Spiffy: Enabling File-System Aware Storage Applications

Kuei Sun, Daniel Fryer, Joseph Chu, Matthew Lakier, Angela Demke Brown and Ashvin Goel
University of Toronto

Abstract

Many file-system applications such as defragmentation tools, file system checkers or data recovery tools, operate at the storage layer. Today, developers of these storage applications require detailed knowledge of the file-system format, which takes a significant amount of time to learn, often by trial and error, due to insufficient documentation or specification of the format. Furthermore, these applications perform ad-hoc processing of the file-system metadata, leading to bugs and vulnerabilities.

We propose Spiffy, an annotation language for specifying the on-disk format of a file system. File-system developers annotate the data structures of a file system, and we use these annotations to generate a library that allows identifying, parsing and traversing file-system metadata, providing support for both offline and online storage applications. This approach simplifies the development of storage applications that work across different file systems because it reduces the amount of file-system specific code that needs to be written.

We have written annotations for the Linux Ext4, Btrfs and F2FS file systems, and developed several applications for these file systems, including a type-specific metadata corruptor, a file system converter, and an online storage layer cache that preferentially caches files for certain users. Our experiments show that applications that use the library to access file system metadata can achieve good performance and are robust against file system corruption errors.

1 Introduction

There are many file-system aware storage applications that bypass the virtual file system interface and operate directly on the file system image. These applications require a detailed understanding of the format of a file system, including the ability to identify, parse and traverse file system structures. These applications can operate in an offline or online context, as shown in Table 1. Examples of offline tools include a file system checker that traverses the file system image to check the consistency of its metadata [17], and a data recovery tool that helps recover deleted files [4].

Online storage applications need to understand the file-system semantics of blocks as they are accessed at runtime (e.g., whether the block contains data or metadata, whether it belongs to a specific type of file, etc.).

Storage Applications	Category	Purpose
Differentiated services [18]	online	performance
Defragmentation tool	either	
File system checker [13]	either	reliability
Data recovery tool [4]	offline	
IO shepherding [12]	online	
Runtime verification [8]	online	
File system conversion tool	offline	administrative
Partition editor [11]	offline	
Type-specific corruption [2]	offline	debugging
Metadata dump tool	offline	

Table 1: Example file-system aware storage applications. Offline applications have exclusive access to the file system; online applications operate on an in-use file system.

These applications improve the performance or reliability of a storage system by performing file-system specific processing at the storage layer. For example, differentiated storage services [18] improve performance by preferentially caching blocks that contain file-system metadata or the data of small files. I/O shepherding [12] improves reliability by using file structure information to implement checksumming and replication. Similarly, Recon [8] improves reliability by verifying the consistency of file-system metadata at the storage layer.

Today, developers of these storage applications perform ad-hoc processing of file system metadata because most file systems do not provide the requisite library code. Even when such library code exists, its interface may not be usable by all storage applications. For example, the `libext2fs` library only supports offline interpretation of a Linux Ext3/4 file system partition; it does not support online use. Furthermore, the libraries of different file systems, even when they exist, do not provide similar interfaces. As a result, these storage applications have to be developed from scratch, or significantly rewritten for each file system, impeding the adoption of new file systems or new file-system functionality.

To make matters worse, many file systems do not provide detailed and up-to-date documentation of their metadata format. The ad-hoc processing performed by these storage applications is thus error-prone and can lead to system instability, security vulnerability, and data corruption [3]. For example, `fsck` can sometimes further corrupt a file system [33]. Some storage applications reduce the amount of file-system specific code in their im-

plementation by modifying their target file system and operating system [18, 12]. This approach only works for specific file systems, and can introduce its own bugs. It also requires custom system software, which may be impractical in virtual machine and cloud environments.

Our aim is to reduce the burden of developing file-system aware storage applications. To do so, we enable file system developers to specify the format of their file system using a domain-specific language so that the file system metadata can be parsed, traversed and updated correctly. We introduce Spiffy,[1] a language for annotating file system data structures defined in the C language. Spiffy allows file system developers to unambiguously specify the *physical* layout of the file system. The annotations handle low level details such as the encoding of specific fields, and the pointer relationships between file system structures. We compile the annotated sources to generate a Spiffy library that provides interfaces for type-safe parsing, traversal and update of file system metadata. The library allows a developer to write actions for different file system metadata structures, invoking file-system specific or generic code as needed, for their offline or online application. We support online applications that need to read metadata, such as differentiated storage services [18], but not ones that need to modify metadata such as online defragmentation.

The generic interfaces provided by the library simplify the development of applications that work across different file systems. Consider an application that shows file-system fragmentation by plotting a histogram of the size of free extents in the file system. This application needs to traverse the file system to find and parse structures that represent free space, and then collect the extent information. With Spiffy, the application code for finding and parsing structures is similar for different file systems. File-system specific actions are only needed for collecting the extent information from the free space structures (e.g., bitmaps for Ext4 and free space extents for Btrfs).

The complexity of modern file systems [16] raises several challenges for our specification-based approach. Many aspects of file system structures and their relationships are not captured by their declarations in header files. First, an on-disk pointer in a file-system structure may be implicitly specified, e.g., as an integer, as shown below. The naming convention suggests that this field is a pointer, but that fact cannot be deduced from the structure definition because it is embedded in file system code.

```
struct foo {
  __le32 bar_block_ptr;
};
```

Second, the interpretation of file system structures can depend on other structures. For example, the size of an inode structure in a Linux Ext3/4 file system is stored in a field within the super block that must be accessed to correctly interpret an inode block. Similarly, many structures are variable sized, with the size information being stored in other structures. Third, the semantics of metadata fields may be context-sensitive. For example, pointers inside an inode structure can refer to either directory blocks or data blocks, depending on the type of the inode. Fourth, the placement of structures on disk may be implicit in the code that operates on them (e.g., an instance of structure B optionally follows structure A) and some structures may not be declared at all (e.g., treating a buffer as an array of integers). Finally, metadata interpretation must be performed efficiently, but it is impractical to load all file-system metadata into memory for large file systems. These challenges are not addressed by existing specification tools, as discussed in Section 7.

In Spiffy, the key to specifying the relationships between file system structures is a pointer annotation that specifies that a field holds an address to a data structure on physical storage. Pointers have an address space type that indicates how the address should be mapped to the physical location. In the `struct foo` example above, this annotation would help clarify that `bar_block_ptr` holds an address to a structure of type `bar`, and its address space type is a (little-endian) block pointer. We expose cross-structure dependencies by using a name resolution mechanism that allows annotations to name the necessary structures unambiguously. We handle context-sensitive fields and structures by providing support for conditional types and conditionally inherited structures. We also provide support for specifying implicit fields that are computed at runtime. Last, annotations can specify the granularity at which the structures should be accessed from storage, allowing efficient data access and reducing the memory footprint of the applications.

Together, these Spiffy features have allowed us to properly annotate three widely deployed file systems, 1) Ext4, an update-in-place file system, 2) Btrfs, a copy-on-write file system, and 3) F2FS, a log-structured file system [15]. We have implemented five applications that are designed to work across file systems: a file system dump tool, a file system corruption tool, a free space display tool, a file system converter, and a storage layer service that preferentially caches data for specific users.

2 Bugs in File-System Applications

We motivate this work by presenting various bugs caused by incorrect parsing of file-system metadata in storage applications (outlined in Table 2). Some of these bugs cause crashes, while others may result in file system corruption. For each bug, we discuss the root cause.

1. An extra memory allocation caused uninitialized bytes

[1] **Sp**ecifying and **I**nterpreting the **F**ormat of **F**iles**y**stems

	Tool	FS	Bug Title	Closed
1	libparted	Fat32	#22266: jump instruction and boot code corrupted with random bytes after fat is resized	2016-05
2	ntfsprogs	NTFS	Bug 723343 - Negative Number of Free Clusters in NTFS Not Properly Interpreted	2014-02
3	e2fsck	Ext4	#781110 e2fsprogs: e2fsck does not detect corruption	2016-05
4	e2fsck	Ext4	#760275 e2fsprogs: e2fsck corrupts Hurd filesystems	2015-05
5	btrfsck	Btrfs	Bug 104141 - Malformed input causing crash / floating point exception in btrfsck	2015-10

Table 2: Bugs due to incorrect parsing of file system formats.

to be written to the boot jump field of Fat32 file systems during resizing. Since Windows depends on the correctness of this field, the bug rendered the file system unrecognizable by the operating system.

2. NTFS has a complex specification for the size of the MFT record. If the value is positive, it is interpreted as the number of clusters per record. Otherwise, the size of the record is $2^{|value|}$ bytes (e.g., -10 would mean that the record size is 1024 bytes). The developers of ntfsprogs were unaware of this detail, and so the GParted partition editing tool would fail when attempting to resize an NTFS partition.
3. The e2fsck file system checker failed to detect corrupted directory entries if the size field of the entries was set to zero, which resulted in no repair being performed. Ironically, other programs, such as debugfs, ls, and the file system itself, could correctly detect the corruption.
4. Ext2/3/4 inodes contain union fields for storing operating system (OS) specific metadata. A sanity check was omitted in e2fsck prior to accessing this field, and repairs were always performed assuming that the creator OS is Linux. Consequently, the file system becomes corrupt for Hurd and possibly other OSs.
5. A fuzzer [34] was able to craft corrupted super blocks that would crash the Btrfsck tool. In response, Btrfs developers added 15 extra checks (for a total of 17 checks) to the super block parsing code.

The common theme among all these bugs is that: 1) they are simple errors that occur because they require a detailed understanding of the file system format; 2) they can cause serious data loss or corruption; and 3) most of these bugs were fixed in less than 5 lines of code. Our domain-specific language allows generating libraries that can sanitize file system metadata by checking various structural constraints before it is accessed in memory. In the presence of corrupted metadata, our libraries generate error codes, rather than crashing the tools or propagating the corruption further. Section 3.1 discusses how our approach can help prevent or detect these bugs.

3 Approach

Our annotation language enables type-safe interpretation of file system structures, in both offline and online contexts. Type safety ensures that parsing and serialization of file system structures will detect data corruption that leads to type violations, thus reducing the chance of corruption propagation, and avoiding crash failures.

Ideally, data structure types and their relationships could be extracted from file system source code. Although the C header files of a file system contain the structural definitions for various metadata types, they are incomplete descriptions of the file system format because information is often hidden within the file system code. Our annotations augment the C language, helping specify parts of a file system's format that cannot be easily expressed in C.

After a file system developer annotates his or her file system's data structures, we use a compiler to parse the annotated structures and to generate a library that provides file-system specific interpretation routines. The library supports traversal and selective retrieval of metadata structures through type introspection. These facilities allow writing generic or file-system specific actions on specific file system metadata structures. For example, the application may wish to operate on the directory entries of a file system. Instead of attempting to parse the entire file system and find all directory entries, which requires significant file-system specific code, a developer using Spiffy would use generic type introspection code to find and operate on all directory entries. However, since the directory entry format may not be the same across file systems, the application may require file-system specific actions on the directory entry structures.

Our annotation-based approach has several advantages. First, it provides a concise and clear documentation of the file system's format. Second, our generated libraries enable rapid prototyping of file-system aware storage applications. The libraries provide a uniform API, easing the development of applications that work across file systems so that the programmer can focus on the logic and not the format of the file systems. Third, our approach requires minimal changes to the file system source code (the annotations are only in the C header files and are backwards compatible with existing binary code), reducing the chance of introducing file system bugs. In contrast, differentiated storage services [18] needed to modify the file system and the kernel's storage stack to enable I/O classification. With our approach, this application can be implemented by using introspec-

```
struct ext4_dir_entry {
  __le32 inode;             /* Inode number */
  __le16 rec_len; /* Directory entry length */
  __u16  name_len;           /* Name length */
  char   name[EXT4_NAME_LEN];  /* File name */
};
```

Figure 1: Ext4 directory entry structure definition.

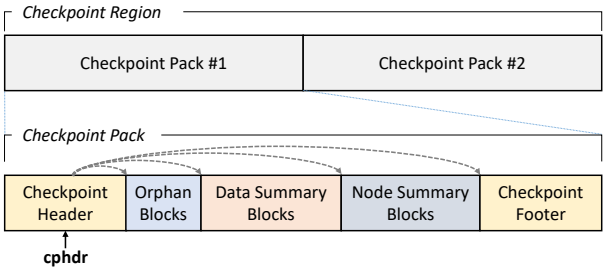

Figure 2: Each F2FS checkpoint pack contains a header followed by a variable number of orphan blocks.

tion at the block layer for an unmodified file system, or at the hypervisor for an existing virtual machine. Finally, file system formats are known to be stable over time, so there is minimal cost for maintaining annotations.

3.1 Designing Annotations

The design of our annotation language for specifying the format of file system structures was motivated by several key concepts.

File System Pointers File system pointers connect the metadata structures in a file system, but they are not well specified in C data structure definitions, as explained in Section 1. The difference between a file system pointer and an in-memory pointer is that the content of an in-memory pointer is always interpreted as the in-memory address of the pointed-to data, but interpreting the address contained by a file system pointer may involve multiple layers of translation. The most common type of file system pointer is a block pointer, where the address maps to a physical block location that contains a contiguous data structure. However, file system structures may also be laid out discontiguously. For example, the journal of an Ext4 file system is a logically contiguous structure that can be stored on disk non-contiguously, as a file. Similarly, Btrfs maps logical addresses to physical addresses for supporting RAID configurations.

Our design incorporates this requirement by associating an *address space* with each file system pointer. Each address space specifies a mapping of its addresses to physical locations. In the case of the Ext4 journal, we use the inode number, which uniquely identifies files in Unix file systems, as an address in the file address space.

Cross-Structure Dependencies File system structures often depend on other structures. For example, the length of a directory entry's `name` in Ext4 is stored in a field called `name_len`, as shown in Figure 1. However, this data structure definition does not provide the linkage between the two fields.[2] Structures may depend on fields in other structures as well. For example, several fields of the super block are frequently accessed to determine the block size, the features that are enabled in the file system, etc. To support these dependencies, we need to name these structures. For example, the expression `sb.s_inode_size` helps determine the size of an inode object, where `sb` is the name assigned to the super block.

The naming mechanism must ensure that a name refers to the correct structure. For example, the F2FS file system contains two checkpoint packs for ensuring file system consistency, as shown in Figure 2. The number of orphan blocks in a F2FS checkpoint pack is determined by a field inside the checkpoint header. Our naming mechanism must ensure that when this field is accessed, it refers to the header structure in the correct checkpoint pack.

Spiffy uses a path-based name resolution mechanism, based on the observation that every file system structure is accessed along a path of pointers starting from the super block. In the simplest case, the automatic `self` variable is used to reference the fields of the same structure. Otherwise, a name lookup is performed in the reverse order of the path that was used to access the data structure. For example, in Figure 2, when we need to reference the checkpoint header (`cphdr` in the figure) while parsing the orphan block, the name resolution mechanism can unambiguously determine that it is referring to its parent checkpoint header. This strategy also makes it easy to use reference counting to ensure that a referenced structure is valid in memory when it needs to be accessed.

Context-Sensitive Types File system metadata are frequently context-sensitive. A pointer may reference different types of metadata, or a structure may have optional fields, based on a field value. For example, the type of a journal block in Ext4 depends on a common field called `h_blocktype`. If the field's value is 3, then it is the journal super block that contains many additional fields that can be parsed. However, if its value is 2, then it is a commit block that contains no other fields. We need to be able to handle such context-sensitive structures and pointers. We use a *when* expression, evaluated at runtime, to support such context-sensitive types. These conditional expressions also allow us to specify when different fields of a union are valid, which enables Spiffy to enforce a strict access discipline at runtime, and would prevent Bug #4 from Section 2.

Computed Fields Sometimes file systems compute a value from one or more fields and use it to locate structures. For example, the block group descriptor table in

[2] Confusingly, `name` has a fixed size in the definition.

Base Class	Member Function	Description
Spiffy File System Library		
Entity	`int process_fields(Visitor & v)`	allows *v* to visit all fields of this object
	`int process_pointers(Visitor & v)`	allows *v* to visit all pointer fields of this object
	`int process_by_type(int t, Visitor & v)`	allows *v* to visit all structures of type *t*
Pointer	`Entity * fetch()`	retrieves the pointed-to container from disk
Container	`int save(bool alloc=true)`	serializes and then persists the container, may assign a new address to the container
FileSystem	`FileSystem(IO & io)`	instantiates a new file system object
	`Entity * fetch_super()`	retrieves the super block from disk
	`Entity * create_container(int type, Path & p)`	creates a new container of metadata *type*
	`Entity * parse_by_type(int type, Path & p, Address & addr, const char * buf, size_t len)`	parses the buffer as metadata *type*, using *p* to resolve cross structure dependencies
File System Developer		
IO	`int read(Address & addr, char * & buf)`	reads from an address space specified by *addr*
	`int write(Address & addr, const char * buf)`	writes to an address space specified by *addr*
	`int alloc(Address & addr, int type)`	allocates an on-disk address for metadata *type*
Application Programmer		
Visitor	`int visit(Entity * e)`	visits an entity and possibly processes it

Table 3: Spiffy C++ Library API.

Ext4 is implicitly the block(s) that immediately follows the super block. However, the exact address of the descriptor blocks depends on the block size, which is specified in the super block. We annotate this information as an implicit field of the super block that is computed at runtime. This approach allows the field to be dereferenced like a normal pointer, allowing traversal of the file system without requiring any changes to the underlying format. A computed field annotation can also be used to specify the size calculation for an NTFS MFT record, avoiding Bug #2 from Section 2.

Metadata Granularity Existing file systems assume that the underlying storage media is a block device and access data in block units. Data structures can exist within such blocks or they can span contiguous physical blocks. Some data structures that span blocks are read in their entirety. For example, the Btrfs B-tree nodes are (by default) 16KB, or 4 blocks, and these blocks are read from disk together. In other cases, the data structure is read in portions. For example, an Ext4 inode table contains a group of inode blocks. The file system does not load the entire table in memory because it can be very large. Instead, it only loads the portions that are needed.

We define an *access unit* for file system structures so that the compiler can generate efficient code for traversing the file system. We call the unit of disk access a *container*. The container size is typically the file system block size but it may span multiple blocks, as in the Btrfs example. A structure that is placed inside a container is called an *object*. Finally, structures that span containers are called *extents*. We load extents on demand, when their containers are accessed.

Constraint Checking The values of metadata fields within or across different objects often have constraints. For example, an Ext4 extent header always begins with the magic number `0xF30A` to help detect corrupt blocks. Similarly, the `name_len` field of an Ext4 directory entry should be less than the `rec_len` field. Such constraints can be specified for each structure so that they can be checked to ensure correctness when parsing the structure. The use of constraint annotations could have helped prevent Bug #1, and detect Bugs #3 and #5 from Section 2.

The set of valid addresses for a metadata container may also have a *placement constraint*. For example, F2FS NAT blocks can only be placed inside the NAT area, which is specified in the F2FS super block. By annotating the placement constraint of a metadata container, Spiffy can verify that the address assigned to newly allocated metadata is within the correct bounds before the metadata is persisted to disk.

3.2 The Spiffy API

Table 3 shows a subset of the API for building Spiffy applications. The API consists of three sets of functions. The first set are automatically generated by Spiffy based on the annotated file system data structures. The second set need to be implemented by file system developers and are reusable across different applications. The last set are written by the application programmer for implementing application and file-system specific logic.

The Spiffy library uses the visitor pattern [9], allowing a programmer to customize the operations performed on each file system metadata type by implementing the `visit` function of the abstract base class `Visitor`.

The `Entity` base class provides a common interface for all metadata structures and their fields. The `process_pointers` function invokes the `visit` function of an application-defined Visitor class on each

```
struct Address {
  int      aspc;   /* address space type */
  long     id;     /* id of the address */
  unsigned offset; /* offset from id */
  unsigned size;   /* size of object */
};
```

Figure 3: Address structure to locate container on disk.

pointer within the entity. The `process_by_type` function allows visiting a specific type of structure that is reachable from the entity. Unlike the other process functions, `process_by_type` will automatically follow pointers. For example, invoking `process_by_type` on the super block with the inode structure as an argument results in visiting all inodes in the file system.

Every container (and extent) has an address associated with it that allows accessing the container from disk. Figure 3 shows the format of an address, consisting of an address space, an identifier and an offset within the address space, and the size of the container. The offset field is used when a container belongs to an extent.

The `Pointer` class stores the address of a container (or an extent), and its `fetch` function reads the pointed-to container from disk. Figure 4 shows the generated code for the `fetch` function for a pointer to a container named `IBlock` (inode block). The file-system developer implements an `IO` class with a `read` function for each address space defined for the file system. When the `IBlock` is constructed, it invokes the constructors of its fields, thus creating all the objects (e.g., inodes) within the container. The constructors for inodes, in turn, invoke the constructors of block pointers in the inodes, which initialize a part of the address (address space, size and offset) of the block pointers based on the annotations. Then the container is parsed, which initializes the container fields in a nested manner, including setting the `id` component of the address of all the block pointers in the inodes contained in the `IBlock`.

The `Path` object is associated with every entity and contains the list of structures that are needed to resolve cross-structure dependencies during parsing or serializing the container. It is set up based on the sequence of constructor calls, with each constructor adding the current object to the path passed to it.

The `save` function serializes a container by invoking nested serialization on its fields. Then, it invokes the `alloc` function for newly created metadata, or when existing metadata has to be reallocated (e.g., copy-on-write allocator). The allocator finds a new address for the container and updates any metadata that tracks allocation (e.g., the Ext4 block bitmap). If the address passes placement constraint checks, the buffer is written to disk.

The `create_container` function constructs empty containers of a given type. The application developer can then fill the container with data and invoke `save` to allocate and write the newly created container to disk.

```
Entity * IBlockPtr::fetch() {
  IBlock * ib;
  Address & addr = this->address;
  char * buf = new char[addr.size];
  this->fs.io.read(addr, buf);
  ib = new IBlock(this->fs, addr, this->path);
  ib->parse(buf, addr.size);
  return ib;
}
```

Figure 4: Example of a generated `fetch` function. `IBlockPtr` is a subclass of `Pointer`.

3.3 Building Applications

Figure 5 shows a sample application built using the Spiffy API. This application prints the type of each metadata block in an Ext4 file system in depth-first order. The `Ext4IO` class implements the block and the file address space, as described in Section 5. The program starts by invoking `fetch_super`, which fetches the super block from a known location on disk and parses it. Then it uses two mutually recursive visitors, `EntVisitor` and `PtrVisitor`, to traverse the file system.

The `EntVisitor::visit` function takes an entity as input, prints its name, and then invokes `process_pointers`, which calls the `PtrVisitor::visit` function for every pointer in the entity. The `PtrVisitor::visit` function invokes `fetch`, which fetches the pointed-to entity from disk, and invokes `EntVisitor::visit` on it.

3.4 Limitations

The correctness of Spiffy applications depends on correctly written annotations. Therefore, if and when file system format changes do occur, the specifications will need to be updated. Spiffy applications will also need to update all file-system specific code that is affected by the format changes. These changes will likely only affect code that directly operates on the updated metadata structures, since the Spiffy library will provide safe traversal and parsing of any intermediate structures.

Currently, we have implemented an online application at the storage layer (metadata caching, see Section 5) that reads file system metadata, but does not modify it. We are exploring modifying file system metadata using Spiffy at the storage layer (which requires hooks into the file system code, e.g., for transactions and allocation [12]), and at the file system level (which enables more powerful applications).

Unlike typical file-system applications that operate at the VFS layer and are file-system independent, Spiffy applications operate directly on file-system specific struc-

```
EntVisitor ev;
PtrVisitor pv;
int PtrVisitor::visit(Entity & e) {
  Entity * tmp = ((Pointer &)e).fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Entity * sup;
  if ((sup = fs.fetch_super()) != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

Figure 5: Code for traversing and printing the types of all the metadata blocks in an Ext4 file system.

tures and are thus file-system dependent. Since file systems share common abstractions (e.g. files, directories, inodes), it may be possible to carefully abstract the functionality that is shared between implementations, reducing file-system dependence even further.

4 File System Applications

We have written five file-system aware storage applications using the Spiffy framework: a dump tool, a free space reporting tool, a type-specific metadata corruptor, a file system conversion tool, and a prioritized block layer cache. The first four applications operate offline, while the last one is an online application.

File System Dump Tool The file system dump tool parses all the metadata in a file system image and exports the result in an XML format, using file system traversal code similar to the example in Figure 5. In addition to `process_pointers`, the entity class provides a `process_fields` method that allows iterating over all fields (not just pointer fields) of the class. The dump tool can be configured to prevent structures such as unallocated inode structures from being exported.

Type-Specific Corruption Tool This tool is a variant of the dump tool that injects file-system corruption in a type-specific manner [2], allowing us to test the robustness of file systems and their tools. When we decide to corrupt a field, we cannot simply modify its in-memory value, since serialization is type-safe. For example, the serializer will refuse to serialize a corrupted value that violates its type constraints. Instead, corruption is performed after a block is serialized but before it is written.

Free Space Tool This tool shows file-system fragmentation by plotting a histogram of the size of free extents. The tool retrieves the metadata structures that store free space information and processes them (e.g., block bitmaps for Ext4, extent items for Btrfs, and segment information table (SIT) for F2FS). This logic is implemented using `process_by_type` (see Table 3) and a custom visit function that processes all the retrieved metadata structures. Code to traverse the file system and parse intermediate structures is provided by our library.

File System Conversion Tool Converting an existing file system into a file system of another type is a time-consuming process, involving copying files to another disk, reformatting the disk, and then copying the files back to the new file system. In-place file system conversion that updates file system metadata without moving most file data can speed up the conversion dramatically. While some such conversion tools exist,[3] they are hard to implement correctly and not generally available.

We have designed an in-place file system conversion tool using the Spiffy framework. Such a conversion tool requires detailed knowledge of the source and the destination file systems, and is thus a challenging application for our approach. In-place conversion involves several steps. First, the file and directory related metadata, such as inodes, extent mappings, and directory entries of the source file system, are parsed into a standard format. Second, the free space in the source file system is tracked. Third, if any source file data occupies blocks that are statically allocated in the destination file system, then those blocks are reallocated to the free space, and the conversion aborted if sufficient free space is not available. Finally, the metadata for the destination file system is created and written to disk. In our current tool, a power failure during the last step would corrupt the source file system. We plan to add failure atomicity in the future.

Our tool currently converts extent-based Ext4 file systems to log-structured F2FS file systems. The source file system is read using a custom set of visitors that efficiently traverse the file system and create in-memory copies of relevant metadata. For example, unused block groups can be skipped while processing block group descriptors. Next, we generate the free space list by reusing components from the free space tool, and then removing F2FS's static metadata area from the list. Then, Ext4 extents in the F2FS metadata area are relocated to the free space with their mappings updated. Finally, F2FS metadata is created from the in-memory copies and written to

[3] The convert utility converts FAT32 to NTFS [27], and updating to iOS 10.3 upgrades the file system from HFS+ to APFS [28]

disk, which involves allocation and pointer management, requiring significant file-system-specific logic.

Fortunately, various pieces of the code can be reused for different combinations of source and destination file system when adapting new file systems. As an example, only the code to copy Btrfs metadata from an existing file system and to list its free space is required to support the conversion from Btrfs to F2FS, since the in-memory data structures are generic across file systems that support VFS. If the file system does not support VFS, suitable default values can be used, which would be helpful for upgrading from a legacy file system such as FAT32.

Prioritized Block Layer Cache We have implemented a file-system aware block layer cache based on Bcache [20]. Our cache preferentially caches the files of certain priority users, identified by the `uid` of the file. This caching policy can dramatically improve workload performance by improving the cache hit rate for prioritized workloads, as shown in previous work [26]. Bcache uses an LRU replacement policy; in our implementation, blocks belonging to priority users are given a second chance and are only evicted if they return to the head of the LRU list without being referenced.

We use a runtime interpretation module, described in more detail in Section 5, to identify metadata blocks at the block layer without any modifications to the file system. We track the data extents that belong to file inodes containing the `uid` of a priority user, so that we can preferentially cache these extents. For Ext4, we use custom visit functions to parse inodes and determine the priority extent nodes. Similarly, we parse the priority extent nodes to determine the priority extent leaves, which contain the priority data extents.

For Btrfs, the inodes and their file extent items may not be placed close together (e.g., within the same B-tree leaf block), and so parsing an inode object will not provide information about its extents. Fortunately, the key of a file extent item is its associated inode number, making it easy to track the file extents of priority users.

5 Implementation

We implemented a compiler that parses Spiffy annotations. The compiler generates the file system's internal representation in a symbol table, containing the definitions of all the file system metadata, their annotations, their fields (including type and symbolic name), and each of their field's annotations. Next, it detects errors such as duplicate declarations or missing required arguments. Finally, the symbol table and compiler options are exported for use by the compiler's backend.

Spiffy's backend generates C++ code for a file-system specific metadata library using Jinja2 [22]. The library can be compiled as either a user space library or as part of a Linux kernel module. We linked our module, including our generated library, into the Linux kernel by porting some C++ standard containers to the kernel environment and integrating the GNU g++ compiler into the kernel build process, which required minor changes.

Every annotated structure is wrapped in a class that allows introspection. Each field in the wrapped class can refer to its name, type and size, and has a reference to the containing structure. The generated library performs various types of error-checking operations. For example, the parsing of offset fields ensures that objects do not cross container boundaries, and that all variable-sized structures fit within their containers. These checks are essential if an application aims to handle file system corruption. When parsing does fail, an error code is propagated to the caller of the `parse` or `serialize` function.

Address Spaces Annotation developers must implement the `IO` interface shown in Table 3. The Ext4 file address space implementation for the `Ext4IO` class (see Figure 5) requires fetching the file contents associated with an inode number. For Btrfs, we currently support the RAID address space for a single device, which only allows metadata mirroring (RAID-1). For F2FS, we support the NID address space, which maps a NID (node id) to a node block. The implementation involves a lookup to see if a valid mapping entry is in the journal. If not, the mapping is obtained from the node address table.

Runtime Interpretation Offline Spiffy applications use variants of the file-system traversal algorithm in Figure 5. Spiffy also supports online file-system aware storage applications via a kernel module that performs file system interpretation at the block layer of the Linux kernel using the generated libraries. These storage applications are typically difficult to write and error prone, since manual parsing code is needed for each block type. However, our implementation only requires a small amount of bootstrap code to support any annotated file system. The rest of the code is file-system independent.

In offline applications, the fetch function reads data from disk and parses the structure. The type of the structure is known from the pointer that is passed to the fetch function. In contrast, for online interpretation, the file system performs the read, and the application just needs to parse it. The `parse_by_type` function in Table 3 allows parsing of arbitrary buffers and constructing the corresponding containers, without the need for an `IO` object to read data from disk. However, it needs to know the type of the block before parsing is possible. Our runtime interpretation depends on the fact that a pointer to a metadata block must be read before the pointed-to block is read. When a pointer is found during the parsing of a block, the module tracks the type of the pointed-to block so that its type is known when it is read.

Our module exports several functions, including `interpret_read` and `interpret_write`, that need to be placed in the I/O path to perform runtime interpretation. These functions operate on locked block buffers. The module maintains a mapping between block numbers and their types. After intercepting a completed read request, it checks whether a mapping exists, and if so, it is a metadata block and it gets parsed. Next, `process_pointers` is invoked with a visitor that adds (or updates) all the pointers that are found in the block into the mapping table. If a parsed block will be referenced later (e.g., super block), we make a copy so that it is available during subsequent parsing of structures that depend on the value of its fields (e.g., parsing the Ext4 inode block requires knowing the size of an inode, which is in the super block). The local copy is atomically replaced when a new version of the block is written to disk.

When the I/O operation is a write, the module needs to determine the type of the written block. A statically allocated block can be immediately parsed because its type will not change. For example, most metadata blocks in Ext4 are statically allocated. However, in Btrfs, the super block is the only statically allocated metadata block. For dynamically allocated blocks, the block must first be labeled as unknown and its contents cached, since its type may either be unknown or have changed. Interpretation for this block is deferred until it is referenced by a block that is subsequently accessed (either read or written), and whose type is known. At that point, the module will interpret all unknown blocks that are referenced.

Since most dynamically-typed blocks are data blocks, they should be discarded immediately to reduce memory overhead. For the Btrfs file system, this is relatively easy because metadata blocks are self-identifying. For Ext4, these blocks need to be temporarily buffered until they can be interpreted. However, we use a heuristic for Ext4 to quickly identify dynamically-typed blocks that are definitely not metadata, to reduce the memory overhead of deferred interpretation. The block is first parsed as if it were a dynamically allocated block (e.g., a directory block or extent metadata block), and if the parsing results in an error, then the block is assumed to be data and discarded. This heuristic could be used in other file systems as well because most file systems have a small number of dynamically allocated metadata block types, or their blocks are self-identifying.

The module currently relies on the file system to issue `trim` operations to detect deallocation of blocks so that stale entries can be removed from the mapping table. Since file systems do not guarantee correct implementation of `trim`, the module additionally flushes out entries for dynamically allocated blocks that have not been accessed recently. This works for a caching application, but may lead to mis-classification for other runtime applications. Accurate classification can be implemented by keeping the previous versions of blocks and comparing the versions at transaction commit time. However, it comes with a higher memory overhead [8].

File System	Line Count	Annotated	Structures
Ext4	491	113	15+10+4
Btrfs	556	151	27+4+1
F2FS	462	127	14+16+5

Table 4: File system structure annotation effort.

6 Evaluation

In this section, we discuss the effort required to annotate the structures of existing file systems, the effort required to write Spiffy applications, and the robustness of Spiffy libraries. We then evaluate the performance of our file-system conversion tool and the file-system aware block-layer caching mechanism.

6.1 Annotation Effort

Table 4 shows the effort required to correctly annotate the Ext4, Btrfs and F2FS file systems. The second column shows the number of lines of code of existing on-disk data structures in these file systems. The lines of code count was obtained using `cloc` [6] to eliminate comments and empty lines. The third column shows the number of annotation lines. This number is less than one-third of the total line count for all the file systems.

The last column is listed as $A+B+C$, with A showing no modification to the data structure (other than adding annotations), B showing the number of data structures that were added, and C showing the number of data structures that needed to be modified. Structure declarations needed to be added or modified for three reasons:

1. We break down structures that benefit from being declared as conditionally inherited types. For example, `btrfs_file_extent_item` is split into two parts: the header and an optional footer, depending on whether it contains inline data or extent information.
2. Simple structures such as Ext4 extent metadata blocks, are not declared in the original source code. However, for annotation purposes, they need to be explicitly declared. All of the added structures in Ext4 belong to this category.
3. Some data structures with a complex or backward-compatible format require modifications to enable proper annotation. For example, Ext4 inode retains its Ext3 definition in the official header file even though the `i_block` field now contains extent tree information rather than block pointers. We redefined the Ext4 inode structure and replaced `i_block` with the extent header followed by four extent entries.

6.2 Developer Effort

Dump Tool: The file system dump tool includes a file-system independent XML writer module, written in 565 lines of code. The main function for each file system is written in 40 to 50 lines of code. The dump tool is helpful for debugging issues with real file systems. In addition, an expert can verify that the annotations are correct when the output of the dump tool matches the expected contents of the file system. Therefore, this tool has become an integral part of our development process.

Type-Specific Corruptor: This tool is written in 455 lines of code, with less than 30 lines of code required for the main function of each file system. The structure that the user wants to corrupt is specified via the command line and the tool uses `process_by_type` to find it, without the need for file-system specific code.

Free Space Tool: The file system free space tool has 271 lines of file-system independent code. File-system specific parts require 76 lines for Ext4, 77 lines for Btrfs, and 194 lines for F2FS. F2FS requires more code due to the complex format of its block allocation information.

Conversion Tool: The Spiffy file system conversion tool framework is written in 504 lines of code. The code for reading Ext4 takes 218 lines, the code to convert to the F2FS file system requires 1760 lines, and the file-system developer code for F2FS, which is reused in other applications such as the dump tool, consists of 383 lines. We also wrote a manual converter tool that uses the `libext2fs` [30] library to copy Ext4 metadata from the source file system, and manually writes raw data to create an F2FS file system. The manual converter has 223 lines of Ext4 code, and 2260 lines for the F2FS code. While the two converters have similar number of lines of code, the Spiffy converter has several other benefits. For the source file system, the manual converter takes advantage of the `libext2fs` library. Writing the code to convert from a different source file system would require significant effort, and would require much more code for a file system such as ZFS that lacks a similar user-level library. On the destination side, the Spiffy converter requires many file-system specific lines of code to manually initialize each newly created object. However, Spiffy checks constraints on objects and uses the `create_container` and `save` functions to create and serialize objects in a type-safe manner, while the manual converter writes raw data, which is error-prone, leading to the types of bugs discussed in Section 2.

Prioritized Cache: The original Bcache code consisted of 10518 lines of code. To implement prioritized caching we added 289 lines to this code, which invoke our generic runtime metadata interpretation framework, consisting of 2158 lines of code. This framework provides hooks to specify file-system specific policies. Our Ext4-specific policy requires 111 lines of code, and the Btrfs-specific policy requires 134 lines of code. Currently, we have not implemented prioritized caching for F2FS, which would require tracking NAT entries, similar to how we track inode numbers for Btrfs to find file extents.

6.3 Corruption Experiments

We use our type-specific corruption tool to evaluate the robustness of Spiffy generated libraries. The experiment fills a 128MB file system image with 12,000 files and some directories, then clobbers a chosen field in a specific metadata structure (e.g., one of the inode structures) to create a corrupted file system image. We corrupt each field in each type of metadata structure three times, twice to a random value and once to zero.

The Spiffy dump tool was able to generate correctly formatted XML files in the face of arbitrary single-field corruptions for all of these images. When corruption is detected during the parsing of a container or a pointer fetch (i.e., pointer address is out-of-bound or fails a placement constraint), an error is printed and the program stops the traversal.

Table 5 describes the crashes we found when we ran existing tools on the same corrupted images. For dumpe2fs (dump tool for Ext4) v1.42.13, we found a single crash when the `s_creator_os` field of the super block is corrupted. For dump.f2fs v1.6.1-1, we observed 5 instances of segmentation faults. Three of the crashes were due to corruption in the super block, and one crash each was detected for the summary block and inode structures. We were unable to trigger any crash-related bugs in btrfs-debug-tree v4.4.

These results are not unexpected since F2FS is a relatively young file system. Btrfs uses metadata checksumming to detect corruption, and thus requires corruption to be injected before checksum generation to fully test the robustness of its dump tool. Lastly, dumpe2fs does not traverse the full file system metadata, and so does not encounter most of the metadata corruption. Our Spiffy dump tool is both more complete and more robust than dumpe2fs, without requiring significant testing effort.

We also tried an extensive set of random corruption experiments, and none of the existing tools crashed, showing that our type-specific corruptor is a useful tool for testing the robustness of these applications.

6.4 File System Conversion Performance

We compare the time it takes to perform copy-based conversion, versus using the Spiffy-based and the manually written in-place file-system conversion tools. The results are shown in Table 6. The experiments are run on an Intel 510 Series SATA SSD. We create the file set using Filebench 1.5-a3 [32] in an Ext4 partition on the SSD,

Tool Name	Structure	Field	Description
dumpe2fs	super block	`s_creator_os`	index out of bound error during OS name lookup
dump.f2fs	super block	`log_blocks_per_seg`	index out of bound error while building nat bitmap
	super block	`segment_count_main`	null pointer dereference after calloc fails
	super block	`cp_blkaddr`	double free error during error handling (no valid checkpoint)
	summary block	`n_nats`	index out of bound error during nid lookup
	inode	`i_namelen`	index out of bound error when adding null character to end of name

Table 5: List of segmentation faults found during type-specific corruption experiments.

# files	Copy Converter	Manual Conv.	Spiffy Conv.
20000	188.2 ± 3.7s	6.6 ± 0.5s	7.0 ± 0.2s
1000	192.7 ± 2.3s	3.3 ± 0.1s	3.8 ± 0.0s
100	195.1 ± 0.2s	3.3 ± 0.1s	3.7 ± 0.1s

Table 6: Time required for each technique to convert from Ext4 to F2FS for different number of files.

and then convert the partition to F2FS. The 20K file set uses the `msnfs` file size distribution with the largest file size up to 1GB. The rest of the file sets have progressively fewer small files. All file sets have a total size of 16GB. For the copy converter, we run `tar -aR` at the root of the SSD partition and save the tar file on a separate local disk. We then reformat the SSD partition and extract the file set back into the partition.

The copy converter requires transferring two full copies of the file set, and so it takes 30x to 50x longer than using the conversion tools, which only need to move data blocks out of F2FS's static metadata area and then create the corresponding F2FS metadata. Both conversion tools take more time with larger file sets since they need to handle the conversion of more file system metadata. The library-assisted conversion tool performs reasonably compared to its manually-written counterpart, with at most a 16.7% overhead for the added type-safety protection that the library offers.

6.5 Prioritized Cache Performance

We measure the performance of our prioritized block layer cache (see Section 4), and compare it against LRU caching with one or two instances of the same workload.

Our experimental setup includes a client machine connected to a storage server over a 10Gb Ethernet using the iSCSI protocol. The storage server runs Linux 3.11.2 and has 4 Intel Processor E7-4830 CPUs for a total of 32 cores, 256GB of memory and a software RAID-6 volume consisting of 13 Hitachi HDS721010 SATA2 7200 RPM disks. The client machine runs Linux 4.4.0 with Intel Processor E5-2650, and an Intel 510 Series SATA SSD that is used for client-side caching. To mimic the memory-to-cache ratio of real-world storage servers, we limit the memory on the client to 4GB and use 8GB of the SSD for write-back caching. The RAID partition is formatted with either the Ext4 or Btrfs file system and is used as the primary storage device. To avoid any

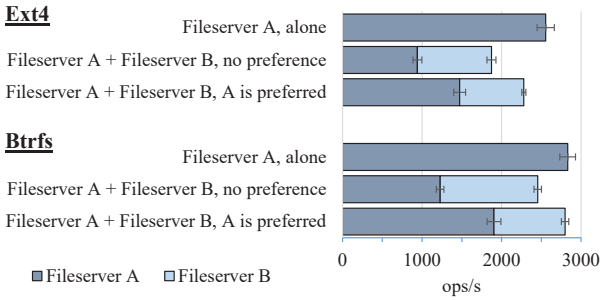

Figure 6: Throughput of prioritized caching over LRU caching with one or two file servers for Ext4 and Btrfs.

scheduling related effects, the NOOP I/O scheduler is used in all cases for both the caching and primary device.

We use a pair of identical Filebench fileserver workloads to simulate a shared hosting scenario with two users where one requires higher storage performance than the other. We generate a total file set size of 8GB with an average file size of 128KB, for each workload. The fileserver personality performs a series of create, write, append, read and delete of random files throughout the experiment. Filebench reports performance metrics every 60 seconds over a period of 90 minutes. Performance initially fluctuates as the cache fills, therefore we present the average throughput over the last 60 minutes of the experiment, after performance stabilizes.

Figure 6 shows the average throughput for each of the experiments in operations per second. The error bars show 95% confidence intervals. First, we establish the baseline performance of a single fileserver instance running alone, which has a cache hit ratio of 64% and 54% for Ext4 and Btrfs, respectively. Next, we run two instances of fileserver to observe the effect of cache contention. We see a drastic reduction in cache hit ratio to 23% and 24% for Ext4 and Btrfs, respectively. Both fileservers have similar performance, which is between 2.3x and 2.7x less than when running alone. When we apply preferential caching to the files used by fileserver A, however, its throughput improves by 60% over non-prioritized LRU caching when running concurrently with fileserver B, with the overall cache hit ratio improving to 46% and 53% for Ext4 and Btrfs, respectively. Prioritized caching also improves the aggregate throughput of the system by 14% to 22%. Giving priority to one of the two jobs implicitly reduces cache contention.

These results show that storage applications using our generated library can provide reasonable performance improvements without changing the file system code.

7 Related Work

A large body of work has focused on storage-layer applications that perform file-system specific processing for improving performance or reliability. Semantically-smart disks [24] used probing to gather detailed knowledge of file system behavior, allowing functionality or performance to be enhanced transparently at the block layer. The probing was designed for Ext4-like file systems and would likely require changes for copy-on-write and log-structured file systems. Spiffy annotations avoid the need for probing, helping provide accurate block type information based on runtime interpretation.

I/O shepherding [12] improves reliability by using file structure information to implement checksumming and replication. Block type information is provided to the storage layer I/O shepherd by modifying the file system and the buffer-cache code. Our approach enables I/O shepherding without requiring these changes. Also, unlike I/O shepherding, Spiffy allows interpreting block contents, enabling more powerful policies, such as caching the files of specific users.

A type-safe disk extends the disk interface by exposing primitives for block allocation and pointer relationships [23], which helps enforce invariants such as preventing access to unallocated blocks, but this interface requires extensive file system modifications. We believe that our runtime interpretation approach allows enforcing such type-safety invariants on existing file systems.

Serialization of structured data has been explored through interface languages such as ASN.1 [25] and Protocol Buffers [31], which allow programmers to define their data structures so that marshaling routines can be generated for them. However, the binary serialization format for the structures is specified by the protocol and not under the control of the programmer. As a result, these languages cannot be used to interpret the existing binary format of a file system.

Data description languages such as Hammer [21] and PADS [7] allow fine-grained byte-level data formats to be specified. However, they have limited support for non-sequential processing, and thus their parsers cannot interpret file system I/O, where a graph traversal is required rather than a sequential scan. Furthermore, with online interpretation, this traversal is performed on a small part of the graph, and not on the entire data.

Nail [3] shares many goals with our work. Its grammar provides the ability to specify arbitrarily computed fields. It also supports non-linear parsing, but its scope is limited to a single packet or file, and so it does not support references to external objects. Our annotation language overcomes this limitation by explicitly annotating pointers, which defines how file system metadata reference each other. We also provide support for address spaces, so that address values can be mapped to user-specified physical locations on disk.

Several projects have explored C extensions for expressing additional semantic information [19, 35, 29]. CCured [19] enables type and memory safety, and the Deputy Type System [35] prevents out-of-bound array errors. Both projects annotate source code, perform static analysis, and add runtime checks, but they are designed for in-memory structures.

Formal specification approaches for file systems [1, 5] require building a new file system from scratch, while our work focuses on building tools for existing file systems. Chen et al. [5] use logical address spaces as abstractions for writing higher-level file system specifications. This idea inspired our use of an address space type for specifying pointers. Another method for specifying pointers is by defining paths that enable traversing the metadata tree to locate a metadata object, such as finding the inode structure from an inode number [14, 10]. These approaches focus on the correctness of file-system operations at the virtual file system layer, whereas our goal is to specify the physical structures of file systems.

8 Conclusion

Spiffy is an annotation language for specifying the on-disk file system data structures. File system developers annotate their data structures using Spiffy, which enables generating a library that allows parsing and traversing file system data structures correctly.

We have shown the generality of our approach by annotating three vastly different file systems. The annotated file system code serves as detailed documentation for the metadata structures and the relationships between them. File-system aware storage applications can use the Spiffy libraries to improve their resilience against parsing bugs, and to reduce the overall programming effort needed for supporting file-system specific logic in these applications. Our evaluation suggests that applications using the generated libraries perform reasonably well. We believe our approach will enable interesting applications that require an understanding of storage structures.

Acknowledgements

We thank the anonymous reviewers and our shepherd, André Brinkmann, for their valuable feedback. We specially thank Michael Stumm, Ding Yuan, Mike Qin, and Peter Goodman for their insightful suggestions. This work was supported by NSERC Discovery.

References

[1] AMANI, S., RYZHYK, L., AND MURRAY, T. Towards a fully verified file system, 2012. EuroSys Doctoral Workshop 2012.

[2] BAIRAVASUNDARAM, L. N., RUNGTA, M., AGRAWA, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Analyzing the effects of disk-pointer corruption. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)* (2008), IEEE, pp. 502–511.

[3] BANGERT, J., AND ZELDOVICH, N. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 615–628.

[4] BUCKEYE, B., AND LISTON, K. Recovering deleted files in linux. `http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/SA/v11/i04/a9.htm`, 2006.

[5] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 18–37.

[6] DANIAL, A. Cloc–count lines of code. *Open source* (2009). `http://cloc.sourceforge.net/`.

[7] FISHER, K., AND WALKER, D. The pads project: an overview. In *Proceedings of the 14th International Conference on Database Theory* (2011), ACM, pp. 11–17.

[8] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage 8*, 4 (Dec. 2012), 15:1–15:29.

[9] GAMMA, E. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[10] GARDNER, P., NTZIK, G., AND WRIGHT, A. Local reasoning for the posix file system. In *European Symposium on Programming Languages and Systems* (2014), Springer, pp. 169–188.

[11] GEDAK, C. *Manage Partitions with GParted How-to*. Packt Publishing Ltd, 2012.

[12] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with I/O shepherding. In *Proc. of the Symposium on Operating Systems Principles (SOSP)* (2007), pp. 293–306.

[13] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: A declarative file system checker. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[14] HESSELINK, W. H., AND LALI, M. I. Formalizing a hierarchical file system. *Electronic Notes in Theoretical Computer Science 259* (2009), 67–85.

[15] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 273–286.

[16] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2013).

[17] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. ffsck: The fast file system checker. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2013).

[18] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proc. of the Symposium on Operating Systems Principles (SOSP)* (2011), pp. 57–70.

[19] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 128–139.

[20] OVERSTREET, K. Linux bcache, Aug. 2016. `https://bcache.evilpiepirate.org/`.

[21] PATTERSON, M., AND HIRSCH, D. Hammer parser generator, march 2014. `https://github.com/UpstandingHackers/hammer`.

[22] RONACHER, A. Jinja2 documentation, 2011.

[23] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 15–28.

[24] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 73–88.

[25] STEEDMAN, D. *Abstract syntax notation one (ASN. 1): the tutorial and reference*. Technology appraisals, 1993.

[26] STEFANOVICI, I., THERESKA, E., O'SHEA, G., SCHROEDER, B., BALLANI, H., KARAGIANNIS, T., ROWSTRON, A., AND TALPEY, T. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 174–181.

[27] TECHNET, M. How to convert fat disks to ntfs. `https://technet.microsoft.com/en-us/library/bb456984.aspx`.

[28] TOM WARREN. Apple is upgrading millions of iOS devices to a new modern file system today. `https://www.theverge.com/2017/3/27/15076244/apple-file-system-apfs-ios-10-3-features`. Accessed: 2017-03-27.

[29] TORVALDS, L., TRIPLETT, J., AND LI, C. Sparse–a semantic parser for c. *see http://sparse.wiki.kernel.org* (2007).

[30] TS'O, T. E2fsprogs: Ext2/3/4 filesystem utilities. `http://e2fsprogs.sourceforge.net/`, 2017.

[31] VARDA, K. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).

[32] WILSON, A. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies* (2008). `https://github.com/filebench/filebench/`.

[33] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (2006), 393–423.

[34] ZALEWSKI, M. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2016.

[35] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 45–60.