

Spiffy: Enabling File-System Aware Storage Applications

Abstract

Many file-system applications such as defragmentation tools, file system checkers or data recovery tools, operate at the storage layer. Today, writing these storage applications requires detailed knowledge of the file-system format. Application developers must spend significant time learning the format, 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. Our approach simplifies the development of these applications, making it easier to write applications that work across different file systems. We have written annotations for the Linux Ext4, Btrfs and F2FS file systems. We have developed three applications that work for these file systems, an offline file system dump tool, 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.

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 [18], 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. For example, whether the block is a data or a metadata block, whether it belongs to a specific type of file, etc. These applications help improve the performance or reliability of a storage system by performing

Storage Applications	Category	Purpose
Differentiated services [19]	online	performance
Defragmentation tool	either	
File system checker [14]	offline	reliability
Data recovery tool [4]	either	
IO shepherding [13]	online	
Runtime verification [9]	online	
File system conversion tool	either	administrative
Partition editor [12]	offline	
Type-specific corruption [2]	either	debugging
Metadata dump tool	offline	

Table 1: Examples of file-system aware storage applications. Offline applications have exclusive access to the file system; online applications operate while the file system is in use.

file-system specific processing at the storage layer. For example, differentiated storage services [19] improve performance by preferentially caching blocks that contain file-system metadata or the data of small files. I/O shepherding [13] improves reliability by using file structure information to implement checksumming and replication. Similarly, the Recon system [9] improves reliability by verifying the correctness of file-system metadata operations 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 [34]. Some storage applications reduce the amount of file-system specific code in their implementation by modifying their target file system and operating system [19, 13]. However, this approach only

works for specific file systems, and it can introduce its own set of bugs. It also requires using 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. Our approach enables file system developers to specify the format of their file system using a domain-specific language so that the file system metadata can be identified, parsed and traversed correctly and automatically. Application developers can thus focus on the logic of their applications, and write file-system specific policies more easily. This separation of the development of applications from the specification and interpretation of file system formats simplifies the process of building these applications, or adapting them for new file systems.

We introduce Spiffy,¹ a language for annotating file system data structures defined in the C language. The annotations augment existing data structure definitions, so that the relationships between file system structures can be explicitly and concisely stated. Spiffy allows file system developers to unambiguously specify the *physical* layout of the file system. We compile the annotated source code to produce a library that enables type-safe parsing and traversal of file system metadata, both for offline or online applications. The generic interfaces provided by the library simplify the development of file-system aware applications, making it easier to write applications that work across different file systems.

The complexity of modern file systems [17] raises several challenges for our specification-based approach. There are many aspects of file system structures and their relationships that are not captured by their declarations in header files. First, an on-disk pointer in a file-system structure may be implicitly specified 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; it requires higher-level knowledge from the application programmer.

```
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 F2FS 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 ints). 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 § 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, and by using a dependency tracking mechanism that ensures that the referenced structures are valid. 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 data to be accessed efficiently, and reducing the memory footprint of the applications.

We have annotated three 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 [16]. We have implemented three applications that are designed to work across file systems, a file system dump tool, a file system converter, and a storage layer service that preferentially caches data for specific users.

The rest of the paper is organized as follows. In § 2, we motivate the need for our approach. § 3 presents the core concepts that led to the design of the annotation language and the library API. § 4 describes the applications that we have implemented using the generated library. § 5 describes the implementation of our system, and § 6 assesses the programming effort needed to annotate each file system and the performance of our applications. We present related work in § 7 and discuss our conclusions in § 8. For reference, Appendix § A shows our file system annotation language with examples of annotated structures for the Ext4, Btrfs and F2FS file systems.

2 Bugs in File-System Applications

We motivate this work by presenting various bugs caused by incorrect parsing of file-system metadata in storage

¹Specifying and Interpreting the Format of Filesystems

	Tool	FS	Bug Title	Closed
1	libparted	Fat32	#222266: 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
6	btrfsck	Btrfs	Bug 59541 - Btrfsck reports free space cache errors when using skinny extents	2013-06

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

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 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 $2^{|\text{value}|}$ 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. In this version of e2fsck, the file system checker failed to detect corrupted directory entries if the size field of the entries was set to zero, which results in no repair being performed. Ironically, other programs, such as debugfs, ls, and the file system itself, can correctly detect the corruption.
4. Ext2/3/4 inodes contain union fields for storing operating system specific metadata. Unfortunately, a sanity check was omitted in e2fsck prior to accessing this field, causing erroneous repairs to be performed when the creator OS is not Linux. Consequently, the file system becomes corrupted for Hurd and possibly other operating systems.
5. A fuzzer that generates test cases that trigger different internal states of its target binary [35] 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.
6. When the skinny metadata feature was added to Btrfs, the developers neglected to also patch Btrfsck, resulting in false error reports. This bug shows the difficulty of keeping all relevant applications up-to-date with changing file system formats.

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 automatically 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.

3 Approach

The purpose of our file system annotation language is to enable safe interpretation of file system structures, in both offline and online contexts, without requiring file-system specific code. 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 their 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 the application writer to create file-system specific policies that are applied to a subset of a file system’s metadata. 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 perform selective traversal using type introspection to find and operate on directory entries. Since the directory entry format may not be the same across file systems, the application may still require file-system specific code, but this is essential to the application logic.

Our annotation-based approach offers several advantages. First, it provides a concise and clear documenta-

```

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.

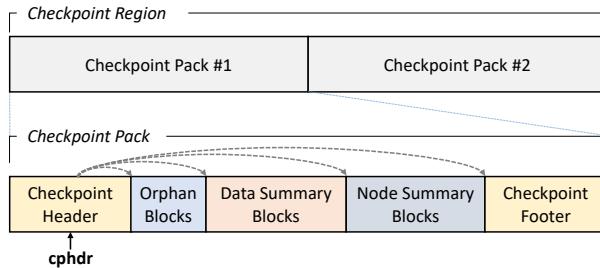


Figure 2: Each F2FS checkpoint pack contains a header followed by a variable number of orphan blocks.

tion 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. For example, differentiated storage services [19] was implemented by modifying the file system and the kernel’s storage stack to enable I/O classification. With our approach, this application can be implemented by using introspection 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. When format changes do occur, the specifications need to be updated, which is easier than modifying all the related tools.

3.1 Designing Annotations

In this section, we describe the concepts that motivated the design of our annotation language for specifying the format of file system structures.

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 § 1. The difference between a file system pointer and an in-memory pointer is that the contents of an in-memory pointer are always interpreted as the in-memory address of the pointed-to data, but interpreting the address con-

tained 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.

Multiple pointers may refer to the same structure. For example, the block group descriptor tables in Ext4 refer to inode tables, which have inode structures in them. Similarly, directory entries have an inode number that also refer to these inode structures, as shown in Figure 1. We prefer to annotate the descriptor tables with a block type pointer because the inode table can then be viewed as a contiguous structure. Annotating the inode number is possible, but will require the annotation developer to implement an inode address space for mapping inode numbers to inode structures.

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 su-

²This is especially confusing because `name` has a fixed size in the definition.

per 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. It 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 its 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 condition* clause, evaluated at runtime, to support such context-sensitive types.

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 Ext4 is implicitly the block 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, but without requiring any changes to the underlying file-system format.

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. For example, only one inode block needs to be fetched from disk to access an inode structure.

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 the 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 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 [10] so that a programmer can 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, including objects (structures, vectors, fields), pointers, containers and extents. The `accept_pointers` function invokes the `visit` function of an application-defined `Visitor` on each pointer within the entity.

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

Base Class	Member Function	Description
Spiffy File System Library		
Entity	int accept_fields(Visitor & v) int accept_pointers(Visitor & v)	allows <i>v</i> to visit all fields of this object allows <i>v</i> to visit all pointer fields of this object
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) Entity * fetch_super() Entity * create_container(int type, Path & p) Entity * parse_by_type(int type, Path & p, Address & addr, const char * buf, size_t len)	instantiates a new file system object retrieves the super block from disk creates a new container of metadata <i>type</i> parses the buffer as metadata <i>type</i> , using <i>p</i> to resolve cross structure dependencies
File System Developer		
IO	int read(Address & addr, char * & buf) int write(Address & addr, const char * buf) int alloc(Address & addr, int type)	reads from an address space specified by <i>addr</i> writes to an address space specified by <i>addr</i> allocates an on-disk address for metadata <i>type</i>
Application Programmer		
Visitor	int visit(Entity * e)	visits an entity and possibly process it

Table 3: Spiffy C++ Library API.

```
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.

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.

Figure 5 shows the **save** function for writing a container to disk. The function serializes the 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

```
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**.

```
int Container::save(bool alloc) {
    size_t len = this->address.size;
    char * buf = new char[len];
    this->serialize(buf, len);
    if (alloc)
        this->fs.io.alloc(this->address,
                           this->metadata_type);
    /* check placement constraint */
    this->fs.io.write(this->address, buf);
}
```

Figure 5: Abbreviated version of the **save** function.

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. With online interpretation, the application already has the buffer containing the metadata, knows its type, and just needs to parse it. The **parse_by_type** factory function allows the programmer to bypass the **fetch** function and allows parsing of arbitrary buffers and constructing the corresponding containers, without the need for an **IO** object to read data from disk.

```

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_type().name << endl;
    return e.accept_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 6: Code for traversing and printing the types of all the metadata blocks in an Ext4 file system.

3.3 Building Applications

Figure 6 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 later in § 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 `accept_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.

Consider another application that shows file-system fragmentation by plotting a histogram of the size of free extents in the file system. This application will require file-system specific logic depending on how free space is represented, e.g., bitmaps for Ext4, and free space extents for Btrfs. This logic is implemented by modifying the code shown in Figure 6 with custom visit functions for just these structures, while the rest of the code in the application will be common across file systems.

4 File System Applications

We wrote three file-system aware storage applications using the Spiffy framework: a dump tool; a file system conversion tool; and a prioritized block layer cache. The first two 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 shown in Figure 6. In addition to `accept_pointers`, the entity class provides an `accept_fields` method that allows iterating over all fields (not just pointer fields) of the class. Since some file system metadata may not be of interest, the XML writer provides APIs to help filter out irrelevant fields or structures. For example, the Ext4 dump tool excludes any unallocated inode structures from being exported to XML. The tool currently supports Ext3/4, Btrfs, and F2FS.

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 necessarily moving file data can speed file system conversion dramatically. Unfortunately, while some such conversion tools exist (e.g., `convert` utility that converts FAT32 to NTFS [29]), they are hard to implement correctly and generally not 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 supports converting 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

```

class Ext4DirEntry : public Entity {
    Ext4DirEntry() :
        inode("inode", "__le32"),
        ...
        name("name", "char[]", *this) {}
public:
    Integer<__le32> inode;
    Integer<__le16> rec_len;
    Integer<__u16> name_len;
    Vector<char> name;
    ...
};


```

Figure 7: Wrapper class for Ext4 directory entry.

block group descriptors. Next, we generate the free space list by processing Ext4’s block bitmaps, while 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 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 [21]. 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 [28]. 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 § 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 the `parse_by_type()` function shown in Table 3 and 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. (These annotations are described in Table 6 of Appendix A) 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 a file-system specific metadata library using Jinja2 [24]. The library can be compiled as either a user space library or as part of a Linux kernel module. We linked our generated library into the Linux kernel by porting some C++ standard containers and integrating the GNU g++ compiler into the kernel build process.

Every annotated structure is wrapped in a class that implements the `Entity` interface. Figure 7 shows an example structure for the Ext4 directory entry. The `name` field is initialized with its name and type for introspection, and also a reference to the structure so that it can reference `self` during parsing. We make each of the fields publicly visible by using the cast and assignment operators in the field’s template class. Application programmers can thus access these fields as if they were accessing the actual C 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.

Currently, the `fetch` function always reads data from storage because we have not implemented an entity cache. This doesn’t affect a tree traversal in which each entity is read once, but if a structure can be accessed using multiple paths, then it would be read multiple times.

Address Spaces We require annotation developers to implement the IO interface shown in Table 3. The `read` method maps a pointer address in an address space to a physical location on disk, and then reads a container of a

given size, specified by `addr.size`, into the buffer `buf`. `addr.offset` is used to read a container within an extent. We require a byte address space implementation so that the super block can be fetched at a fixed byte offset on disk. The super block usually contains the block size, which enables a block address space implementation.

The Ext4 file address space implementation for the `Ext4IO` class (see Figure 6) requires fetching the file contents associated with an inode number. It requires reading the corresponding inode structure, converting the size and offset arguments into a list of physical block numbers, fetching these blocks into memory, and combining the blocks together. 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 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 shown in Figure 6. Spiffy also supports online file-system aware storage applications. To do so, we programmed a module to perform file system interpretation at the block layer of the Linux kernel using the generated libraries. This class of file system application is 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 module only parse the block that is read by calling `FileSystem::parse_by_type()`. 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 encountered during the parsing a block, the module tracks the type of the pointed-to block. As a result, when the pointed-to block is read, its type is known.

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. The module maintain a mapping between block numbers and their types. After intercepting a completed read request, it checks whether a mapping exists in the mapping table, and if so, it is a metadata block and it gets parsed. Next, `accept_pointers` is invoked with a visitor that adds (or updates) all the pointers that are

found in the block into the mapping table.

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.

Our runtime interpretation system is currently used to read file system blocks. Supporting applications that write blocks, such as IO shepherding [13], will require an IO manager that can make its own IO requests.

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 [9]. However, it comes with a higher memory overhead.

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 then evaluate the performance of our file-system conversion tool and the file-system aware block-layer caching mechanism.

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.

6.1 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` [7] 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,³ 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 for data.
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 Application-Developer Effort

In this section, we evaluate the effort required to develop Spiffy applications.

Dump Tool: The file system dump tool is a simple tool in which the file-system independent XML writer module is written in 482 lines of C++ code, and the main function for each file system is written in 30 to 60 lines

³We consider the vector type to be an annotation and not a structure for this calculation.

of code, depending on the number of structures the programmer wants to skip. The dump tool is helpful for debugging issues with real file systems, and to verify the correctness of the annotations. In particular, 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 tool in our development process.

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` [31] 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. In this case, the two converters have similar number of lines of code, but the Spiffy converter has several other benefits. On the source side, the manual converter takes advantage of the `libext2fs` library. Changing this code for a different file system would require significant changes, and would require much more code for a file system such as ZFS that does not have a similar user-level library. On the destination side, the main reason that the Spiffy converter requires many file-system specific lines of code is that each newly created object needs to be initialized, and the initialization has to be performed manually. However, Spiffy uses the `create_container` and `save` functions to create and serialize objects in a type-safe manner, and checks constraints on objects, while the manual converter writes raw data, which is error-prone, leading to the types of bugs discussed in § 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 invokes 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; we expect that the code size would be similar to the other file systems.

6.3 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 5. The experiments are run on an

# of files	20000	5000	1000	100
Copy Convertor	188.17 ± 3.65 s	190.28 ± 2.15 s	192.74 ± 2.28 s	195.11 ± 0.18 s
Manual Converter	6.55 ± 0.53 s	3.46 ± 0.17 s	3.29 ± 0.11 s	3.25 ± 0.11 s
Spiffy Converter	7.03 ± 0.2 s	4.01 ± 0.09 s	3.84 ± 0.03 s	3.71 ± 0.13 s

Table 5: Time required for each technique to convert from Ext4 to F2FS for different number of files.

Intel 510 Series SATA SSD. We create the file set using Filebench 1.5-a3 [33] in an Ext4 partition on the SSD, 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 30 to 50 times 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 longer time with larger files 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. This shows the feasibility of using the library for general use when working with file system metadata.

6.4 Prioritized Cache Performance

We measure the performance of our prioritized block layer cache (see § 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 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

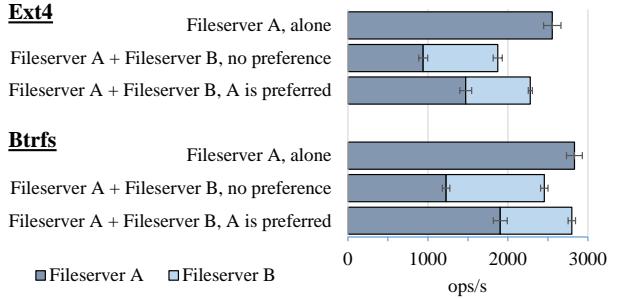


Figure 8: Throughput of prioritized caching over LRU caching with one or two file servers for Ext4 and Btrfs.

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 is set to report performance metrics every 60 seconds over a period of 90 minutes. However, since each experiment starts with an empty cache, the performance initially fluctuates. Thus, we present the average of the results from the last 60 minutes of the experiment, after performance stabilizes.

Figure 8 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. Interestingly, 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 the need to change 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. Seminal work on semantically-smart disks [26] 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. The Spiffy annotations avoid the need for probing, helping provide accurate block type information based on runtime interpretation that is mostly file-system agnostic.

I/O shepherding [13] improves reliability by using file structure information to implement checksumming and replication. The file system semantic information is provided to the storage layer I/O shepherd by modifying the file system and the buffer-cache code. Our approach will enable I/O shepherding without requiring any such changes. Furthermore, our approach enables adding support for a new file system more easily.

A type-safe disk extends the disk interface by exposing primitives for block allocation and pointer relationships [25], which helps enforce invariants such as preventing accesses to unallocated blocks, or allowing accesses to a Block X only after a Block Y that has a valid reference to Block X has been accessed. Using a type-safe disk requires extensive modifications to file systems. We believe that our runtime interpretation approach makes it feasible to enforce such type-safety invariants for existing file systems.

Serialization of structured data has been explored through interface languages such as ASN.1 [27] and Protocol Buffers [32], 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 [22] and PADS [8] 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 [20, 36, 30]. CCured [20] enables type and memory safety, and the Deputy Type System [36] 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.

Symbolic execution [5], model checking [34] and fuzz-testing [35] have been used to find file system bugs. In contrast, we proactively avoid bugs by adding type safety checks in the generated parsing and serializing routines. Nonetheless, these techniques would be helpful for detecting annotation errors that may be encountered during the development phase.

Formal specification approaches for file systems [1, 6] require building a new file system from scratch, while our work focuses on building tools for existing file systems. Chen et al. [6] use logical address spaces as abstractions for writing higher-level file system specifications in a concise manner. 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 when given an inode number [15, 11]. 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 a 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. Overall, we believe our approach will enable interesting applications that require an understanding of storage structures.

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] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 10.
- [6] CHEN, H., ZIEGLER, D., CHAJED, T., CHILIPALA, 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.
- [7] DANIAL, A. Cloc—count lines of code. *Open source* (2009). <http://cloc.sourceforge.net/>.
- [8] 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.
- [9] 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.
- [10] GAMMA, E. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
- [11] 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.
- [12] GEDAK, C. *Manage Partitions with GParted How-to*. Packt Publishing Ltd, 2012.
- [13] 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.
- [14] 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).
- [15] HESSELINK, W. H., AND LALI, M. I. Formalizing a hierarchical file system. *Electronic Notes in Theoretical Computer Science* 259 (2009), 67–85.
- [16] 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.
- [17] 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).
- [18] 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).
- [19] 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.
- [20] 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.
- [21] OVERSTREET, K. Linux bcache, Aug. 2016. <https://bcache.evilpiepirate.org/>.
- [22] PATTERSON, M., AND HIRSCH, D. Hammer parser generator, march 2014. <https://github.com/UpstandingHackers/hammer>.
- [23] RODEH, O., BACIK, J., AND MASON, C. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage* 9, 3 (Aug. 2013), 9:1–9:32.
- [24] RONACHER, A. Jinja2 documentation, 2011.
- [25] 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.
- [26] 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.
- [27] STEEDMAN, D. *Abstract syntax notation one (ASN. 1): the tutorial and reference*. Technology appraisals, 1993.
- [28] 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.
- [29] TECHNET, M. How to convert fat disks to ntfs. <https://technet.microsoft.com/en-us/library/bb456984.aspx>.
- [30] TORVALDS, L., TRIPLETT, J., AND LI, C. Sparse—a semantic parser for c. see <http://sparse.wiki.kernel.org> (2007).
- [31] TS’O, T. E2fsprogs: Ext2/3/4 filesystem utilities. <http://e2fsprogs.sourceforge.net/>, 2017.
- [32] VARDA, K. Protocol buffers: Google’s data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).
- [33] WILSON, A. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies* (2008). <https://github.com/filebench/filebench/>.
- [34] 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.
- [35] ZALEWSKI, M. American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>, 2016.
- [36] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRACK, 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.

Keyword	Description	Arguments	Meaning
FSSTRUCT	File system structure	name=IDENT unit=UNIT base=TYPE, when=BOOL size=INT location=INT	Name of the structure for cross referencing Can be object, container, or extent Structure inherits from base when condition is true Size of the structure Location of the super block or placement constraint
POINTER	Field is a pointer to a file system structure	aspc=IDENT type=TYPE when=BOOL size=INT name=IDENT, expr=INT	Name of an address space type (not used by FIELD) Type of the referenced structure Pointer is valid when condition is true Size of the referenced metadata Name of an implicit pointer, its expression
FIELD	Field is an offset to a structure within the same container	name=IDENT	Name of the address space type
CHECK	Constraint check	expr=BOOL	Condition for the structure's correctness
VECTOR	1. Defines a vector type 2. Defines a flexible array field	name=IDENT type=TYPE count=INT size=INT sentinel=BOOL	Name of the vector or the field name of the array Structure type of the contained elements Number of elements in the vector Size of the vector, in bytes Sentinel value that specifies the end of vector
ADDRSPACE	An address space		

IDENT is a valid C identifier. TYPE is the type name of a structure or vector type. BOOL and INT are syntactically valid, dynamically scoped, C expressions that evaluate to a boolean and integer type.

Table 6: Spiffy file system annotations.

A Annotation Language

Spiffy uses annotations on C structures to specify the format of file-system structures. We chose this approach to reduce duplication of structure definitions. The annotations are defined using C preprocessor macros. They are designed to be compatible with existing code by expanding to empty code during normal compilation. Although many annotations can be added to existing structures, sometimes we need to add new structures or modify existing structures when they are a poor fit for our needs.

Table 6 shows the list of annotations supported by Spiffy. Each annotation is written using one or more keywords, followed by their arguments. We now describe each annotation.

FSSTRUCT, FSSUPER These annotations are written by replacing the `struct` keyword in a C structure with `FSSTRUCT` or `FSSUPER`. They help distinguish file system metadata from in-memory file system structures so that our compiler only parses C data structures marked with these two annotations. The `FSSUPER` annotation identifies the root of the file system. The `location` argument describes its physical location as an offset (in bytes) from the beginning of the file system image. For `FSSTRUCT`, the `location` argument optionally specifies its placement constraint. The `name` argument is used by

a descendant to reference this structure (see § 3.1). The `unit` argument specifies the access unit of the structure, with `object` being the default unit.

The `base-when` argument enables supporting context-sensitive types. It defines a structure that is derived from a base structure when the condition is true. Conceptually, the derived structure is appended to the base structure, similar to the way inheritance is implemented in object oriented languages. Figure 9 shows an example in which the F2FS inode structure is inherited by either a directory inode structure or a file inode structure, depending on the mode of the inode. The use of two derived inode structures allows using different types in the two structures. For example, we use a `dir_block` pointer in the directory inode and a `data_block` pointer in the file inode.

Notice that the two arguments in the `FSSTRUCT` definition of `f2fs_inode` reference the super block using the name `sb`. In addition, the `location` argument specifies its placement constraint so that incorrect allocation will not result in clobbering parts of the F2FS static metadata area. The `$self` notation refers to the address of the container (see Figure 3).

POINTER, FIELD, ADDRESSPACE The `POINTER` annotation is used to specify the address type and the pointed-to type of a pointer. It allows fetching a structure from disk and parsing it with the correct type informa-

```

#define BLOCK_SIZE 1 << sb.log_blocksize
FSSTRUCT(location=$self.id >= sb.main_blkaddr,
         size=BLOCK_SIZE) f2fs_inode {
    __le16 i_mode;
    ...
};

typedef FSSTRUCT(base=struct f2fs_inode,
                 when=self.i_mode & S_IFDIR) {
    POINTER(aspc=block, type=dir_block)
    __le32 i_addr[DEF_ADDRS_PER_INODE];
    POINTER(aspc=nid, type=dir_direct_block)
    __le32 i_dnid[2];
    ...
} f2fs_dir_inode;
typedef FSSTRUCT(base=struct f2fs_inode,
                 when=self.i_mode & S_IFREG) {
    POINTER(aspc=block, type=data_block)
    __le32 i_addr[DEF_ADDRS_PER_INODE];
    POINTER(aspc=nid, type=data_direct_block)
    __le32 i_dnid[2];
    ...
} f2fs_reg_inode;

```

Figure 9: Annotation for file and directory inode structures in F2FS

tion. As an example, we annotate the `s_journal_inum` field in the Ext4 super block, shown in Figure 10, to indicate that it points to an `ext4_journal` type, in the file address space.

File systems may use the same pointer field to reference different types of metadata. The `when` argument is used to specify context-sensitive pointers. For example, Figure 11 shows that the Btrfs “tree of tree” root points to a B-tree leaf when the level of the tree is 0, or else it points to a B-tree node. In this case, two pointer annotations are needed to specify each of the pointed-to types and their `when` clauses. The `size` argument is useful when the structure that contains the pointer also stores the information about the size of the pointed-to structure. This may be the case when the pointed-to structure is variable-sized or a data block.

Spiffy supports implicit pointers with the `name-expr` argument, which names a pointer and specifies an expression for computing the address value. For example, Figure 10 shows that we added an implicit field to the end of the Ext4 super block, because it does not have a pointer field to the block group descriptor table. The descriptor table is located at block 2 if the block size is 1024 bytes, or block 1 for every other block size.

The `FIELD` annotation is similar to a pointer, but it is used to specify offset fields that reference a structure within the *same* container. Unlike a file system pointer, a field access does not require fetching data from disk, and hence it does not require an address space.

```

FSSUPER(name=sb, location=1024)
    ext4_super_block {
        __le32 s_blocks_count;      // # of blocks
        __le32 s_log_block_size;   // block size
        __le32 s_blocks_per_group; // blocks per group
        __le16 s_inode_size;       // size of inode
        ...
        /* pointer to journal in file address space */
        POINTER(aspc=file, type=ext4_journal)
        __le32 s_journal_inum;
        ...
        /* implicit pointer to group descriptors */
        POINTER(name=s_block_group_desc, aspc=block,
                type=ext4_group_desc_table,
                expr=s_log_block_size ? 1 : 2);
    };

```

Figure 10: Annotated Ext4 super block.

```

ADDRSPACE(name=raid);
FSSUPER(name=sb, location=0x10000)
    btrfs_super_block {
        ...
        POINTER(aspc=raid, type=struct btrfs_node,
                when=self.root_level > 0)
        POINTER(aspc=raid, type=struct btrfs_leaf,
                when=self.root_level == 0)
        __le64 root;
        ...
        u8 root_level; /* depth of root tree */
        ...
    } __attribute__((__packed__));

```

Figure 11: Annotated Btrfs super block.

The `ADDRSPACE` annotation specifies an address space for a pointer type. Figure 11 shows that the Btrfs pointers have a `raid` address type. Later, in § 5, we describe how the annotation developer implements this annotation.

VECTOR The `VECTOR` annotation helps specify variable-length arrays of structures. It can be placed inside or outside structure definitions. When placed inside, it defines an implicit field of a structure. When placed outside, it defines a new type, such as the `ext4_dir_block` structure in Figure 12. The size of the vector can be specified using any of the `count`, `size` or `sentinel` arguments. The `size` argument is useful when the elements are variable-sized and that the number of elements cannot be easily deduced. The `sentinel` argument specifies a boolean condition for determining the last element of a vector. All combinations of the three arguments are valid, and parsing ends as soon as one of the stopping conditions are met. Vector types have access units but the compiler can automatically deduce this information based on the access units of their elements. A

```

FSSTRUCT(name=eh) ext4_extent_header {
    __u16 eh_magic, eh_entries;
    __u16 eh_max, eh_depth;
    __u32 eh_generation;
    CHECK(expr=self.eh_magic == EXT4_EXT_MAGIC);
};

FSSTRUCT(size=BLOCK_SIZE) ext4_extent_leaf {
    struct ext4_extent_header eb_hdr;
    FIELD(count=self.eb_hdr.eh_entries)
    struct ext4_extent eb_extent[];
};

VECTOR(name=ext4_dir_block, size=BLOCK_SIZE,
       type=struct ext4_dir_entry);

```

Figure 12: Annotations for Ext4 extent header and leaf, and Ext4 directory block

vector that contains objects is a container (e.g., an inode block), and a vector that contains containers or extents is an extent (e.g., inode table).

CHECK The CHECK annotation allows specification of arbitrary constraints associated with a structure. These checks are performed both after parsing a structure, and before serializing it. The annotation acts as an assertion, which upon failure, results in a parsing or a serialization error. Figure 12 shows an example where the CHECK annotation is used to verify that the extent header contains the correct magic number.

A.1 Ext4

The Linux Ext4 file system is the most popular Linux file system. Unlike its predecessor Ext3, it uses extent-based allocation instead of block-based allocation, which significantly reduces metadata block usage for contiguous allocations. We have modified and added some Ext4 data structures so that they can be specified correctly. For backward compatibility, the Ext4 developers decided to leave the `i_block` field of the inode structure definition alone, although the space it occupies is now used for an extent tree. We redefined an Ext4 inode so that it now properly contains a extent header followed by four extent entries. We also support Ext3’s block-based allocation scheme, which is not shown here for brevity. We also added a definition for the extent leaf blocks, shown in Figure 12, which was omitted in the original header file.

A.2 Btrfs

Btrfs is a copy-on-write file system that stores in data structures in a number of B-trees [23]. Each B-tree uses two types of containers, an internal node that contains a sorted list of key-pointer pairs, and a leaf node that

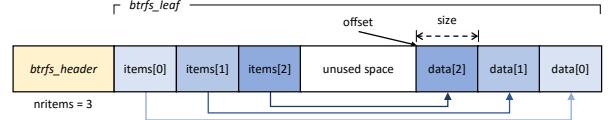


Figure 13: Btrfs leaf node layout.

```

FSSTRUCT(unit=container,
         size=sb.leafsize) btrfs_leaf {
    struct btrfs_header header;
    VECTOR(name=items, type=struct btrfs_item,
           count=self.header.nritems)
};

#define METADATA_LOCATION \
    (sizeof(struct btrfs_header)+self.offset)
FSSTRUCT() btrfs_item {
    struct btrfs_disk_key key;
    __le32 offset, size;
    FIELD(name=metadata, size=self.size,
          type=struct btrfs_file_extent_item,
          expr=METADATA_LOCATION,
          when=self.key.type ==
              BTRFS_EXTENT_DATA_KEY)
    FIELD(name=metadata, size=self.size,
          type=struct btrfs_inode_item,
          expr=METADATA_LOCATION,
          when=self.key.type ==
              BTRFS_INODE_ITEM_KEY);
/* followed by 15 more implicit fields,
   each with a different type */
};

```

Figure 14: Btrfs leaf node and item structure.

contains a set of keys and their associated metadata objects. The internal node structure is relatively simple, so we describe the annotation for a leaf node. Btrfs places all of the file system’s metadata objects (e.g., inode, directory entries) in reverse order, starting from the end of the B-tree leaf block, as shown in Figure 13. For each metadata object, there is a corresponding `btrfs_item` object that stores the offset and size of the metadata object. For example, `items[0]` stores the offset and size for `data[0]`.

Figure 14 shows the annotated Btrfs leaf node (`btrfs_leaf`), containing a vector of `btrfs_item` structures. The `btrfs_item` structure defines implicit fields with the `FIELD` annotation. These fields use the `when` clause to point to all the different types of metadata objects that can be stored in a leaf object. The `offset` field is an offset to a metadata object from the end of the `header` field of `btrfs_leaf`, and so we must add this value to obtain the offset from the beginning of the container (the leaf node).

```

typedef FSSTRUCT(name=cphdr, rank=container,
    base=struct f2fs_checkpoint) {
    POINTER(repr=block, type=f2fs_orphan_blocks,
        expr=$self.id + 1,
        when=self.ckpt_flags & CP_ORPHAN_PRESENT_FLAG);
    ...
} f2fs_checkpoint_header;

```

Figure 15: F2FS checkpoint header annotations.

A.3 F2FS

F2FS is a relatively new, log-structured file system optimized for NAND flash storage devices. Its on-disk layout is partitioned into fixed-sized segments composed of a set of contiguous blocks, with each segment sized in units of the SSD’s erase block size to minimize wear. The file system contains five static metadata areas, and one main area for data blocks and dynamically allocated metadata, such as the inode shown in Figure 9. The static metadata area consists of a pair of checkpoint packs, as shown in Figure 2, and various lookup tables for space and pointer management.

A unique challenge with annotating F2FS is its use of heterogeneous extents, i.e., extents with different types of metadata blocks. F2FS has a super block to the first checkpoint pack, shown in Figure 2. The remaining metadata blocks must be referenced by using implicit pointers. However, the block addresses of these metadata blocks depend on the address of the checkpoint header. Therefore, we use a special variable, `$self`, to allow implicit pointers to specify metadata blocks that exist at certain block offsets from the current container. Figure 15 shows the annotation for the implicit pointer that points to a vector of orphan blocks.