Towards a Highly Adaptable Filesystem Framework for Linux

Suparna Bhattacharya
Linux Technology Center
IBM Software Lab, India
suparna@in.ibm.com

Dilma Da Silva
IBM T.J. Watson Research Center, USA
dilmasilva@us.ibm.com

Abstract

Linux® is growing richer in independent general purpose file systems with their own unique advantages, however, fragmentation and divergence can be confusing for users. Individual file systems are also adding an expanding number of options (e.g. ext3) and variations (e.g. reiser4 plugins) to satisfy new requirements. Both of these trends indicate a need for improved flexibility in file system design to benefit from the best of all worlds. We explore ways to address this need, using as our basis, KFS (K42 file system), a research file system designed for fine-grained flexibility.

KFS aims to support a wide variety of file structures and policies, allowing the representation of a file or directory to change on the fly to adapt to characteristics not well known a priori, e.g. list-based to tree-based, or small to large directory layouts. It is not intended as yet another file system for Linux, but as a platform to study trade-offs associated with adaptability and evaluate new algorithms before incorporation on an established file system. We hope that ideas and lessons learnt from the experience with KFS will be beneficial for Linux file systems to evolve to be more adaptable and for the VFS to enable better building-block-based code sharing across file systems.

1 Introduction

The Linux 2.6 kernel includes over 40 filesystems, about 0.6 million lines of code in total. The large number of Linux filesystems as well as their sheer diversity is a testament to the power and flexibility of the Linux VFS. This has enabled Linux to support a wide range of existing file system formats and protocols. Interestingly, this has also resulted in a growing number of new file systems that have been developed for Linux. The 2.5 development series saw the inclusion of four(ext3, reiserFS, JFS, and XFS) general purpose journalling filesystems, while in recent times multiple cluster filesystems have been submitted to the mainline kernel, providing users a slew of alternatives to choose from. Allowing multiple independent general purpose filesystems to co-exist has also had the positive effect of enabling each to innovate in parallel within its own space making different trade-offs and evolving across multiple production releases. New file systems have a chance to prove themselves out in the real world, letting time pick the best one rather than standardize on one single default filesystem[6].

At the same time, the multiplicity of filesystems that essentially address very similar needs also sometimes leads to unwarranted fragmentation and divergence from the perspective of users

who may find themselves faced with complex administrative choices with associated lock-in to a file system format chosen at a certain point in time. This is probably one reason why most users tend to simply adopt the default file system provided by their Linux distribution (i.e. ext3 or reiserfs), despite the availability of possibly more advanced filesystems like XFS and JFS, which might have been more suitable for their primary workloads. Each individual filesystem has been evolving to expand its capabilities by adding support for an increasing number of options and variations, to satisfy new requirements, and make continuous improvements while also maintaining compatibility.

We believe that these trends point to a need for a framework for a different kind of flexibility in file system design for the Linux kernel, one that allows continuous adaptability to evolving requirements, but reduces duplicate effort while providing users the most appropriate layout and capabilities for the workload and file access patterns they are running.

This work has two main goals: (1) to investigate how far a design centered on supporting dynamic customization of services can help to address Linux's needs for flexible file systems and (2) to explore performance benefits and trade-offs involved in a design for flexibility.

The basis of our exploration is the HFS/KFS research effort started a decade ago. In the Hurricane File System (HFS), the support for dynamic alternatives for file layout were motivated by the scalability requirements of the workloads targetted by the Hurricane operating system project. The K42 File System (KFS) built on the flexibility basis of HFS, expanding it by encorporating the architectural principles in the K42 Research Operating System project. While KFS was designed as a separate filesystem of its own, in this work we explore what it would take to apply similar techniques to existing filesystems, where such fine-grained flexibility was not an original design consideration. We also update KFS to work with Linux 2.6, aiming at carrying out experimental work that can provide concrete information about the impact of KFS's approach on addressing workload-specific performance requirements.

The rest of the paper is organized as follows: Section 2 illustrates how adaptability is currently held in Linux filesystems; Section 3 presents the basic ideas from KFS's design; Section 4 discusses KFS's potential as an adaptable filesystem framework for Linux and Section 5 describes required future work to enable this vision. Section 6 concludes.

2 Adaptability in Linux filesystems

2.1 Flexibility provided by the VFS layer

The Linux Virtual File System (VFS) [24] is quite powerful in terms of the flexibility it provides for implementing filesystems, whether disk-based filesystems, network filesystems or special purpose pseudo filesystems. This flexibility is achieved by abstracting key filesystem objects, i.e. the super block, inode and file (for both files and directories) including the address space mapping, the directory entry cache, and methods associated with each of these objects. It is possible to allow different inodes in the same filesystem to have different operation vectors. This is used, for example, by ext3 to support different journalling modes elegantly, by some types of stackable/filter filesystems to provide additional functionality to an existing filesystem, and even for specialized access modes determined by open mode, e.g. execute-in-place support. Additionally, the inclusion of extended attributes support in the VFS methods

allows customizable persistent per-inode state to be maintained, which can be used to specify variations in functional behavior at an individual file level.

The second aspect of flexibility ensues from common code provided for implementation of various file operations, i.e. generic routines which can be invoked by filesystems or wrapped with additional filesystem-specific code. The bulk of interfacing between the VFS and the Virtual Memory Manager (VMM), including read-ahead logic, page-caching and access to disk blocks for block-device-based filesystems, happens in this manner. Some of these helper routines (e.g. *mpage_writepages())* accept function pointers as arguments, e.g. for specifying a filesystem specific *getblock()* routine, which also allows for potential variation of block mapping and consistency and allocation policies even within the same filesystem.

Another category of helper routines called *libfs* [4] is intended for simplifying the task of writing new virtual filesystems, though currently targeted mainly at stand-alone virtual filesystems developed for special-purpose interfacing between kernel and user space, as an alternative to using *ioctls* or new system calls.

2.2 Flexibility within individual filesystems

Over time, the need to satisfy new requirements while maintaining on-disk compatibility to the extent possible has led individual filesystems to incorporate a certain degree of variability within each filesystem specific implementation, using mount options, fcntls/ioctls, file attributes and internal layering. In this sub-section we cover a few such examples. While we limit this discussion to a few disk-based general purpose filesystems, similar approaches apply to other filesystem types as well.

We do not presently consider file systems that are not intended to be part of the mainline Linux kernel tree.

2.2.1 Ext3 options and backward compatibility

One of the often cited strengths of the ext3 filesystem is its high emphasis on backwards and forwards compatibility and dependability, even as it continues evolving to incorporate new features and enhancements. This has been achieved through a carefully designed compatibility bitmap scheme [22], the use of mount options and *tune2fs* to turn on individual features per filesystem, conversion utilities to ease migration to new incompatible features (e.g. using resize2fs-type capability), and per-file flags and attributes that can be controlled through the *chattr* command.

Three compatibility bitmaps in the super block determine if and how an old kernel would mount a filesystem with an unknown feature bit marked in each of these bitmaps: read-write (COMPAT), read-only (RO_COMPAT), and incompatible (INCOMPAT)). For safety reasons, though, the filesystem checker *e2fsck* takes the stringent approach of not touching a filesystem with an unknown feature bit even if it is in the COMPAT set, recommending the usage of a newer version of the utility instead. Backward compatibility with an older kernel is useful during a safe revert of an installation/upgrade or in enabling the disk to be mounted from other Linux systems for emergency recovery purposes. For these reasons interesting techniques have been used in the development of features like directory indexing [16] making interior index nodes look like deleted directory entries and clearing directory-indexing flags when updating directories in older kernels to ensure compatibility as far as possible. Similar considerations are being debated during the

design of file pre-allocation support to avoid exposing uninitialized pre-allocated blocks if mounted by an older kernel. With the inclusion of per-inode compatibility flags, the granularity of backward compatibility support can be narrowed down to a per-file level. This may be useful during integration of extent maps.

The use of mount options and *tune2fs* makes it possible for new, relatively less established or incompatible features to be turned on optionally with explicit administrator knowledge for a few production releases before being made the default. Also, in the future, advanced feature sets may be bundled into a higher level group that signifies a generational advance of the filesystem [23]. Mount options are also used for setting consistency policies (i.e. journalling mode) on a per filesystem basis. Additionally, ext3 makes use of persistent file attributes (through the introduction of the *chattr* command), in combination with the ability to use different operation vectors for different inodes, to allow certain features/policies (e.g. full data journalling support for certain log files, preserving reservation beyond file close for slow-growing files) to be specified for individual files.

2.2.2 JFS and XFS

Although JFS [20] does not implement compatibility bitmaps as ext3 does, its on-disk layout is scalable, and backward compatibility has not been much of an issue. The on-disk directory structure was changed shortly after JFS was ported from OS/2® to Linux, causing a version bump in the super block. Since then, there has been no need to change the on-disk layout of JFS. The kernel will still support the older, OS/2-compatible, format.

JFS uses extent-based data structures and uses 40 bits to store block offsets and addresses. The on-disk layout supports various block sizes, although the kernel currently only supports a 4 KB block size. Without increasing the block size, JFS can support files and partitions up to 4 petabytes in length. B+ trees are used to implement both the extent maps and directories. Inodes are dynamically allocated as needed, and extents of free inodes are reclaimed. JFS can store file names in 16-bit unicode, translating from the code page specified by the iocharset mount option. The default is to do no translation.

JFS supports most mount options and chattr flags that ext3 does. Journaling can be temporarily disabled with the nointegrity mount flag. This is primarily intended to speed up the population of a partition, for instance, from backup media, where recovery would involve reformatting and restarting the process, with no risk of data loss.

XFS [21] also has a scalable ondisk layout, uses extent based structures, variable block sizes, dynamic inode allocation and separate allocation groups. It supports an optional real-time allocator suitable for media streaming applications.

2.2.3 Reiser4 plugins

Much like ext3 and JFS, the reiserfs v3 filesystem included in the current Linux kernel uses mount options and persistent inode attributes (set via chattr) to provide new features and variability of policies. For example, the hash function for directories can be selected by a mount option, as can some tuning of the block allocator, while tail-merging can be disabled on both a per mount point or per inode basis. It supports the same journalling modes as ext3.

The next generation of reiserfs, the reiser4 filesystem [15] (not yet in the mainline Linux

kernel), includes an internal layering architecture that is intended to allow for the development for different kinds of plugins to make extensions to the filesystem without having to upgrade/format to a new version. This approach enables the addition of new features like compression, support for alternate semantics, directory ordering, security schemes, block allocation and consistency policies, and node balancing policies for different items. The stated goal of this architecture is to enable and encourage developers to build plugins to customize the filesystem with features desired by applications. From the available material at this stage, it is not clear to us yet the extent to which plugins are intended to address fine-grain flexibility or dynamic changes of on-disk data representation beyond tail formatting and item merging in dancing trees. Also, at a first glance our (possibly mistaken) perception is that reiser4 flexibility support comes with the price of complexity: the code base is large, and the interfaces appear to be tied to reiser4 internals.

2.3 Limitations of current approaches

While there is a considerable amount of flexibility within the existing framework, both at the VFS level and within individual file systems, there are some observations and issues emerging with the evolution of multiple new filesystems and new generations of filesystems that have been in existence for a while.

- Code commonality is supported at higher levels but not for lower level data manipulation, e.g. with the inclusion of extents support for ext3 there would be over 5 B+ separate tree implementations across individual filesystems.
- While the combination of per-inode operation vectors and persistent attribute flags allows for flexibility at per inode level, and alternate allocation schemes can potentially be encapsulated in the *get_blocks()* function pointer used for a given operation, there is no general framework to support different layouts for different files in a cohesive manner, to move from an old scheme to a new one in a modular fashion, or supply different meta-data or data allocation policies for a group of files, because the inode representation is typically fixed upfront.
- There is no framework for filesystems to provide their building blocks for use by other filesystems, for example even though OCFS2 [5] chose to start with a lot of code from ext3 as its base, this involved copying source code and then editing it to add all the function it needed for clustering support and scalability. As a result, extensions like 64-bit and extents support from OCFS2 can not be applied back as extensions to ext3 as an option. Because of its long history of simplicity and dependability, ext2/3 is often the preferred choice for basing experimentation for advanced capabilities [9], so the ability to specialize behaviour starting from a common code base is likely to be useful.
- Difficulty with making changes to the on-disk format for an existing file system results in implementers getting locked into supporting a change once made, and hence requires very careful consideration and make take years into real deployment especially for incompatible features. Even compatible features on ext2/3 are incompatible with older filesystem checkers.

3 Overview of KFS

KFS builds on the research done by the Hurricane File System (HFS) [14, 13, 12]. HFS was designed for (potentially large-scale) shared-memory multiprocessors, based on the principle that, in order to maximize performance for applications with diverse requirements, a file system must support a wide variety of file structures, file system policies, and I/O interfaces. As an extreme example, HFS allows a file's structure to be optimized for concurrent random-access write-only operations by 10 threads, something no other file system can do. HFS explored its flexibility to achieve better performance and scalability. It proved that its flexibility came with little processing or I/O overhead. KFS took HFS's principles further by eliminating global data structures or policies. KFS runs as a file system for the K42 [10] and Linux operating systems.

The basic aspect of KFS's enablement of fine-grained customization is that each virtual or physical resource instance (e.g., a particular file, open file instance, block allocation map) is implemented by a different set of (C++) objects. The goal of this object-oriented design is to allow each file system element to have the logical and physical representation that better matches its size and access pattern characteristics. Each element in the system can be serviced by the object that best fits its requirements; if the requirements change, the component representing the element in KFS can be replaced accordingly. Applications can achieve better performance by using the services that match their access patterns, scalability, and synchronization requirements.

When a KFS file system is mounted, the blocks on disk corresponding to the superblock are read, and a *SuperBlock* object is instantiated to represent it. A *BlockMap* object is also instantiated to represent block allocation information. Another important object instantiated at file system creation time is the *RecordMap*, which keeps the association between file system elements, their object type, and their disk location. In many traditional Unix file systems, this association is fixed and implicit: every file or directory corresponds to an inode number; inode location and inode data representation is fixed a priori. Some file systems support dynamic block allocation for inodes and a set of alternative inode representations. In KFS, instead of trying to accommodate new possibilities for representation and dynamic policies incrementally, we take the riskier route of starting with a design intended to support change and diversity of representations. KFS explores the impact of an architecture centered on supporting the design and deployment of evolving alternative representations for file system resources. The goal is to learn how far this architecture can go in supporting flexibility, and what are the trade-offs involved in this approach.

An element (file or directory) in KFS is represented in memory by two objects: one providing a logical view of the object (called Logical Server Object, or LSO), and one encapsulating its persistent characteristics (Physical Server Object, or PSO).

Figure 1 portrays the scenario where three files are instantiated: a small file, a very large file, and a file where extended allocation is being used. These files are represented by a common logical object (LSO) and by PSO objects tailored for their specific characteristics: PSOsmall, PSOextent, PSOlarge. If the small file grows, the PSOsmall is replaced by the appropriate object (e.g., PSOlarge). The *RecordMap* object is updated in order to reflect the new object type and the (potential) new file location on disk.

KFS file systems may spawn multiple disks. Figure 2 pictures a scenario where file X is being replicated on the two available disks,

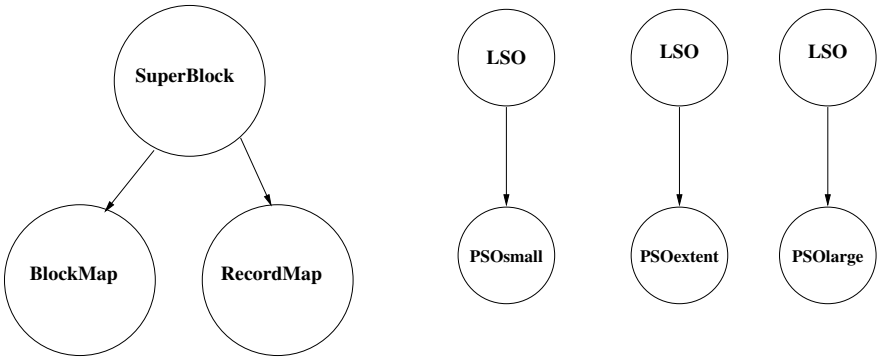

Figure 1: Objects representing files with different size and access pattern characteristics in KFS.

while file Y is being striped on the two disks in a round-robin fashion, and file Z is also being replicated, but with its content being compressed before going to disk.

In the current KFS implementation, when a file is created the choice of object implementation to be used is explicitly made by the file system code based on simplistic heuristics. Also, the user could specify intended behavior by changing values on the appropriate objects residing on /proc or use of extended attributes to provide hints about the data elements they are creating or manipulating.

As the file system evolves, compatibility with "older" formats is kept as long as the file system knows how to instantiate the object type to handle the old representation.

The performance experiments with KFS for Linux 2.4 indicate that KFS's support for flexibility doesn't result in unreasonable overheads. KFS on Linux 2.4 (with an implementation of inode and directory structures matching ext2) was found to run with a performance similar to ext2 on many workloads and 15% slower on some. These results are described in [18]. New performance data is being made available at [11] as we tune KFS's integration with Linux 2.6.

There are two ongoing efforts on using KFS's flexible design to quickly prototype and evaluate opportunities for alternative policies and data representation:

- Co-location of meta-data and data: a prototype of directory structure for embedding file attributes in directory entries where we extend the ideas proposed in [8] by doing meta-data and block allocation on a per-directory basis;

- Local vs global RecordMap structure: although KFS's design tried to avoid the use of global data structures, its initial prototype implementation has a single object (one instance of the *RecordMap* class) responsible for mapping elements onto type and disk location. As our experimentation progressed, it became clear that this data structure was hindering scalability and flexibility, and imposing performance overhead due to lack of locality between the RecordMap entry for a file and its data. Our current design explores associating different RecordMap objects with different parts of the name space.

More information about KFS can be found at [18, 7].

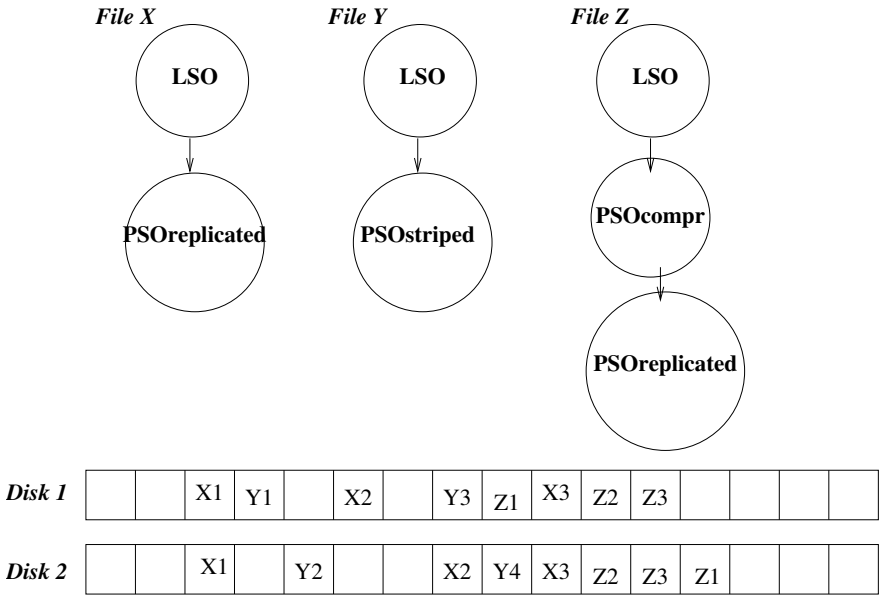

Figure 2: KFS objects and block allocation for files X (replicated; blocks X1, X2, X3), Y (striped; blocks Y1, Y2, Y3, Y4), and Z (compressed and replicated; blocks Z1, Z2, Z3).

4 Learnings from KFS towards an adaptable filesystem framework for Linux

What possibilities does the experience with KFS suggest towards addressing some of the concerns discussed in section 2.3? Intuitively, it appears that the ideas originating from HFS, and developed further in KFS, of a bottom-up building-block-based, fine-grained approach to flexibility for experimenting with specialized behaviour and data organization on a per-element basis, could elegantly complement the current VFS approach of abstracting higher levels of commonality that allows an abundance of filesystems. While KFS was developed as a separate filesystem, we do not believe that adding yet another filesystem to Linux is the right answer. Instead developing simple methods of applying KFS-like flexibility to existing filesystems incrementally, while possibly non-trivial, may be a promising approach to pursue.

4.1 Switching formats

While many of the file systems mentioned in Section 2 (ext2/3, reiserfs, JFS, XFS, OCFS2) are intended to be general-purpose file systems, each appears to have its own sweet-spot usage scenarios or access patterns that sets it apart from the rest. Various comparison charts [1, 17, 19] exist that highlight the strengths and weaknesses of these alternatives to enable administrators to make the right choice for their systems when they format their disks at the time of first installation. However, predicting the nature of workloads ahead of time, especially when mixed access patterns may apply to different files in the same filesystem at different times, is difficult. Unlike configuration options or run-time tunables, switching a choice of on-disk format on a system is highly disruptive and resoure consuming. With an adaptable framework that can alter representations on a per-element basis in response to significant changes in access patterns, living with a less than optimal solution as a result of being locked into a

given on-disk filesystem representation would no longer be necessary.

We have described earlier (section 3) that in KFS it is possible to create new formats and work with them, with other formats being simultaneously active as well. This is possible due to (1) the association between a Logical Storage Object (LSO) and its associated Physical Storage Object (PSO) is not fixed, and (2) the implementation of local rather than global control structures for different types of resource allocation. We plan to experiment with abstracting this mechanism so that it can be used by existing filesystems, for example for upgrading to new file representations like 64-bit and extents support in ext3 including the new multi-block allocator implementation from Alex Tomas [2]. We would like to compare the results with the current approach in ext3 for achieving the format switch, which relies on per-inode flags and alternate inode operation vectors.

4.2 Evaluation of alternate layouts and policies

The ability to switch formats and layout policies enables KFS to provide a platform for determining optimal layout and allocation policies for a filesystem through ongoing experimentation and comparative evaluation. Typically, layout decisions are very closely linked to the particular context in which the file system is intended to be used, e.g. considering underlying media, nature of application workloads, data access patterns, available space, filesystem aging/fragmentation, etc. With the mechanisms described in the previous sub-section in place, we intend to demonstrate performance advantages over the long run from being able to choose and switch from alternative representations, for example from a very small file oriented representation with data embedded within the inode, to direct, indirect block mapping, to extents maps based on file size, distribution and access patterns.

4.3 Assessment of overheads imposed by flexibility

It is said that any problem in computer science can be solved by adding an extra level of indirection. The only problem is that indirections do not come for free, especially if it involves extra disk seeks to perform an operation. It is for this reason that ext2/3, for example, attempts to allocate indirect blocks contiguously with the data blocks they map to, and why support for extended attributes storage in large inodes has been demonstrated to deliver significant performance gains for Samba workloads [3] compared to storage of attributes in a separate block. This issue has been a primary design consideration for KFS. The original HFS/KFS effort has been specifically conceptualized with a view towards enabling high performance and scalability through fine-grained flexibility, rather than with an intent of adding layers of high level semantic features.

Initial experiments with KFS seem to indicate that the benefits of flexibility outweigh overheads, given a well-designed meta-data caching mechanism and right choice of building blocks where indirections go straight to the most appropriate on-disk objects when a file element is first looked up. The ability to have entirely different and variable-sized “inode” representations for different types of files amortizes the cost across all subsequent operations on a file. It remains to be verified whether this is proved to be valid on a broad set of workloads and whether the same argument would apply in the context of adding flexibility to existing filesystems without requiring invasive changes.

Would the reliance on C++ in KFS be concern for application to existing Linux filesystems?

Inheritance has proven to be very useful during the development of per-element specialization in KFS, as it simplified coding to a great extent and enabled workload-specific optimizations. However, being able to move to C with a minimalist scheme may be a desirable goal when working with existing filesystems in the linux kernel.

4.4 Ability to drop experimental layout changes easily

As described in section 2.3, the problem of being stuck with on-disk format changes once made necessitates a stringent approach towards adoption of layout improvements which may result in years of lead time into actual deployment of state-of-the-art filesystem enhancements.

In KFS, as we add new formats, we can still work with old ones for compatibility, but the old ones can be kept out of the mainstream implementation. The code is activated when reading old representations, and we can on the fly "migrate" to the new format as we access it, albeit at a certain run-time overhead.

This does not however address the issue of handling backward compatibility with older kernels. Perhaps it would make sense to include a compatibility information scheme at a per-PSO level, similar to the ext2/3 filesystem's superblock level compatibility bitmaps. For example, in the situation where we are able to extend a given PSO type to a new scheme in a way that is backward compatible with the earlier PSO type (e.g. as in case of the directory indexing feature), we would like to indicate that so that an older kernel does not reject it.

5 Future work

Section 4 has discussed ongoing work on KFS that, in the short term, may result in useful insights for achieving an adaptable filesystem framework for Linux. In this section we discuss new work to be done that is essential to realizing this adaptable framework.

5.1 Adaptability in the filesystem checker and tools

With adaptable building blocks and support for multiple alternate formats as part of a per element specialization, it follows that file-system checker changes would be required to be able to recognize, load, and handle new PSOs as well as perform some level of general consistency checks based on the indication of location and sizes common to all elements, and parsing the record map(s). As with existing filesystem checkers, backward compatibility remains a tricky issue. The PSO compatibility scheme discussed earlier could be used to flag situations where newer versions of tools are required. Likewise, other related utilities like low-level backup utilities (e.g dump), migration tools, defragmenter, debugfs, etc would need to be dynamically extendable to handle new formats.

5.2 Address the problem of sharing granular building blocks across filesystems

KFS was not originally designed with the intent of enabling building-block sharing across existing file systems. However, given potential benefits in factoring core ext2/3 code, data-consistency schemes and additional capabilities (e.g. clustering), as well as extensions to libfs beyond its current limited scope of application, it is natural to ask whether KFS-type constructs could be useful in this context.

Could the same principles that allow per element flexibility through building block composition be taken further to enable abstraction of these building blocks in a way that not tightly tied to the containing filesystem? Design issues that may need to be explored in order to evaluate the viability of this possibility include figuring out how a combination of PSOs, e.g. alternate layouts and alternate consistency, schemes could be built efficiently in the context of an existing filesystem in a manner that is reusable in the context of another filesystem. The effort involved in trying to refactor existing code into PSOs may not be small; a better approach may be to start this with a few simple building blocks and use the framework for new building blocks created from here on.

5.3 Additional Issues

Another aspect that needs further exploration is whether the inclusion of building blocks and associated specialization adds to overall testing complexity for distributions, or if the framework can be enhanced to enable a systematic approach to be devised to simplify such verification.

6 Conclusions

We presented KFS and discussed that the experience so far indicates that KFS can be a powerful approach to support flexibility of services in a file system down to a per-element granularity. We also argued that the approach does not come with unacceptable performance overheads, and that it allows for workload-specific optimizations. We believe that the flexibility in KFS makes it a good prototype environment for experimenting with new file system resource management policies or file representations. As new emerging workloads appear, KFS can be useful to the Linux community by providing evaluation results for alternative representations and by advancing the application of different data layouts for different files within the same filesystem, determined either statically or dynamically in response to changing access patterns.

In this paper we propose a new exploration of KFS: to investigate how its building-block approach could be abstracted from KFS's implementation to allow code sharing among file systems, providing a library-like collection of alternative implementations to be experimented with across file systems. We do not have yet evidence that this approach is feasible, but as we improve the integration of KFS (originally designed for the K42 operating system) with Linux 2.6 we hope to have a better understanding of KFS's general applicability.

Acknowledgements

We would like to thank Dave Kleikamp for contributing the section on JFS flexibility, and Christoph Hellwig for his inputs on XFS.

Orran Krieger started KFS based on his experience with designing and implementing HFS. He has been a steady source of guidance and motivation for KFS developers, always available for in-depth design reviews that improved KFS and made our work more fun. We thank him for his insight and support.

We would like to thank Paul McKenney, Val Henson, Mingming Cao, and Chris Mason for their valuable review feedback on early drafts of this paper, and H. Peter Anvin and Mel Gorman for helping us improve our proposal. This paper might never have been written but for insightful discussions with Stephen Tweedie, Theodore T'so, and Andreas Dilger over the

course of the last year. We would like to thank them and other ext3 filesystem developers for their steady focus on continuously improving the ext3 filesystem, devising interesting techniques to address conflicting goals of dependability vs state-of-the-art advancement. Finally we would like to thank the linux-fsdevel community for numerous discussions over years which motivated this paper.

This work was partially supported by DARPA under contract NBCH30390004.

Availability

KFS is released as open source as part of the K42 system, available from a public CVS repository; for details refer to the K42 web site: `http://www.research.ibm.com/K42/`.

Legal Statement

References

[1] Ray Bryant, Ruth Forester, and John Hawkes. Filesystem performance and scalability in linux 2.4.17. In *USENIX Annual Technical Conference*, 2002.

[2] M. Cao, T.Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium(OLS)*, pages 69–96, 2005.

[3] Jonathan Corbet. Which filesystem for samba4? `http://lwn.net/Articles/112566/`.

[4] Jonathan Corbet. Creating linux virtual file systems. `http://lwn.net/Articles/57369/`, November 2003.

[5] Jonathan Corbet. The OCFS2 filesystem. `http://lwn.net/Articles/137278/`, May 2005.

[6] Alan Cox. Posting on linux-fsdevel. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=112558745427067&w=2`, September 2005.

[7] Dilma da Silva, Livio Soares, and Orran Krieger. KFS: Exploring flexilibity in file system design. In *Proc. of the Brazilian Workshop in Operating Systems*, Salvador, Brazil, August 2004.

[8] Greg Ganger and Frans Kaashoek. Embedded inodes and explicit gruopings: Exploiting disk bandwith for small files. In *Proceedings of the 1997 Usenix Annual Technical Conference*, pages 1–17, January 1997.

[9] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 filesystems. In *Proceedings of the Ottawa Linux Symposium(OLS)*, 2006.

[10] The K42 operating system, `http://www.research.ibm.com/K42/`.

[11] Kfs performance experiments. `http://k42.ozlabs.org/Wiki/KfsExperiments`, 2006.

[12] O. Krieger and M. Stumm. HFS: A performance-oriented flexible filesystem based on build-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, 1997.

[13] Orran Krieger. *HFS: A Flexible File System for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

[14] Orran Krieger and Michael Stumm. HFS: A flexible file system for large-scale multiprocessors. In *Proceedings of the DAGS/PC Symposium (The Second Annual Dartmouth Institute on Advanced Graduate Studies in Parallel Computation)*, 1993.

[15] Namesys. Reiser4. `http://www.namesys.com/v4/v4.html`, August 2004.

[16] Daniel Phillips. A directory index for ext2. In *5th Annual Linux Showcase and Conference*, pages 173–182, 2001.

[17] Justin Piszcz. Benchmarking File Systems Part II.

[18] Livio Soares, Orran Krieger, and Dilma Da Silva. Meta-data snapshotting: A simple mechanism for file system consistency. In *SNAPI'03 (International Workshop on Storage Network Architecture and Parallel I/O)*, pages 41–52, 2003.

[19] John Troy Stepan. Linux File Systems Comparative Performance. *Linux Gazette*, January 2006. `http://linuxgazette.net/122/TWDT.html`.

[20] IBM JFS Core Team.

[21] SGI XFS Team. XFS: a high-performance journaling filesystem.

[22] Stephen Tweedie and Theodore Y Ts'o. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference*, pages 235–244, 2002.

[23] Stephen C Tweedie. Re: Rfc: mke2fs with dir_index, resize_inode by default, March, 2006.

[24] Linux kernel sourcecode vfs documentation. file `Documentaion/filesystems/vfs.txt`.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin