# Spiffy: Enabling File-System Aware Storage Applications

16

KUEI SUN, DANIEL FRYER, RUSSELL WANG, SAGAR PATEL, JOSEPH CHU, MATTHEW LAKIER, ANGELA DEMKE BROWN, and ASHVIN GOEL, University of Toronto, Canada

Many file-system applications such as defragmentation tools, file-system checkers, or data recovery tools, operate at the storage layer. Today, developers of these file-system aware storage applications require detailed knowledge of the file-system format, which requires significant 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, an online storage layer cache that preferentially caches files for certain users, and a runtime file-system checker. Our experiments show that applications built with the Spiffy library for accessing file-system metadata can achieve good performance and are robust against file-system corruption errors.

CCS Concepts: • **General and reference → Reliability**; • **Software and its engineering → File systems management**; **Compilers**; **Source code generation**; **Domain specific languages**;

Additional Key Words and Phrases: Annotation language, metadata parsing and serialization, file-system traversal, robustness, generic file-system aware applications, Ext4, Btrfs, F2FS

**ACM Reference format:**
Kuei Sun, Daniel Fryer, Russell Wang, Sagar Patel, Joseph Chu, Matthew Lakier, Angela Demke Brown, and Ashvin Goel. 2020. Spiffy: Enabling File-System Aware Storage Applications. *ACM Trans. Storage* 16, 3, Article 16 (August 2020), 39 pages.
https://doi.org/10.1145/3386368

This work was supported by NSERC Discovery.

Authors' addresses: K. Sun, D. Fryer, R. Wang, S. Patel, J. Chu, M. Lakier, A. D. Brown, and A. Goel, University of Toronto, The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Room SFB540, Ontario M5S 3G4, Toronto, Canada; emails: kuei.sun@mail.utoronto.ca, dfryer@cs.toronto.edu, russell.wang@mail.utoronto.ca, sagu.patel@mail.utoronto.ca, josephchu21@gmail.com, matthew.lakier@mail.utoronto.ca, demke@cs.toronto.edu, ashvin@eecg.toronto.edu.

1553-3077/2020/08-ART16 $15.00
https://doi.org/10.1145/3386368

## 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.

Offline tools operate on the file-system image when the file system is not being used. Examples of such tools include a file-system checker that traverses the file-system image to check the consistency of its metadata [23], and a data recovery tool that helps recover deleted files [5].

Online applications operate at the storage or block layer while the file system is in use. These 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). Online 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 [24] improve performance by preferentially caching blocks that contain file-system metadata or the data of small files. I/O shepherding [16] improves reliability by using file structure information to implement checksumming and replication. Similarly, Recon [11] improves reliability by verifying the consistency of file-system metadata at the storage layer.

Today, developers of both offline and online 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 [43]. Some storage applications reduce the amount of file-system-specific code in their implementation by modifying their target file system and operating system [16, 24]. 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. We chose this approach because commonly deployed, local file systems are written in C, and our annotation-based approach allows reusing existing data structures in C, compared to writing and maintaining the complete data-structure specification in a separate 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 an application developer to write actions for different file-system metadata structures, invoking file-system-specific or generic code as needed, for their

[1]**Sp**ecifying and **I**nterpreting the **F**ormat of **F**ilesy**s**tems.

Table 1. Example File-System Aware Storage Applications

| Storage Applications | Category | Purpose |
|---|---|---|
| Differentiated services [24] | online | performance |
| Defragmentation tool | either | |
| File-system checker [17] | either | reliability |
| Data recovery tool [5] | offline | |
| IO shepherding [16] | online | |
| Runtime verification [11] | online | |
| File-system conversion tool | offline | administrative |
| Partition editor [15] | offline | |
| Type-specific corruption [2] | offline | debugging |
| Metadata dump tool | offline | |

Offline applications have exclusive access to the file system; online applications operate on an in-use file system.

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

Fig. 1. Example of a structure definition for file-system metadata.

offline or online application. For offline applications, we support both reading and writing file-system metadata. However, for online applications, we currently only support reading metadata (e.g., differentiated storage services [24] or Recon [11]) but not modifying metadata (e.g., 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 [22] 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 in Figure 1. The naming convention suggests that this field is a pointer but that cannot be deduced from the structure definition because the information is embedded in file-system code.

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. This field must be accessed before an inode block can be interpreted correctly. 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, many applications need to identify objects in the file system, but these identities are type-specific and not available in current data-structure specifications. For example, suppose a file-system corruption tool needs to target a specific inode object, such as the root directory inode. Ideally, the corruption tool would allow specifying an inode by its unique id, i.e., its inode number. However, the relationship between inode numbers and their corresponding inode objects is not specified as part of the file-system structure definition. These challenges are not addressed by existing specification tools, as discussed in Section 8.

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 Figure 1 example, 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. Finally, annotations can be used to specify the identity of metadata structures so that applications can operate at object granularity, with the ability to locate and compare specific objects, or their different versions.

Together, these Spiffy features have allowed us to annotate three widely deployed file systems with very different metadata structures.

*Ext4.* The Extended file systems (ext) are a group of update-in-place file systems. At the time of writing, 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.

*Btrfs.* The B-tree file system is a copy-on-write file system that stores its metadata in a number of B-trees [28]. 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 contains a set of keys and their associated file-system metadata objects.

*F2FS.* The Flash-Friendly file system [21] 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.

We have implemented six applications that are designed to work across file systems. Four of them are offline applications: a file-system dump tool, a file-system corruption tool, a free space display tool, and a file-system converter. The other two are online applications: a storage layer service that preferentially caches data for specific users, and a runtime file-system checker based on Recon [11].

The rest of the article is organized as follows. In Section 2, we motivate the need for our approach by describing various parsing-related bugs in file-system applications. Section 3 presents the core concepts that underlie the design of the annotation language and the library API. Section 4 shows our file-system annotation language with examples of annotated structures for the Ext4, Btrfs, and F2FS file systems. Section 5 describes the applications that we have implemented using the generated library. Section 6 describes the implementation of our system, and Section 7 evaluates our approach in terms of programming effort, robustness, and performance. We present related work in Section 8 and discuss our conclusions in Section 9.

Table 2. Bugs Due to Incorrect Parsing of File-System Formats

| | 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 | #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 file systems | 2015-05 |
| 5 | e2fsck | Ext4 | #1187032 - missing first_meta_bg boundary check leading to heap buffer overflow | 2014-08 |
| 6 | e2fsck | Ext4 | #1768556 - crafted ext4 partition leads to out-of-bounds write | 2019-12 |
| 7 | btrfsck | Btrfs | #104141 - Malformed input causing crash/floating point exception in btrfsck | 2015-10 |
| 8 | btrfsck | Btrfs | #59541 - Btrfsck reports free space cache errors when using skinny extents | 2013-06 |

## 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, as shown 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 Master File Table (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 1,024 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 was Linux. Consequently, the file system becomes corrupt for Hurd and possibly other OSs.
(5) Meta block groups were introduced in Ext3 to increase the maximum file-system size from 256 TB to 512 PB for a file system using a 4 KB block size. When the feature was first introduced, the libext2fs library failed to perform a boundary check on the number of block group descriptor blocks, which could lead to a buffer overflow if the super block field `s_first_meta_bg` is corrupted.
(6) Ext4 v1.42 introduced a new feature that tracks usage statistics for implementing quotas on the file system. However, several sanity checks were missing when attempting to read usage information from the quota files. As such, a corrupted quota file could cause a crash, or worse yet, cause an out-of-bounds write on the heap that can allow arbitrary code execution.

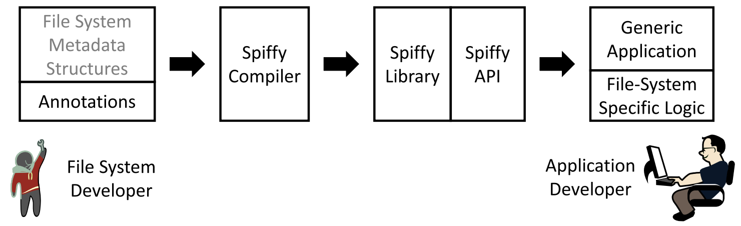

Fig. 2. Developing file-system aware storage applications with Spiffy.

(7) A fuzzer [44] 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.
(8) When the skinny metadata feature was added to Btrfs, the developers neglected to also patch Btrfsck, resulting in false error reports.

The common theme among all these bugs is that (1) they occur because developers may lack a detailed understanding of the file-system format and its evolution; (2) they can cause serious data loss or corruption; (3) most of these bugs were fixed in less than five lines of code; and (4) it is difficult to keep all relevant applications up-to-date with changing file-system formats. 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 corrupt 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 and identification 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 avoiding crash failures and reducing the chance of corruption propagation. Identification enables applications that require looking up or comparing specific file-system structures and their versions.

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.

Figure 2 shows how Spiffy applications are developed. After a file-system developer annotates his or her file system's data structures, we use the Spiffy compiler to parse the annotated structure definitions 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 still 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

```
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 */
};
```

Fig. 3. Ext4 directory entry structure definition.

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 [24] 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 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 [22], so there is minimal cost for maintaining annotations.

## 3.1 Designing Annotations

Our annotation language provides the ability to specify the type and identity of file-system data structures, and to check constraints on them.

*3.1.1 Specifying Types.* Next, we describe several key concepts that form the basis for specifying the type of file-system structures.

*File-System Pointers.* In a file system, pointers connect the metadata structures. However, they are not well specified in C data-structure definitions, as explained in Section 1. Unlike an in-memory pointer whose value is always interpreted as the in-memory address of the pointed-to data, interpreting a file-system pointer may involve multiple layers of translation. For example, 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 (see Section 6.2 for more detail).

*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 3. 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, and so forth. To support these dependencies, we need to *name* these structures. For example, the Spiffy expression `sb.s_inode_size` helps determine the size of an inode object, where `sb` is the name assigned to the super block.

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

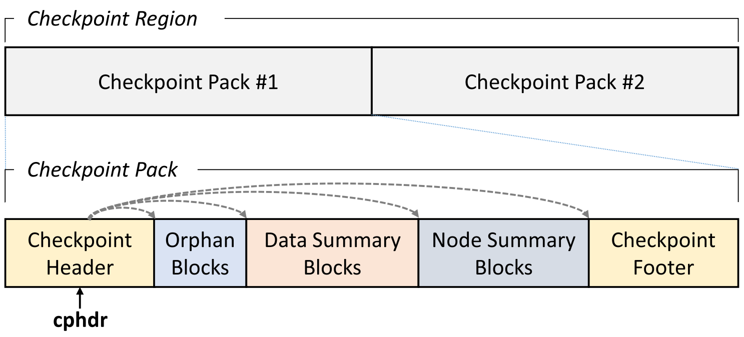

Fig. 4. Each F2FS checkpoint pack contains a header followed by a variable number of orphan blocks.

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 4. The number of orphan blocks in an F2FS checkpoint pack is determined by a field in the checkpoint header. Our naming mechanism ensures correct reference to the associated checkpoint header when the field is accessed.

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 4, when we need to reference the checkpoint header (`cphdr` in the figure) while identifying the orphan blocks in the checkpoint pack, the name resolution mechanism can unambiguously determine that it is referring to its parent checkpoint header. We use reference counting to ensure that a referenced structure is valid in memory when it needs to be accessed, which also avoids excessive copying.

*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, which 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 Ext4 is implicitly the block or blocks that immediately follow 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.

*3.1.2 Specifying Identity.* So far, we have described the design of annotations for specifying the type of the metadata structures. However, some file-system aware applications require the ability to identify objects even if their locations (or contents) change over time. For example, a Btrfs inode object is placed inside a B-tree leaf node block. During its lifetime, the inode object can be moved to a different offset within the block, or to another block.

We allow file-system developers to declare an *identity* expression for each metadata structure so that at runtime, each instance of the structure will be assigned a unique type-specific identity value by evaluating the identity expression. We use existing identity definition whenever possible, e.g., inode numbers for inode structures, Btrfs keys for B-tree items, and so forth. Otherwise, we use an expression that logically identifies the objects, e.g., the index of the block group descriptor structure in Ext4.

Once the identity for a given type of object has been specified, it can be used to locate an object of that type, compare different versions of the same object, sort a group of objects, or check if a group of objects are placed in correct order. To do so, Spiffy supports relational operators (e.g., equality and less than operators) for object identity. This logical identity-based approach is different from physical content equality.

While there are multiple ways to define a unique identifier for an object, there is often one natural identifier for each type of object. For example, since many file systems implement the virtual file-system (VFS) interface, they must use inode numbers as an identifier for inode structures. Although it is possible to specify the identity of an inode structure by other means, such as its byte offset from the start of disk (when inodes are located in static locations on disk), this value is much less meaningful within the context of a generic file-system tool.

To specify identity correctly, the file-system developer must know the identifier that makes an object unique. Spiffy currently makes no assumptions or checks on whether the identity specification will generate unique identity, since doing so may require scanning the entire file system.

The Spiffy identity specification allows locating specific objects of a given type. Currently, we enumerate all objects of a given type and then filter a specific object based on its identity. Supporting an efficient lookup in general will require specifying a data-structure-specific search algorithm (e.g., a B-tree search in Btrfs), but this is currently not supported in our annotations. In some simple cases, an alternative is to generate an efficient lookup based on solving constraints on the identity specification. For example, the identity of an Ext4 inode is specified by indexing into the block group descriptor table and then indexing into the corresponding inode table, as shown in Figure 8 on page 14. Thus, given an inode identity value, the index values for the two tables can be obtained using a method similar to a two-level page table lookup.

*3.1.3 Checking Constraints.* 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, #5, #6, and #7 from Section 2.

The set of valid addresses for a metadata block 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 this constraint for metadata blocks, Spiffy can verify that the address assigned to newly allocated metadata is within the correct bounds before the metadata is persisted to disk. Similarly, some data structures need to be placed in a sorted order. This constraint can be specified by using object identities, as shown in Section 4.4.

## 3.2 The Spiffy API

Table 3 shows the core API for building Spiffy applications. The API consists of three sets of functions. The Spiffy file-system library functions are automatically generated by Spiffy based on the

Table 3. Spiffy C++ Library API

| 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* |
| | `int compare(Entity & e, Visitor & v)` | allows *v* to process the difference between *this* and *e* |
| 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)`<br>`Entity * fetch_super()` | instantiates a new file-system object<br>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 |
| | `int diff(Entity * e, Field * a, Field * b)` | callback function during *compare* invocation |

annotated file-system data structures. The second set of functions need to be implemented by file-systems developers and are reusable across different applications. The third set of functions are written by the application programmer for implementing application and file-system-specific logic.

The Spiffy library uses the visitor pattern [12], 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 file-system metadata structures. Spiffy classifies these structures as *containers*, *objects*, or *extents*. A container is an addressable structure with at least one pointer annotation pointing to it. For example, an F2FS checkpoint block is a container since the super block points to it. A container is sized at the granularity of file-system blocks, i.e., it consists of one or more consecutive file-system blocks. A container is also a unit of disk access, i.e., it is loaded from and stored on disk in its entirety. An object is a non-addressable structure that does not have any direct pointers to it, and it lies within a container. An extent is a type of container that contains a vector of objects, containers, or other extents. Like a container, it is addressable, but its elements are loaded on demand. For example, the inode table is an extent, and the inode blocks of the table are loaded as inodes are accessed.

The `process_pointers` function invokes the `visit` function of an application-defined `Visitor` class on each pointer within the entity. The `process_by_type` function allows visiting a

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

Fig. 5. Address structure to locate container on disk.

```
1  Entity * IBlockPtr::fetch() {
2    IBlock * ib;
3    Address & addr = this->address;
4    char * buf = new char[addr.size];
5    this->fs.io.read(addr, buf);
6    ib = new IBlock(this->fs, addr, this->path);
7    ib->parse(buf, addr.size);
8    return ib;
9  }
```

Fig. 6. Example of a generated `fetch` function. `IBlockPtr` is a subclass of `Pointer`.

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 type id of an inode structure[3] as an argument results in visiting all inodes in the file system. The `compare` function allows comparing two entities of the same type. The application must implement the `diff` function of the `Visitor` class to process the differences between the two entities. For each field that is different, the `diff` function is invoked.

In the Spiffy API, every container has an associated address that allows it to be accessed from disk. Figure 5 shows the format of an address, consisting of an address space, an identifier, an offset within the address space, and the size of the container. The offset field is used when a container is part of an extent, which is used in the `read()` implementation.

The `Pointer` class stores the address of a container, and its `fetch` function reads the pointed-to container from disk. Figure 6 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. On line 6, 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 shown in Figure 7 serializes a container by invoking nested serialization on its fields. Then, on line 6, it invokes the `alloc` function for newly created metadata, or when existing

[3]All annotated structures are given a unique type id in the generated Spiffy library.

```
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);
}
```

Fig. 7. Abbreviated version of the save function.

Table 4. List of Properties of an Entity that Can Be Introspected

| Property | Description |
|---|---|
| `index` | index of element (0 if not part of an array) |
| `name` | name of field (blank if not a field) |
| `type` | name of type (in string) |
| `addr` | address of container or extents |
| `size` | the actual size of the entity |
| `traits` | traits of entity (e.g., integral, array) |
| `id` | identity of entity |
| `blocksize` | block size of file system (super block only) |

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.

Spiffy supports type introspection, which allows the programmer to write generic code for operating on file-system metadata. Table 4 shows a list of properties that each `Entity` base class implements. These can be used in annotations or in Spiffy applications. For example, in Figure 17 on page 20, the type of the input Entity *e* is printed.

## 3.3 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.

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

Our storage-layer online applications read file-system metadata but do not modify it. We are currently exploring modifying file-system metadata at the storage layer using Spiffy. We expect this will require an infrastructure similar to IO shepherding [16] to support transactions and allocation.

Table 5. Spiffy File-System Annotations

| Keyword | Description | Arguments | Meaning |
|---|---|---|---|
| FSSTRUCT | File system structure | name=IDENT | Name of structure for cross referencing |
| | | ident=EXPR | Identity expression for the structure |
| FSSUPER | File system super block | base=TYPE,when=BOOL | Structure inherits base when condition is true |
| | | size=INT | Size of the structure |
| | | location=INT,BOOL | Super block offset or placement constraint |
| | | blocksize=INT | Block size of the file system (FSSUPER only) |
| POINTER | Field is a pointer to a container | aspc=IDENT | Name of an address space type |
| | | type=TYPE | Type of the referenced structure |
| OFFSET | Field is an offset to an object within container | when=BOOL | Pointer/offset valid when condition is true |
| | | size=INT | Size of the referenced metadata |
| | | name=IDENT,expr=INT | Name of an implicit pointer, its expression |
| ADDRSPACE | An address space | name=IDENT | Name of the address space type |
| VECTOR | Defines a vector field | name=IDENT | Name of the vector/extent |
| | | type=TYPE | Structure type of the contained elements |
| EXTENT | Defines an extent type | count=INT | Number of elements in vector/extent |
| | | size=INT | Size of vector/extent, in bytes |
| | | sentinel=BOOL | Sentinel value specifying end of vector/extent |
| CHECK | Constraint check | expr=BOOL | Condition for the structure's correctness |

IDENT is a valid C identifier. TYPE is the type name of a structure, vector, or extent type. BOOL, INT, EXPR are syntactically valid, dynamically scoped, C expressions. BOOL and INT evaluate to a Boolean and integer type, while EXPR can also be a string or tuple type. For each group of annotations, the arguments are applicable to all keywords within the group unless otherwise specified.

## 4 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 simply 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 5 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.

### 4.1 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 the Spiffy 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. The `blocksize` argument specifies the block size of the file system. The `name` argument is used by a descendant to

```
FSSTRUCT(name=in, size=sb.s_inode_size,
        ident=$(gd).index*sb.s_inodes_per_group + $self.index + 1) ext4_inode {
  /* fields of ex4_inode structures */
};
```

Fig. 8. Ext4 inode structure annotation.

```
FSSTRUCT(ident=($(in).id, self.name), size=self.rec_len) ext4_dir_entry {
  __le32  inode;
  __le16  rec_len;   /* size of structure */
  __le16  name_len;
  VECTOR(name=name, type=char, size=name_len);
};
EXTENT(name=ext4_dir_block, size=$(sb).blocksize, type=struct ext4_dir_entry);
```

Fig. 9. Ext4 directory entry and directory block annotations.

reference this structure (see Section 3.1). For FSSTRUCT, the location argument optionally specifies its placement constraint, as described in Section 3.1.3.

The ident argument specifies the identity expression of the structure. By default, a structure has no identity. To specify identity, one can reference fields of the same or other structures, as well as the properties of a structure (see Table 4). For example, in Figure 8, the identity of an inode is specified using the index property of the parent block group descriptor. To differentiate property from regular fields, the $ operator is used.[4]

The identity of a metadata object may have multiple constituents. For example, Figure 9 shows the identity specification for the Ext4 directory entry structure. Since it is possible to have two entries with the same file name across two different directories, the name is not a sufficiently unique identifier. Therefore, the identity is specified as a tuple, where $(in).id is the identity of the associated inode and self.name is the file name.

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 10 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 size argument in the FSSTRUCT definition of f2fs_inode references 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. Note $(self).addr refers to the address of the container (see Table 4 and Figure 5).

## 4.2 POINTER, OFFSET, ADDRSPACE

The POINTER annotation is used to specify the address type and the pointed-to type of a pointer. It allows fetching a container from disk and parsing it using the correct type information. As an example, we annotate the s_journal_inum field in the Ext4 super block, shown in Figure 11, to indicate that it points to an ext4_journal type in the *file* address space.

[4]The syntax is inspired by JQuery, which is a JavaScript library.

```
FSSTRUCT(size=$(sb).blocksize, location=$(self).addr.id >= sb.main_blkaddr) 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;
```

Fig. 10. Annotations for file and directory inode structures in F2FS.

```
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=self.s_log_block_size ? 1 : 2);
};
```

Fig. 11. Annotated Ext4 super block.

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 12 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` expression. The `size` argument in the pointer annotation 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 11 shows that we

```
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__));
```

Fig. 12. Annotated Btrfs super block.

```
FSSTRUCT() 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=$(sb).blocksize) ext4_extent_leaf {
  struct ext4_extent_header eb_hdr;
  VECTOR(name=eb_extent, type=struct ext4_extent, count=self.eb_hdr.eh_entries);
};
```

Fig. 13. Annotations for Ext4 extent header and leaf structures.

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 1,024 bytes, or block 1 for every other block size.

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

The ADDRSPACE annotation specifies an address space for a pointer type. Figure 12 shows that the Btrfs pointers have a raid address type. In Section 6, we describe how the annotation developer implements this annotation.

## 4.3 VECTOR, EXTENT

The VECTOR and EXTENT annotations help specify variable-length arrays of structures. The VECTOR annotation is placed inside structures, and it defines an implicit array field of a structure, such as the name field in the Ext4 directory structure shown in Figure 9, or the eb_extent field in the Ext4 extent leaf shown in Figure 13.

The EXTENT annotation is placed outside a structure definition. It defines a new type, such as the ext4_dir_block structure in Figure 9. The EXTENT annotation generates a Spiffy extent containing an array of objects, containers, or other extents (see Section 3.2).

The size of the vector or extent can be specified using any of the count, size, or sentinel arguments. The size argument is useful when the elements are variable-sized and the number of elements cannot be easily deduced. The sentinel argument specifies a Boolean condition for

```
#define METADATA_LOCATION (sizeof(struct btrfs_header)+self.offset)
FSSTRUCT(ident=self.key) btrfs_item {
  struct btrfs_disk_key key;
  __le32 offset, size;
  OFFSET(name=metadata, size=self.size, type=struct btrfs_file_extent_item,
        expr=METADATA_LOCATION, when=self.key.type == BTRFS_EXTENT_DATA_KEY)
  OFFSET(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 */
};

FSSTRUCT(size=sb.leafsize) btrfs_leaf {
  struct btrfs_header header;
  VECTOR(name=items, type=struct btrfs_item, count=self.header.nritems);
  CHECK(expr=for i in (0, self.header.nritems-1)
            $(self.items[i]).id < $(self.items[i+1]).id);
}
```

Fig. 14. Btrfs B-tree leaf structure annotation.

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.

## 4.4 CHECK

The CHECK annotation allows specifying arbitrary constraints on a structure. These checks are performed both after parsing a structure, and before serializing it. This annotation acts as an assertion, which upon failure, results in a parsing or a serialization error. Figure 13 shows an example where the CHECK annotation is used to verify that the Ext4 extent header contains the correct magic number.

In file systems that use sorted data structures, such as B-trees, metadata objects must be placed in a particular order to guarantee that they can be found using the intended search algorithm for the data structure. Therefore, Spiffy can be used to detect any ordering-related corruption. As an example, Btrfs uses B-trees to store all of its metadata objects, and B-tree items must be sorted in monotonically increasing order, based on their identity (i.e., their associated `btrfs_key`). Figure 14 shows the annotated leaf node data-structure definition. The CHECK annotation verifies correct ordering of elements in the array using the identity of `btrfs_items`. This identity is specified as the key field in this structure. The CHECK expression loops through the array and checks that the identity of the current element is strictly less than the next element. This check is added to the generated parsing and serialization routines, allowing developers using Spiffy to detect ordering violations before accessing this data structure.

## 4.5 Ext4

We have modified and added some Ext4 data structures so that they can 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 correctly defines an extent header followed by four extent entries. We also support Ext3's block-based allocation scheme, which is not shown here for brevity.

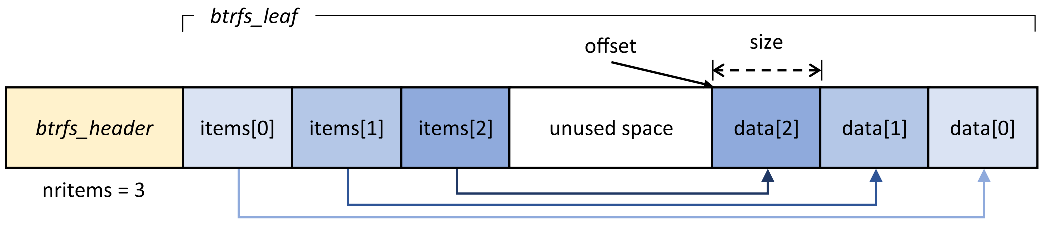

Fig. 15. Btrfs leaf node layout.

We also added a definition for the extent leaf blocks, shown in Figure 13, which was omitted in the original header file.

We show the structure annotation for the Ext4 inode in Figure 8. The `name` argument specifies the name of the structure for cross referencing. For example, Figure 9 shows that this name is used to specify the identity of directory entries. The `size` argument specifies the size of the inode structure. The `ident` argument specifies the identity of the structure, which is its inode number. In this case, the inode number is calculated from its block group number and its index in the inode table. The block group number is obtained from the index of the parent block group descriptor structure, which is referenced through the name `gd`.

In Figure 9, we show the annotated Ext4 directory entry structure. This structure is uniquely identified by its associated inode and its file name (file names within the same directory must be unique). Therefore, its identity is specified as a tuple with two elements. The size of the directory entry structure is specified by the `rec_len` field.

## 4.6 Btrfs

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 15. 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 header and a vector of `btrfs_item` structures, which is defined as a flexible array member that contains `header.nritems`. This figure shows that the `btrfs_item` structure defines implicit offset fields with the `OFFSET` annotation. These fields use the `when` expression 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).

## 4.7 F2FS

F2FS contains five static metadata areas, and one main area for data blocks and dynamically allocated metadata, such as the inode shown in Figure 10. The static metadata area consists of a pair of checkpoint packs, as shown in Figure 4 on page 8, and various lookup tables for space and pointer management.

F2FS has a pointer in the super block to the first checkpoint pack. After that, all other blocks in the checkpoint region are accessed using pointer arithmetic in the code. In Figure 16, we show a partial annotation for the checkpoint header. We use an implicit pointer to point to a vector of orphan blocks, which are present when the `when` condition is true. The expression for the implicit pointer is calculated using the address of its container (i.e., the checkpoint header within the same

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

Fig. 16. F2FS checkpoint header annotations.

checkpoint pack). This expression shows that the orphan blocks are located one block after the start of the checkpoint header block, as shown in Figure 4.

## 4.8 Future Work

We designed Spiffy to be general, and we expect it would be straightforward to annotate popular file systems in other operating systems, such as the New Technology File System (NTFS) for the Windows operating system.

NTFS has a relatively simple metadata layout in the block address space because almost everything in NTFS is a file, including the inode table, known as a master file table (MFT). The file system starts with the super block (partition boot sector) as usual. It contains a block pointer[5] to the first block of the master file table, where the first inode (MFT record) stores the metadata on the extents allocated to the MFT. It is used to find subsequent blocks used by the MFT. Other metadata, including block allocation bitmap, journal, and volume information are also stored as files (a.k.a. metafiles) with a designated inode number.

To annotate NTFS, we would first declare a file address space, similar to the Ext4 journal's file address space. In the super block, we would add an implicit file address space pointer to the master file table. This is necessary to present the master file table as a contiguous entity. In similar fashion, we would annotate all other metafiles and reference them using implicit file address space pointers. The MFT record comes in two forms: resident (inlined data) vs. non-resident (extent-based allocation). We would annotate these two forms as context-sensitive types, with the `when` argument in the structure annotation.

## 5 FILE-SYSTEM APPLICATIONS

We have written six 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, a prioritized block-layer cache, and a runtime file-system checker. The first four applications operate offline, while the last two are online applications.

## 5.1 Offline Applications

Figure 17 shows a sample offline application built using the Spiffy API. Our actual offline applications are implemented using variations of the file traversal code shown in Figure 17.

This application prints the type of each metadata block in an Ext4 file system in depth-first order. The `Ext4IO` class on line 16 implements the block and the file address space, as described in Section 6.2. 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 on line 11 takes an entity as input, prints its type, and then invokes `process_pointers`, which calls the `PtrVisitor::visit` function on line 3 for every

[5]Confusingly named logical cluster number in NTFS terminology, where a cluster means a block.

```
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.$type() << 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();
  }
}
```

Fig. 17. Code for traversing and printing the types of all the metadata blocks in an Ext4 file system.

pointer in the entity. The `PtrVisitor::visit` function invokes `fetch`, which fetches the pointed-to entity from disk, and invokes `EntVisitor::visit` on it.

*5.1.1 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. 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.

*5.1.2 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. We currently support corrupting an object at random, or by identity. In the latter case, we iterate through all objects of the specified type until we find the object with the matching identity.

*5.1.3 Free Space Tool.* This tool shows file-system fragmentation by generating a bitmap of free blocks in the file system and plotting a histogram of the size of free extents. The tool retrieves the metadata structures that store free space information and processes them. This logic is implemented using `process_by_type` (see Table 3) and a file-system-specific `visit` function that processes all the retrieved metadata structures. The Ext4 implementation uses the Ext4 block bitmaps within the group descriptor table. The Btrfs implementation uses the extent start and size values from the extent items and metadata items (which include system extents) in the extent tree. The F2FS implementation uses the segment information table (SIT) from the latest valid checkpoint. It also uses the SIT entries in the checkpoint journal, recording them with higher precedence than the segment table. The code to traverse the file system and parse intermediate structures is provided by the Spiffy library.

*5.1.4 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,[6] 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 this version of the tool, a power failure during the last step would corrupt the source file system. A newer version of the tool supports failure atomicity through journaling [35].

Our tool 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 (so this area is not available for dynamic allocation). Then, Ext4 extents in this static 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.

## 5.2 Online Applications

Spiffy 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.

[6]The convert utility converts FAT32 to NTFS [37], and updating to iOS 10.3 upgrades the file system from HFS+ to APFS [38].

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 write request or 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.

We provide two types of runtime interpretation. The first type, which we call snapshot-based interpretation, caches only the latest version of blocks. This method has low memory overhead, but it suffers from occasional misclassification of block types due to the inability to detect deallocation of blocks. The second type, which we call differencing-based interpretation, caches both the previous and current versions of blocks so that it can detect deallocation of blocks by observing that a non-null pointer is set to null. We describe the implementation of these methods in more detail in Section 6. Next, we discuss their use in two online applications that we have implemented.

*5.2.1 Prioritized Block-Layer Cache.* We have implemented a file-system aware block-layer cache based on Bcache [26]. 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 [34]. 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 the snapshot-based interpretation module to identify the types of metadata blocks at the block layer, without requiring any modifications to the file system. This approach works for a caching application since a misclassification will not lead to incorrect operation.

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.

*5.2.2 Runtime File-System Checker.* Recon is a runtime file-system consistency checker that can protect the integrity of file-system metadata by checking the consistency of file-system update operations before they are committed to disk [11]. Recon intercepts IO requests from the file system and independently interprets the metadata read or written by the file system. Before a file-system transaction commits, it enumerates all the logical data-structure updates made by the file system and verifies their correctness against a set of consistency invariants. If a violation is detected, all checkpoint operations associated with the transaction are stopped to prevent corruption of the file system.

We refactored the original Recon code so that it performs generic metadata interpretation and differencing for all of our annotated file systems using the Spiffy library. Unlike our prioritized block-layer caching application, this code uses the differencing-based interpretation to track both the old and new versions of objects, guaranteeing accurate classification of block types. We compare two versions of a block (e.g. Block *A*) with pointers to learn whether a pointer is removed (i.e., from a valid address to null). If the change removes the last pointer to another block (e.g., Block *B*), then we can deduce that Block *B* has been freed.

This application uses object identities and the `compare` function (see Table 3 on page 10) to perform logical differencing of the old and new versions of an object. Then it uses the `diff` function to operate on the logical changes to a data-structure field that has been updated. We provide more details in Section 6.4.

## 6 IMPLEMENTATION

We implemented a compiler that parses Spiffy annotations. The compiler uses Python Lex-Yacc (PLY 3.4) [4] as its parser generator and lexical analyzer. The grammar, written in Yacc, is based on the ANSI C grammar. The compiler is invoked with a set of C header files containing the annotated data structures (e.g., `spiffy --name Ext4 ext4_fs.h`). It parses the annotations and the annotated structures, and ignores the rest of the source code. We verify that the Boolean and integer expressions used in annotations are syntactically correct by attempting to compile the expressions using a C++ compiler.

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 the Jinja2 [29] templating engine that is typically used for generating dynamic HTML content. The code generator works by processing template filters and tags in the source template files, and the output of the compiler is a pair of C++11 source and header files that can be used by applications. These files can be compiled as either a user space library or a part of a Linux kernel module. We linked our kernel 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 implements the `Entity` interface that is used by applications such as the simple file-system traversal application shown in Figure 17. Figure 18 shows an example structure for the Ext4 directory entry shown in Figure 9. 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 wrapper classes allow introspection so the class can access its properties (see Table 4 on page 12) and each field in the wrapped class can reference its containing structure. The generated library performs various types of error-checking operations. For example, the parsing of offset

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

Fig. 18. Wrapper class for Ext4 directory entry.

```
int BtrfsLeaf::parse(char * & buf, int & len) {
  int ret, i = 0;
  ret = header.parse(buf, len);
  if (ret < 0) return ret;
  ret = items.parse(buf, len);
  if (ret < 0) return ret;
  while (i < header.nritems-1) {
    if (!(items[i].$id() < items[i+1].$id()))
      return ERR_CORRUPT;
    i++;
  }
}
```

Fig. 19. Generated parsing code for the `btrfs_leaf` structure with the ordering constraint check inserted. The `$id`() function returns the identity of the object.

fields ensures that objects do not cross container boundaries, and similarly, all variable-sized objects fit within their containers. In Figure 19, we show a condensed version of the C++ parsing code that is generated from the `btrfs_leaf` annotation shown in Figure 14. The expression in the `CHECK` annotation is converted to equivalent C++ code that will return an error if the `items` array is not sorted correctly based on the identity of its elements. 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.

Our path-based name resolution mechanism resolves a name in the reverse order of the path from the super block. For example, suppose the path is $A \rightarrow B_1 \rightarrow C \rightarrow B_2 \rightarrow D$, where each symbol is a unique structure type, and $B_1$ and $B_2$ are separate instances of the same type. Structure $C$ would resolve $B$ as $B_1$, but $D$ would resolve $B$ as $B_2$, and not see $B_1$. However, it can use $C.B$ to access $B_1$. This mechanism is implemented by associating a path object and a parent entity with each entity. After a pointer is used to fetch and parse an entity, its path object is created by performing a shallow copy of the parent's path object, and appending a pointer to the current entity. The shallow copy increments entity reference counts, ensuring that the names in the path can be referenced correctly. When a name is not specified for a structure (in the `FSSTRUCT` annotation), the corresponding entity is not added to the path.

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.

## 6.1 Access Unit

Most 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) 16 KB, or four 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 support these types of operations by classifying data structures as objects, containers, or extents, as explained in Section 3.2. Containers are multiples of block sizes, and Spiffy loads and stores data at container granularity. Extents are vectors of objects or containers, and they are loaded on demand at container granularity when a container is accessed. One complication occurs when an extent consists of a vector of objects. In this case, there are no containers within the extent. We solve this problem by defining implicit containers (by default, of block size) that contain these objects, and we also check that the objects do not cross these container boundaries.

## 6.2 Address Spaces

Annotation developers must implement the `IO` interface shown in Table 3 for offline applications. The `read` function 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`. We show its use in Figure 6. The `write` function writes an in-memory data structure to disk, with the same parameters as `read`. It is used by the `save` function shown in Figure 7.

At minimum, Spiffy requires 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. Supporting other address spaces involves more work.

The Ext4 file address space implementation for the `Ext4IO` class (see line 16 on Figure 17, page 20) 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.

## 6.3 Snapshot-Based Interpretation

As described in Section 5.2, for online applications, metadata blocks are parsed as the requests are intercepted at runtime. When the I/O operation is a read, the block is parsed immediately since its type is already known from the mapping table. After parsing, the block can be immediately discarded.

However, 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

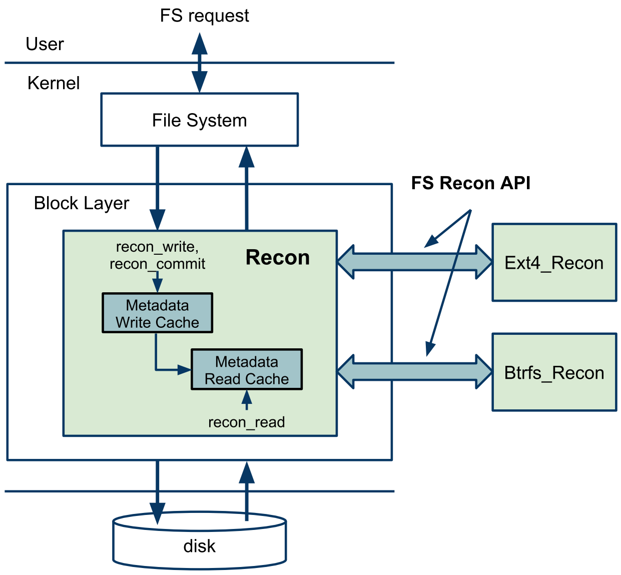

Fig. 20. Recon architecture (figure reproduced from [11]).

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.

## 6.4 Differencing-Based Interpretation

As shown in Figure 20, Recon [11] interposes between the file system and the underlying device and intercepts all IO requests. Recon performs runtime verification in three steps: metadata interpretation, type-specific differencing, and invariant checking. Since invariant checking is file-system specific, we focus on how we use the Spiffy libraries to build a generic framework for metadata interpretation and type-specific differencing.

A runtime checker needs to know about all the changes made to the file system during a transaction so that it can check their correctness before committing the transaction. To do so, Recon compares the updated state of the file system against its previous state. The updated state is available in the metadata write cache and the previous state is available in the metadata read cache, shown in Figure 20. While both of these caches contain file-system blocks, the comparison is performed

Table 6. Block Differencing Actions for Each Metadata Block

| | Not in write cache | In write cache |
|---|---|---|
| Not in read cache | ignore | objects created |
| In read cache, freed | objects deleted | objects compared |
| In read cache, allocated | ignore | objects compared |

```
def location_based_differencing(cache):
  for newblk in cache.write:
    oldblk = cache.read.find(newblk.blknr)
    if oldblk:
      compare(newblk, oldblk)
      oldblk.done = True
    else:
      created(newblk)
  for oldblk in cache.read:
    if not oldblk.done:     % not in write cache
      if oldblk.freed:
        deleted(oldblk)
```

Fig. 21. Block differencing pseudocode.

by interpreting the metadata blocks to generate logical objects, and then performing type-specific differencing for all objects that have been modified. This differencing generates *change records* for each object field that is modified. The change records are used for invariant checking.

To perform type-specific differencing, we use two methods: *block differencing*, or *set differencing*. Block differencing is more efficient and is used when all the objects in a block remain at their fixed locations in the block while they are alive (i.e., from allocation until deallocation). We call such blocks *location invariant*. We use block differencing for most Ext4 blocks since they are location invariant. Set differencing is used when objects in a block may move while being alive. For example, we use set differencing for Ext4 directory blocks since live directory entry objects may be moved within directory blocks. Similarly, we use set differencing for most Btrfs structures since they are allocated using copy-on-write, and so the updated and the previous versions of an object are placed at different locations. For F2FS, we use block differencing for the static metadata blocks, and set differencing for the dynamic data blocks that are written in a log-structured manner.

*6.4.1 Block Differencing.* Table 6 shows the different types of actions taken during block differencing, based on whether the block is in the read cache and the write cache. If a block is not in the write cache, then the block has not been updated, and so it does not need to be compared. However, the block may have been freed, and so all objects within the freed block are marked as deleted. When a block is in the write cache but not in the read cache, then the block is newly allocated, thus all the objects within the block are processed as newly created. If a block is in both the read and write cache, then we compare the two copies of the object. In the case when the block is freed, the write may have zeroed the block.

Figure 21 shows the pseudocode for the block differencing algorithm. The `compare` function takes two arrays of objects (in unsorted order) and compares the two versions of each object to generate change records. The `created` and `deleted` functions also generate change records but make no comparison since a created object has no previous state, and a deleted object has no next state.

*6.4.2 Location Invariant Blocks.* Set differencing is a general technique that will work for all blocks, but as mentioned earlier, block differencing is more efficient. To perform block differencing, we need to know the blocks that are location invariant. We use a heuristic based on the identity specification of objects to determine location invariant blocks. When the identity expression contains just the index property, as shown in Table 4 or immutable fields from the super block, then we assume that the object is location invariant. In particular, the identity expression must not contain any contents of file-system objects.

For example, the Ext4 inode identity, shown in Figure 8, is location invariant since it uses index information and a constant from the super block. However, the Ext4 directory entry structure shown in Figure 9 is not location invariant since its identity contains `self.name`, which is a field in the structure. Similarly, the `btrfs_item` shown in Figure 14 is not location invariant since its identity is the key field in the structure. A block is considered location invariant if it only contains location invariant objects, and all these objects are of the same type.

*6.4.3 Handling Temporary Objects.* While our approach allows generic differencing of file-system blocks, some file-system-specific code is required for handling the transaction mechanisms that file systems employ to maintain crash consistency. For example, in Ext4, all metadata writes are first logged into a journal, and then after the commit block is written to the journal, all the journal blocks are checkpointed to their final locations. However, a runtime file-system checker must check the invariants *before* the transaction commits.

The generic block differencing method described in Section 6.4.1 requires the physical block numbers in the read and the write cache to match (see line 3 in Figure 21). However, at the point when the file system commits, the blocks written to the journal have not been checkpointed yet, and thus there would not be a matching write cache entry for an updated block. As such, we need to map the blocks from their temporary locations in the journal to their final locations for performing the generic differencing. While the file system keeps this mapping in the journal since it needs to checkpoint the metadata blocks, this procedure cannot be generated from our annotations.

Thus, for each file system, we create a temporary mapping of blocks from their temporary locations to their final locations during metadata interpretation, and then use this mapping to create the final versions of the blocks in the write cache[7] (as if they have already been checkpointed) before performing the type-specific differencing. Next, we describe how we handle the transaction mechanism of each of the file systems that we support.

*Ext4.* In Ext4, metadata are journaled at the block granularity, which makes it simple to map each block to its final destination in the write cache. For example, suppose an inode block is placed in journal block $J$, and its final location is physical block $P$. Then we would insert block $J$ into the write cache as block $P$, which enables performing differencing correctly. This mapping information is obtained from the journal descriptor block. We ignore any metadata writes to blocks outside the journal, since those writes are part of the checkpointing process. Note that Recon only interprets metadata blocks and ignores all data blocks, so the data journaling mode of Ext4 does not affect the correctness of Recon's operations.

*Btrfs.* Btrfs implements copy-on-write semantics, which means all block writes go to a new but final location, and so no mapping is necessary. However, Btrfs uses a log tree, which is similar to a journal, to alleviate write amplification in fsync-heavy workloads. Currently, we have disabled this optimization for simplicity. To support the log tree, we would need to replay the items in the

[7]Note that this only affects in-memory blocks in the write cache. We do not alter updates to physical disk.

Table 7. Set Differencing Actions for Each Metadata Block

| | Not in write cache | In write cache |
|---|---|---|
| Not in read cache | ignore | add to new set |
| In read cache, freed | add to old set | add to old set |
| In read cache, allocated | ignore | add to both sets |

```
def set_based_differencing(cache, set):
  for newblk in cache.write:
    oldblk = cache.read.find(newblk.blknr)
    if not oldblk or not oldblk.freed:
      for obj in newblk:
        set.new.add(obj)
  for oldblk in cache.read:
    if oldblk.freed or cache.write.find(oldblk.blknr):
      for obj in oldblk:
        set.old.add(obj)
  compare(set.new.intersection(set.old), set.old.intersection(set.new))
  created(set.new.difference(set.old))
  deleted(set.old.difference(set.new))
```

Fig. 22. Set differencing pseudocode.

log and add them to the new set during the differencing step, before comparing the new and the old sets.

*F2FS.* F2FS is a log-structured file system designed for SSDs. It uses object-level journaling to mitigate write amplification on SSDs and to improve its performance. While Ext4 journals at block granularity and so the final version of the block is available in the journal, with F2FS, we need to replay the items in the journal to create the final version of the block. To do so, we create a copy of the old block, use the journaled object to overwrite the corresponding object in the copy, and then place the updated copy in the write cache. Our Ext4 approach of remapping blocks is an optimization of this general approach.

In addition, F2FS uses two physical blocks for each metadata block to implement checkpointing. It maintains a bitmap to indicate which physical block is active. Therefore, we also have to map a logical block address to one of its two physical blocks. For example, logical block address $L$ for a metadata block may be mapped to either physical block $P_a$ or $P_b$. When physical block $P_b$ is written, we perform a reverse map to find its associated logical block address so that we can cache it as block $L$ in the write cache. Meanwhile, block $L$ in the read cache would be mapped to physical block $P_a$. At commit time, we verify that the active bitmap does indeed toggle from $P_a$ to $P_b$.

*6.4.4 Set Differencing.* Table 7 shows the different types of actions taken during set differencing, based on whether the block is in the read cache and the write cache. These actions are similar to block differencing except that blocks with the same id are not directly compared, since they may not contain the same objects. Instead, newly created items are added to a *new* set, deleted items are added to an *old* set, and updated items are added to both the new and old sets, and then these items are compared.

Figure 22 shows the pseudocode for the set differencing algorithm. We use the blocks in the read and write caches to create the old and the new sets. This code assumes that the type of a

Table 8. File System Structure Annotation Effort

| File System | Ext4 | Btrfs | F2FS |
|---|---|---|---|
| Line Count | 491 | 556 | 462 |
| Annotated | 113 | 153 | 127 |
| Structures | 15+10+4 | 27+4+1 | 14+16+5 |
| Identities | 6 | 1 | 11 |

block does not switch within a transaction, a requirement for ensuring crash consistency in file systems [31]. In particular, a block must be freed in a previous transaction before it can be reused. As a result, we can look up blocks by the same id in the read and the write cache, since they will contain objects of the same type, even if they don't contain the same set of objects. For example, for a directory block in Ext4, its previous version must either be a directory block (belonging to the same directory) or a free block. Similarly, the next version must be a directory block (belonging to the same directory), or a free block. Our assumption holds for copy-on-write file systems trivially, since the previous and the next version of the same block will not be at the same location.

Once the new and the old sets are created, we perform the differencing. For objects that exist in both sets, we invoke the `compare` function with the new and the old versions of the objects. Note that when `set.new` intersects with `set.old`, the resulting set contains the new versions of objects that also exist in the old set, whereas when `set.old` intersects with `set.new`, the resulting set contains the old versions of the objects. The `created` and `deleted` functions are invoked for the newly created and deleted objects.

## 7 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, the file-system aware block-layer caching mechanism, and the runtime checker.

### 7.1 Annotation Effort

Table 8 shows the effort required to correctly annotate the Ext4, Btrfs, and F2FS file systems. The second row shows the number of lines of code of existing on-disk data-structure definitions in these file systems. The lines of code count were obtained using `cloc` [8] to eliminate comments and empty lines. The third row shows the number of annotation lines. This number is less than one-third of the total line count for all the file-system structure definition code.

The fourth row 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

Table 9. Lines of Code in C++ for Each Tool

| Tool Name | Generic | Ext4 | Btrfs | F2FS |
|---|---|---|---|---|
| Dump Tool | 565 | 50 | 45 | 37 |
| Metadata Corruptor | 455 | 28 | 28 | 20 |
| Free Space Tool | 271 | 76 | 77 | 194 |
| Conversion Tool | 504 | 218 (read) | | 1,760 (write) |
| Runtime Interpreter | 2,158 | 111 | 134 | |
| Runtime Differencing | 2,826 | 197 | 48 | 473 |

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, as described in Section 4.5.

The last row shows the number of structures that were annotated with identity to enable runtime consistency checking. Currently, we did not add identity to structures if they have no associated consistency invariants,[8] or if the structure is part of the transaction mechanism (e.g., Ext4 journal descriptor block or F2FS checkpoint block). In general, structures relating to inodes, directory entries, block allocation, and block mapping are typically checked by their respective file-system checkers, and thus have an associated identity. For Btrfs, since all file-system metadata are placed within the B-tree, they all share the same identity (i.e., `btrfs_key`).

## 7.2 Application Developer Effort

Table 9 summarizes the effort required to build each of our tools.

**Dump Tool:** The file-system dump tool includes a file-system independent XML writer module. For each file system, we specify their export options in 37 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 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 any file-system-specific code.

**Free Space Tool:** The file-system free space tool has a file-system independent component to traverse the file system and to plot histograms. File-system-specific parts are needed to process allocation metadata. F2FS requires more code due to the complex format of its block allocation information.

**Conversion Tool:** The Spiffy file-system conversion tool framework currently supports reading an existing Ext4 file system and converting it to an F2FS file system. In addition, 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` [40] library to parse Ext4 metadata from the source file system, and then manually write raw data to create an F2FS file system. The manual converter has 223 lines of Ext4 code, and 2,260 lines for the F2FS code. While the two converters have asimilar 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

[8]Spiffy automatically checks structural integrity [36] during parsing.

Table 10. Lines of Code for the Spiffy Runtime File-System Checker

| | Generic | Ext4 | Btrfs | F2FS |
|---|---|---|---|---|
| Journaling | - | 197 | 48 | 315 |
| Address Space | - | - | - | 158 |
| Interpretation | 905 | - | - | - |
| Differencing | 194 | - | - | - |
| Kernel Module | 1,727 | - | - | - |
| Manual | - | 2,099 | 2,897 | - |

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 10,518 lines of code. To implement prioritized caching, we added 289 lines to this code, which invoke our generic, snapshot-based metadata interpretation framework, listed as Runtime Interpreter in Table 9. This framework provides hooks to specify file-system-specific policies, which we have implemented for Ext4 and Btrfs. 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.

**Runtime Checker:** Table 10 breaks down the programming effort needed to build the runtime file-system checker using Spiffy with identity support (summarized in Table 9 under Runtime Differencing). We omit the file-system-specific code for invariant checking and show only the lines of code for building generic metadata interpretation and type-specific differencing. The kernel module code is also generic and implements a Linux device mapper used to intercept block IO from the file system to disk. It also contains top level logic and calls file-system-specific functions. For all three file systems, we need to write a small amount of code to handle the temporary objects created by their transactional mechanism. For Btrfs, while no block remapping is necessary, we still need to implement the Recon API [11].

In contrast, the manual versions of Recon require substantial boilerplate code for interpretation and differencing. The original, manual Ext4, and the Btrfs versions of Recon took a significant amount of development time and effort, and as a result, the F2FS manual implementation was never attempted. With Spiffy, the F2FS implementation required understanding its journal format, and implementing the `NID` address space to access the indirect blocks. The online implementation of this address space is similar in principle to the offline version described in Section 6.2.

## 7.3 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

Table 11. List of Segmentation Faults Found During Type-Specific Corruption Experiments

| 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 |

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 11 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 five 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. Although Spiffy is designed primarily to build new and robust file-system applications, it can also be used to build tools that expose bugs in existing applications. In other words, building the type-specific corruption tool with Spiffy helps make the tool itself robust and also makes it easier to support multiple file systems.

## 7.4 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 12. The experiments are run on an Intel 510 Series SATA SSD. We create the file set using Filebench 1.5-a3 [42] 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 1 GB. The rest of the file sets have progressively fewer small files. All file sets have a total size of 16 GB. 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.

Table 12. Conversion Time from Ext4 to F2FS for Different Number of Files

| # files | Copy Converter | Manual Conv. | Spiffy Conv. |
|---|---|---|---|
| 20,000 | 188.2 ± 3.7s | 6.6 ± 0.5s | 7.0 ± 0.2s |
| 1,000 | 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 |

The copy converter requires transferring two full copies of the file set, and so it takes 30× to 50× 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.

## 7.5 Prioritized Cache Performance

We measure the performance of our prioritized block layer cache (see Section 5.2.1), 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 10 Gb Ethernet using the iSCSI protocol. The storage server runs Linux 3.11.2 and has four Intel Processor E7-4830 CPUs for a total of 32 cores, 256 GB 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 4 GB and use 8 GB 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 users where one requires higher storage performance than the other. We generate a total file set size of 8 GB with an average file size of 128 KB, 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 23 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 2.3× and 2.7× 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.

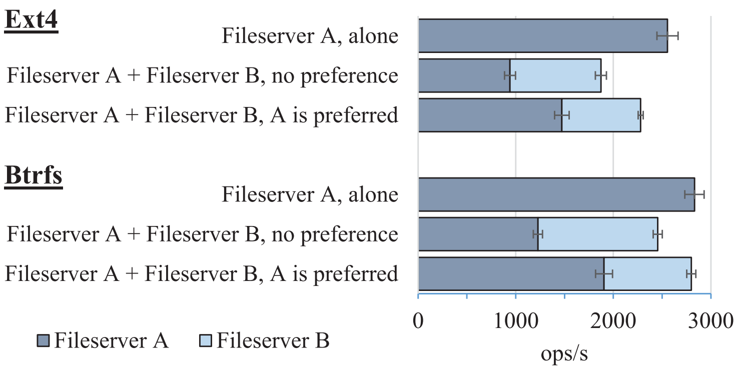

Fig. 23. Throughput of prioritized caching over LRU caching with one or two file servers for Ext4 and Btrfs.

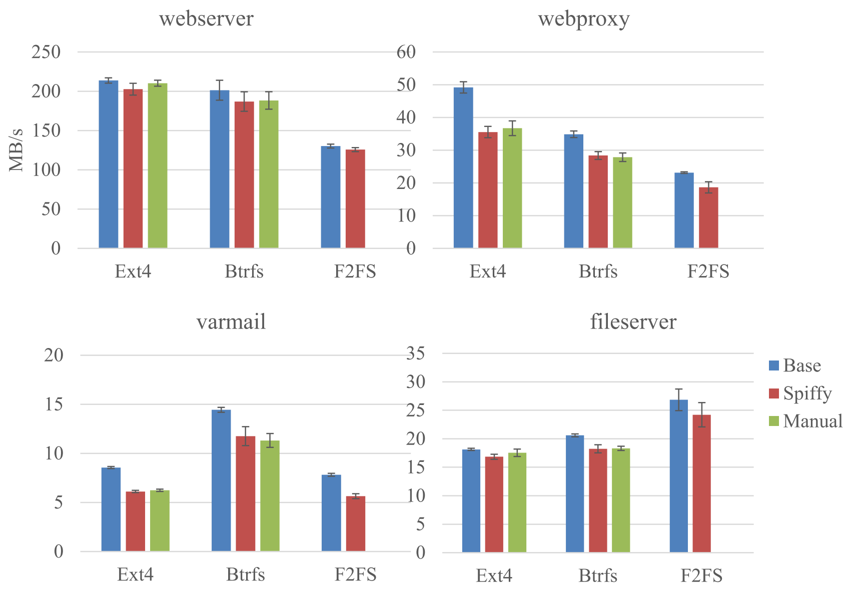

Fig. 24. Throughput of various Filebench workloads without Recon (base), with Spiffy-based Recon implementation (Spiffy), and manually written Recon implementation (Manual), for Ext4, Btrfs, and F2FS.

## 7.6 Runtime Checker Performance

We measure the performance of three file systems on four Filebench workloads to compare the overhead of our Spiffy-based runtime checker (see Section 5.2.2) against manually written runtime checkers. We use the client machine described in Section 7.5. The experiments are run on the Intel 510 Series SATA SSD and main memory is limited to 5 GB. We replicate the workloads used in [11], excluding msnfs, which did not work with our version of Filebench. Since we only want to evaluate parts of the checker that can be written generically using Spiffy, we disabled invariant checking in all the experiments.

Figure 24 shows native performance versus the performance of running different versions of the Recon runtime file-system checkers for each file system, in megabytes per second. In general, we observe that the Spiffy-based implementation performs similarly to the manually written versions, showing negligible overhead across all workloads on both Ext4 and Btrfs.

Spiffy-Recon performs well on the read-mostly webserver workload (10:1 read-to-write ratio), with an overhead of 5.2%, 7.2%, and 3.5% for Ext4, Btrfs, and F2FS, respectively, against baseline. However, for the webproxy workload, which is also read-mostly (5:1 read-to-write ratio), the overhead increases to 27.7%, 18.7%, and 19.5%, respectively. Webproxy uses a very large flat directory, with millions of files, and so the overhead increases since we need to use set differencing for directory entries. For varmail, a sync-heavy workload, the overhead is between 18.6% and 28.5%. This result is expected since committing transactions creates additional work for Recon. Lastly, for fileserver, we notice an overhead of 7.2% to 11.5% despite the workload being write-heavy, which suggests that Recon has higher overheads when there is more metadata that needs to be processed through set differencing, or when the file system performs frequent commits.

## 8 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 [32] 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 [16] 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 [30], 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 [33] and Protocol Buffers [41], 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 [27] and PADS [10] 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 [25, 39, 46]. CCured [25] enables type and memory safety, and the Deputy Type System [46] 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, 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. 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, 18]. 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.

Our concept of identity is akin to a primary key in a relational database [9, 13]. A primary key is a special column whose value is designated to uniquely identify each row or record in the table.

In markup languages such as HTML [19] and LaTeX [20], identity is used to reference a specific element in the document. Spiffy, in contrast, requires file-system developers to specify an expression so that all instances of a type can be uniquely identified when the expression is evaluated at runtime.

In distributed systems, specifying identifiers correctly helps reconstruct execution flow, which is vital for debugging and performance analysis. In order to unambiguously identify an object, the identifers must include the entire causal chain. For example, to uniquely identify a thread, one must specify its process identifier (pid) and the host name. Stitch [45] depends on the above principle to recreate execution flow from unmodified log statements. ÜberTrace [7] adds unique request identifiers to events and propagates them through Facebook's system components to allow inference of causal relationship between events. Similarly, to specify identity for file-system metadata, we often require references to parent structures along the path of pointers leading to the object.

## 9 CONCLUSION

Spiffy is an annotation language for specifying the format of 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.

## ACKNOWLEDGMENTS

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.

## REFERENCES

[1] Sidney Amani, Leonid Ryzhyk, and Toby Murray. 2012. Towards a fully verified file system. EuroSys Doctoral Workshop 2012.

[2] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawa, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2008. Analyzing the effects of disk-pointer corruption. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*. IEEE, 502–511.

[3] Julian Bangert and Nickolai Zeldovich. 2014. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 615–628.

[4] David Beazley. 2013. PLY (Python Lex-Yacc). Retrieved on June 26, 2020 from http://www.dabeaz.com/ply/.

[5] Brian Buckeye and Kevin Liston. 2006. Recovering Deleted Files in Linux. Retrieved on June 26, 2020 from http://citeseerx.ist.psu.edu/viewdoc/download?.

[6] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 18–37.

[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow.

[8] Al Danial. 2009. Cloc–count lines of code. *Open Source* (2009). Retrieved June 26, 2020 from http://cloc.sourceforge.net/.

[9] Ramez Elmasri and Shamkant B. Navathe. 2011. *Database Systems*. Vol. 9. Pearson Education, Boston, MA.

[10] Kathleen Fisher and David Walker. 2011. The PADS project: An overview. In *Proceedings of the 14th International Conference on Database Theory*. ACM, 11–17.

[11] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage* 8, 4 (Dec. 2012), Article 15, 29 pages.

[12] Erich Gamma. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, India.

[13] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2000. *Database System Implementation*. Vol. 672. Prentice Hall: Upper Saddle River, NJ.

[14] Philippa Gardner, Gian Ntzik, and Adam Wright. 2014. Local reasoning for the POSIX file system. In *European Symposium on Programming Languages and Systems*. Springer, 169–188.

[15] Curtis Gedak. 2012. *Manage Partitions with GParted How-to*. Packt Publishing Ltd.

[16] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'07)*. 293–306.

[17] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. SQCK: A declarative file system checker. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[18] Wim H. Hesselink and Muhammad Ikram Lali. 2009. Formalizing a hierarchical file system. *Electronic Notes in Theoretical Computer Science* 259 (2009), 67–85.

[19] Ian Hickson and David Hyatt. 2011. Html5. *W3C Working Draft WD-html5-20110525, May* (2011).

[20] Leslie Lamport. 1994. *LATEX: A Document Preparation System: User's Guide and Reference Manual*. Addison-Wesley.

[21] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 273–286.

[22] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2014. A study of Linux file system evolution. *ACM Transactions on Storage (TOS)* 10, 1 (2014), 3.

[23] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. ffsck: The fast file system checker. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*.

[24] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. 2011. Differentiated storage services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'11)*. 57–70.

[25] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, New York, NY, 128–139. DOI: https://doi.org/10.1145/503272.503286

[26] Kent Overstreet. 2016. Linux Bcache. Retrieved on June 26, 2020 from https://bcache.evilpiepirate.org/.

[27] Meredith Patterson and Dan Hirsch. [n.d.]. Hammer Parser Generator (March 2014). Retrieved on June 26, 2020 from https://github.com/UpstandingHackers/hammer.

[28] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage* 9, 39 (Aug. 2013), Article, 32 pages. DOI: https://doi.org/10.1145/2501620.2501623

[29] Armin Ronacher. 2011. Jinja2 Documentation.

[30] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. 2006. Type-safe disks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 15–28.

[31] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. 2005. A logic of file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'05)*.

[32] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2003. Semantically-smart disk systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'03)*. 73–88.

[33] D. Steedman. 1993. *Abstract Syntax Notation One (ASN. 1): The Tutorial and Reference*. Technology appraisals.

[34] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. 2015. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. ACM, 174–181.

[35] Kuei Sun, Matthew Lakier, Angela Demke Brown, and Ashvin Goel. 2018. Breaking apart the {VFS} for managing file systems. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*.

[36] Kuei Jack Sun. 2013. *Robust Consistency Checking for Modern Filesystems*. Ph.D. Dissertation. University of Toronto.

[37] Microsoft TechNet. [n.d.]. How to Convert FAT Disks to NTFS. Retrieved on June 26, 2020 from https://technet.microsoft.com/en-us/library/bb456984.aspx.

[38] Tom Warren. [n.d.]. Apple is upgrading millions of iOS devices to a new modern file system today. Retrieved March 27, 2017 from https://www.theverge.com/2017/3/27/15076244/apple-file-system-apfs-ios-10-3-features.

[39] Linus Torvalds, Josh Triplett, and Christopher Li. 2007. Sparse—A semantic parser for C. Retrieved on June 26, 2020 from http://sparse.wiki.kernel.org.

[40] Theodore Ts'o. 2017. E2fsprogs: Ext2/3/4 filesystem utilities. Retrieved on June 26, 2020 from http://e2fsprogs.sourceforge.net/.

[41] Kenton Varda. 2008. Protocol buffers: Google's data interchange format. *Google Open Source Blog, available at least as early as July*. 2008.

[42] Andrew Wilson. 2008. The new and improved FileBench. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. https://github.com/filebench/filebench/.

[43] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 393–423.

[44] Michal Zalewski. 2016. American fuzzy lop. Retrieved on June 26, 2020 from http://lcamtuf.coredump.cx/afl/.

[45] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. 2016. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 603–618. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhao.

[46] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. 2006. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 45–60.

Received August 2019; revised January 2020; accepted March 2020