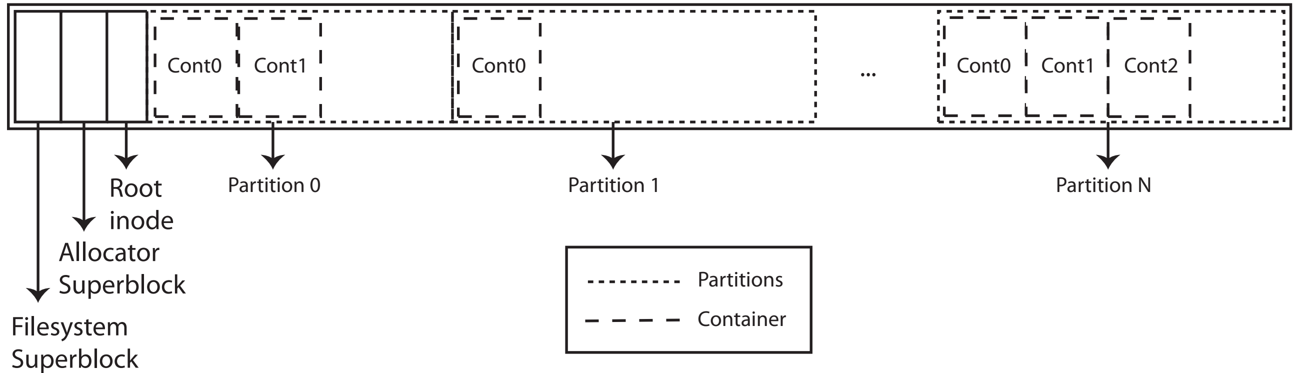

Figure 1: Allocator disk layout.

3 Allocator Design

The allocator is responsible for allocation and deallocation of disk blocks. Figure 1 shows the layout of our allocator on disk. First of all comes the filesystem superblock, then the allocator superblock and at last the root directory inode. Each has size equal to 4KB. The root directory inode is allocated in partition 0. We split each partition in smaller parts called containers. There are 20 different types of containers. In each container type we store objects of the same size. The available container types support different object sizes and begin with object size equal to 4Kb (*type0*) and end with object size equal to 2Gb (*type19*). The size of remaining types is:

$$sizeof(typeX) = sizeof(type(X-1)) * 2$$

For example container with type 0 allocates only 4Kb blocks and container with type 19 stores 2Gb blocks. Each container has a fixed size depending of it's type and the number of available objects is equal to the $container_size/object_size$. For each container we keep 4Kb of metadata containing the bitmap for allocated blocks and other information needed. We assign to each container type a seperate free-list. When an allocation is performed we firstly search the corresponding free-list and if there is no block available we have to allocate a new container. In the case of the free-list is not empty we simply get the block from it. For the deallocation process we simply add the deallocated block in the corresponding free-list. In both cases we have to update the on-disk bitmap to ensure persistanse. We also have seperate locks for each partition and container type to minimize contention. This allocator design achieves low overhead allocation and deallocation of fixed size blocks from 4Kb to 2Gb. The filesystem has to manage these type of blocks to create arbitary size files.

4 Filesystem Design

pFS is designed to implement the minimum required functionality for a filesystem to be functional. We implement only the namespace management in the filesystem layer, and provide the remaining functionality below the filesystem layer. Block allocation is handled via calls to a dedicated allocator module described in Section 3. Caching and journaling are addressed by distinct modules which are beyond of this work.

pFS has to translate file operations to block operations. The resulting block operations are issued to the next layer (DRAM cache) taking into account the partitioned cache design. To achieve the translation from file operations to block operations, each inode contains an arrays of pointers to disk blocks. Each of these are used for a fixed block size. This block managment allows the creation for files with maximum size equals to 231Gb.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the IOLANES (FP7-ICT-248615) and HiPEAC2 (FP7-ICT-217068) projects.

References

[KMB] Yannis Klonatos, Manolis Marazakis, and Angelos Bilas. A Scaling Analysis of Linux I/O Performance. Poster in Eurosys 2011.

[KMHK12] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 51:1–51:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.