SDC 18
September 24-27, 2018
Santa Clara, CA
www.storagedeveloper.org

Spiffy: Enabling File-System Aware Storage Applications

Kuei (Jack) Sun
University of Toronto

Introduction

- File-system aware applications
  - E.g. partition editor, file system checker, defragmentation tool
  - Operate directly on file system metadata structures
    - Require detailed knowledge of file system format on disk
    - Bypass VFS layer
  - Essential for successful deployment of file system

Problem

- Tools have to be developed from scratch for each file system
- Tools developed only by experts
- Bugs lead to crash, corruption, security vulnerability
- Example: bug 723343 in ntfsprogs
  - NTFS stores the size of MFT record as either:
    - # of clusters per record, if value > 0
    - $2^{|value|}$, if value < 0
  - ntfsprogs misinterprets this field, corrupting NTFS when resizing partitions

Root Cause

- File-system applications are difficult to write
  - File system format complex and often poorly documented
  - Require detailed knowledge of format
  - Cannot be reused across file systems
  - Need to handle file system corruption

Goals

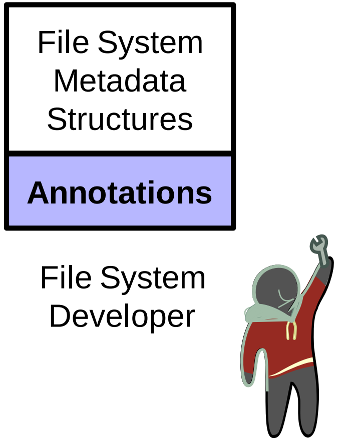
- Simplify development of file-system aware applications
  - Reduce file-system specific code
  - Enable code reuse across file systems
- Improve robustness of these applications
  - Enable correct traversal of file system metadata
  - Ensure type safe access to file system structures
    - Helps detect corruption for both read and write
    - Helps reduce error propagation, and further corruption

Approach: Spiffy Framework

- File system developers specify the format of their file system
- Spiffy uses specification to generate parsing and serialization library
- Developers use library to build robust file-system aware applications

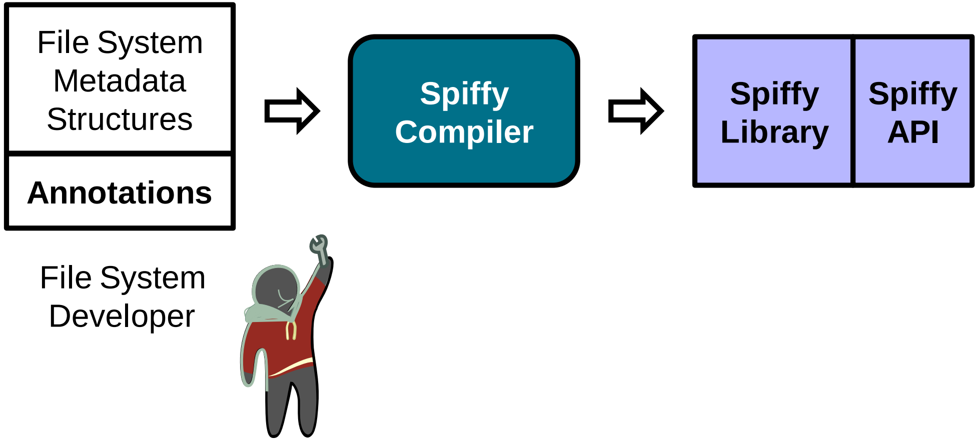
Specifying Format

- File system developers annotate metadata structures in header files of existing source code



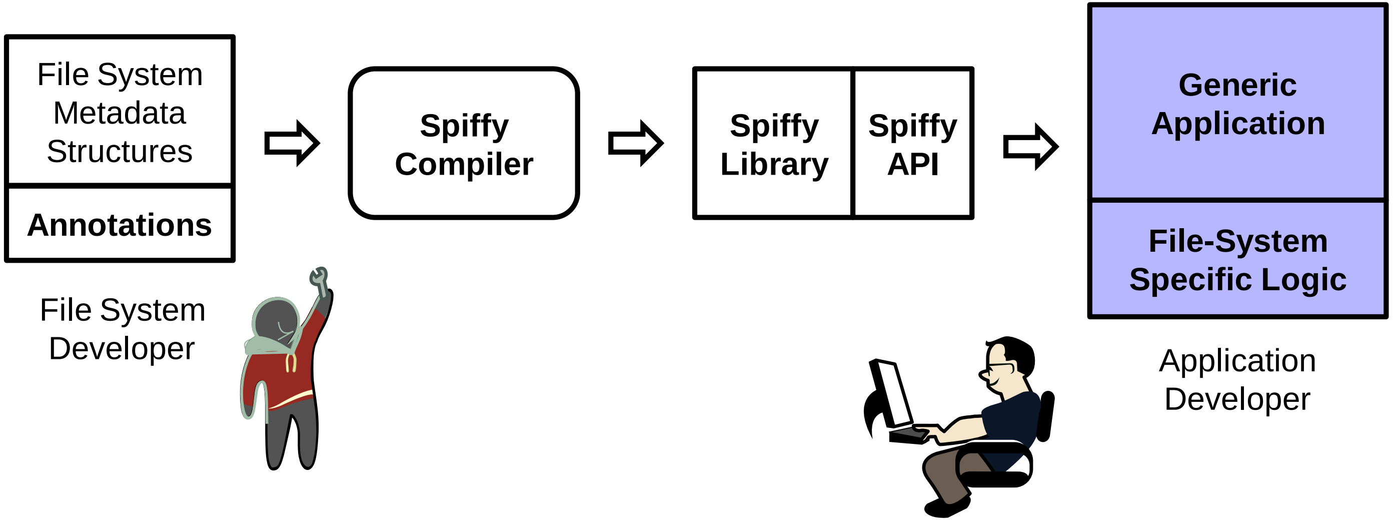
Generating Library

- Spiffy compiler processes annotated metadata structures to generate library that provides a generic API for type-safe parsing, traversal and serialization of file system structures



Building Applications

- Application developers use Spiffy library to build robust tools that work across file systems



File System Metadata Structures

Annotations

File System Developer

Spiffy Compiler

Spiffy Library

Spiffy API

Generic Application

File-System Specific Logic

Application Developer

Talk Outline

- Problem
  - Hard to write robust file system applications
- Approach
- Spiffy Annotations
- Spiffy Library
- Spiffy Applications
- Evaluation
- Conclusion

Need for Annotations

- Need complete specification of the file system format
  - Allows type-safe parsing and updates of file system structures
- Challenge
  - Data structure definitions in source files are incomplete

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

  - bar_block_ptr is “probably” a pointer to type “bar_block”
  - However, its hard to deduce this type information

Need for Annotations

- Solution
  - Annotate structures to supply missing information

```
FSSTRUCT() foo {
    __le32 size;

    POINTER(..., type=bar_block)
    __le32 bar_block_ptr;
};
```

Need for Annotations

- Solution
  - Annotate structures to supply missing information

```
FSSTRUCT() foo {
    __le32 size;

    POINTER(..., type=bar_block)
    __le32 bar_block_ptr;
};
```

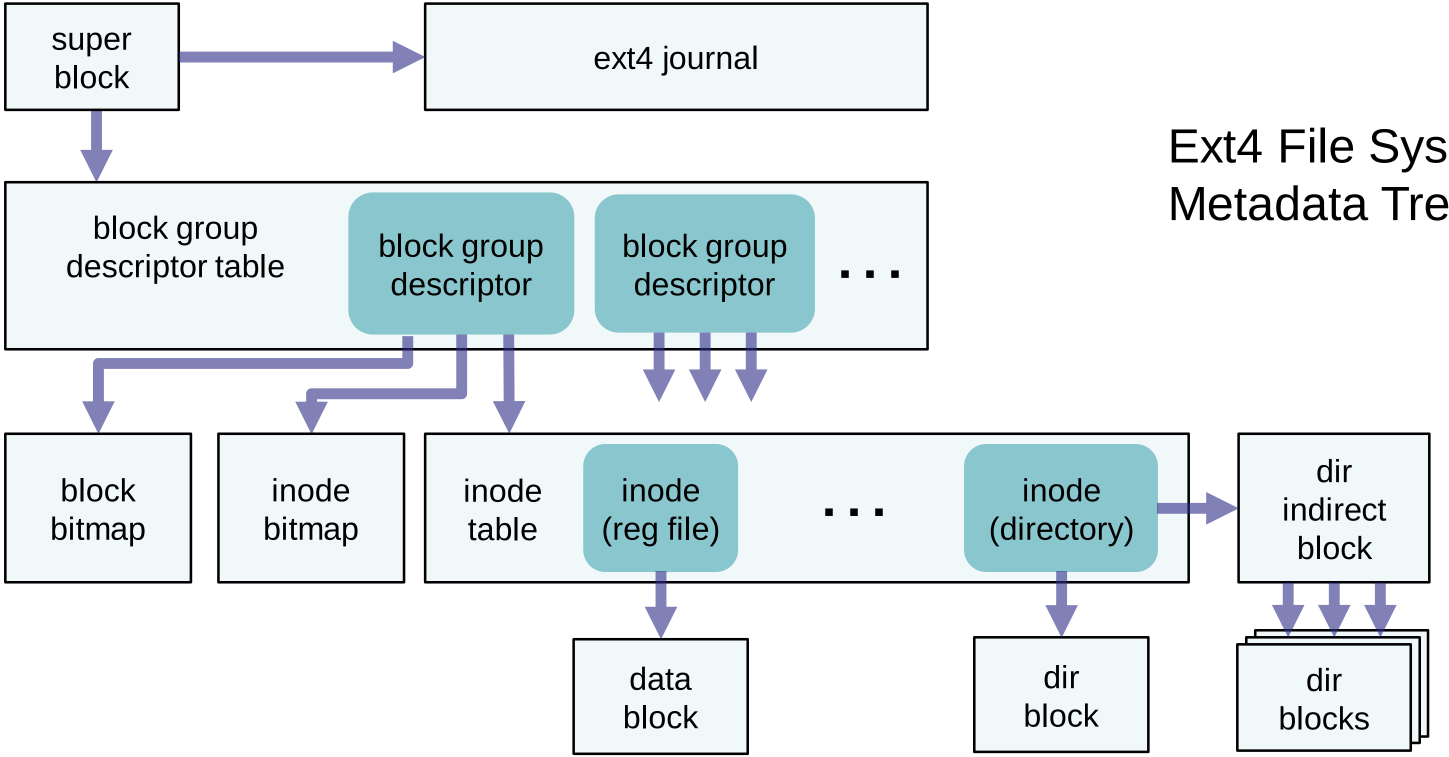
Need for Annotations

- Solution
  - Annotate structures to supply missing information

```
FSSTRUCT() foo {
    __le32 size;

    POINTER(..., type=bar_block)
    __le32 bar_block_ptr;
};
```

Pointer Annotations



Ext4 File System Metadata Tree

Pointer Address Space

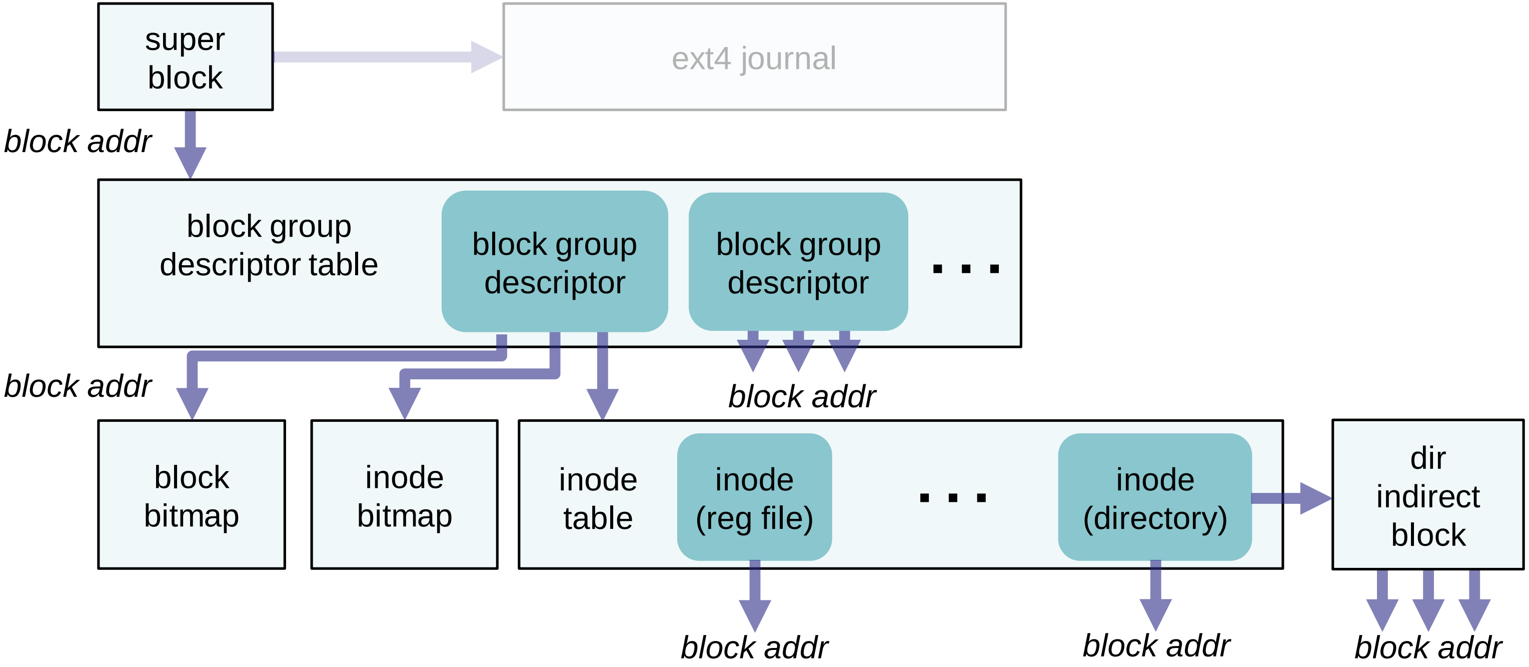
- Main challenge: File system pointers can store different types of logical addresses
  - Need different mappings to obtain physical address
- Solution: Pointer annotations specify an *address space* that indicates how the address should be mapped to physical location

```
POINTER(aspc=block, type=bar_block)
```

- Examples: Block and File address spaces

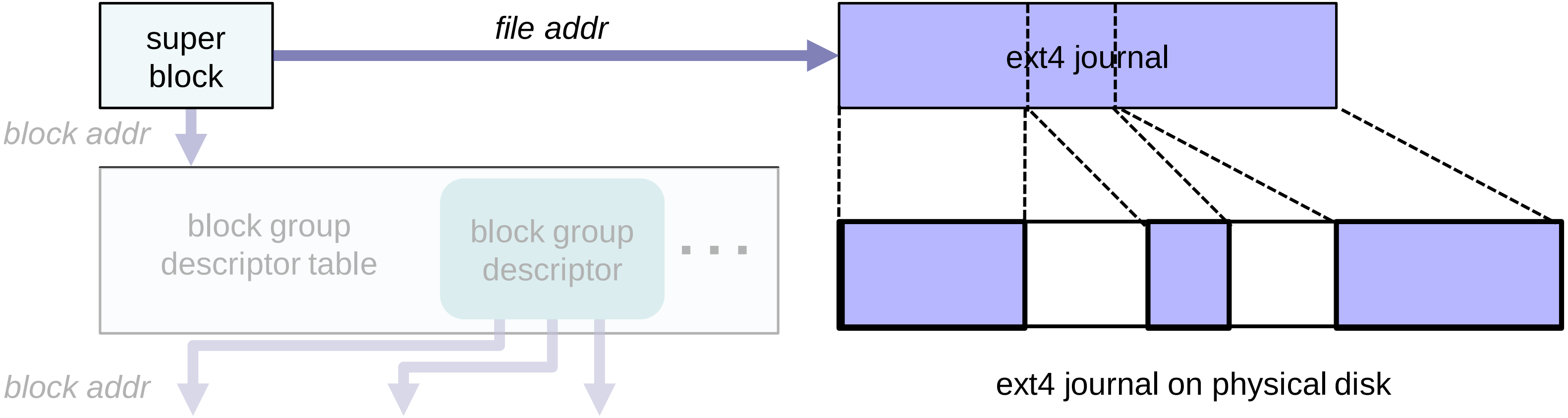
Block Address Space

- Block address is the block number in the file system



File Address Space

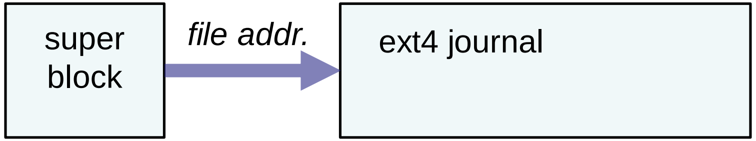
- File address is an index into the inode table for a file
  - E.g. Ext4 journal is stored as a regular file
  - Regular file may be physically discontiguous
  - Requires mapping logical blocks of the file to their physical locations



Super Block

- Super block is the root of every file system tree
  - Specified using FSSUPER annotation
  - *location* specifies address of super block in byte offset

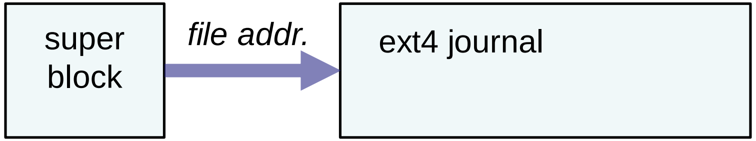
```
FSSUPER(location=1024) ext4_super_block
{
  __le32 s_log_block_size;
  ...
  POINTER(aspc=file,
          type=ext4_journal)
  __le32 s_journal_inum;
};
```



Super Block

- Super block is the root of every file system tree
  - Specified using FSSUPER annotation
  - *location* specifies address of super block in byte offset

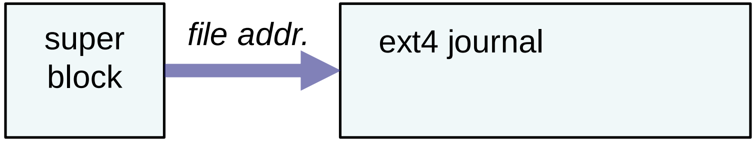
```
FSSUPER(location=1024) ext4_super_block
{
  __le32 s_log_block_size;
  ...
  POINTER(aspc=file,
          type=ext4_journal)
  __le32 s_journal_inum;
};
```



Super Block

- Super block is the root of every file system tree
  - Specified using FSSUPER annotation
  - *location* specifies address of super block in byte offset

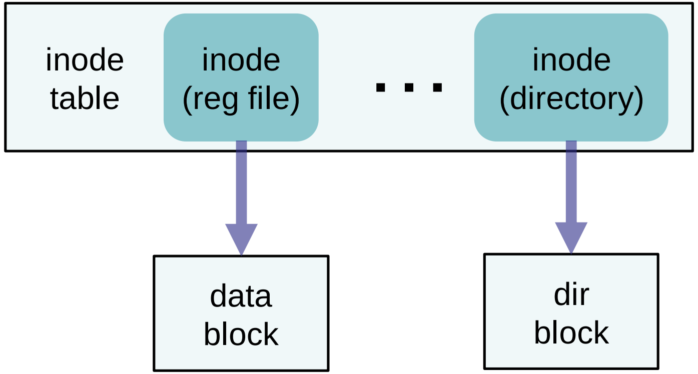
```
FSSUPER(location=1024) ext4_super_block
{
  __le32 s_log_block_size;
  ...
  POINTER(aspc=file,
          type=ext4_journal)
  __le32 s_journal_inum;
};
```



Context-Sensitive Types

- A field may refer to different types of metadata
  - Pointers in inode structure can point to directory or data blocks
- Supported by specifying *when* condition in pointer annotation

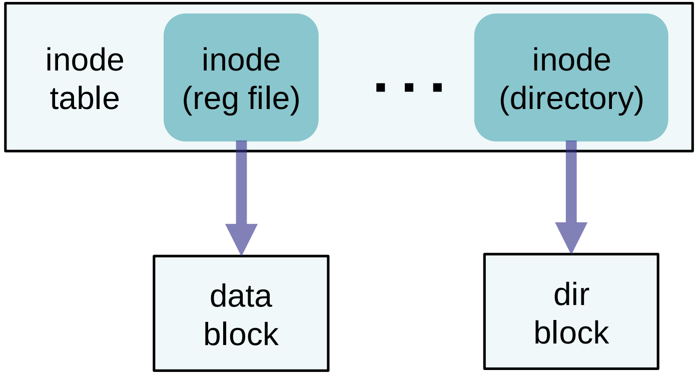
```
FSSTRUCT(...) ext4_inode {
  __le16 i_mode;
   …
  POINTER(aspc=block, type=dir_block,
          when=self.i_mode & S_IFDIR)
  POINTER(aspc=block, type=data_block,
          when=self.i_mode & S_IFREG)
  __le32 i_block[EXT3_NDIR_BLOCKS];
   …
};
```



Context-Sensitive Types

- A field may refer to different types of metadata
  - Pointers in inode structure can point to directory or data blocks
- Supported by specifying *when* condition in pointer annotation

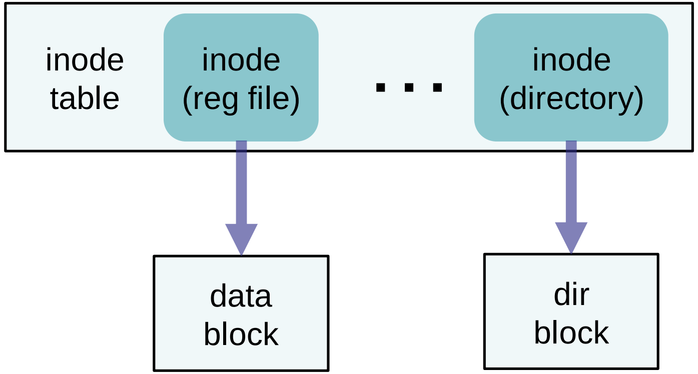
```
FSSTRUCT(...) ext4_inode {
  __le16 i_mode;
   …
  POINTER(aspc=block, type=dir_block,
          when=self.i_mode & S_IFDIR)
  POINTER(aspc=block, type=data_block,
          when=self.i_mode & S_IFREG)
  __le32 i_block[EXT3_NDIR_BLOCKS];
   …
};
```



Context-Sensitive Types

- A field may refer to different types of metadata
  - Pointers in inode structure can point to directory or data blocks
- Supported by specifying *when* condition in pointer annotation

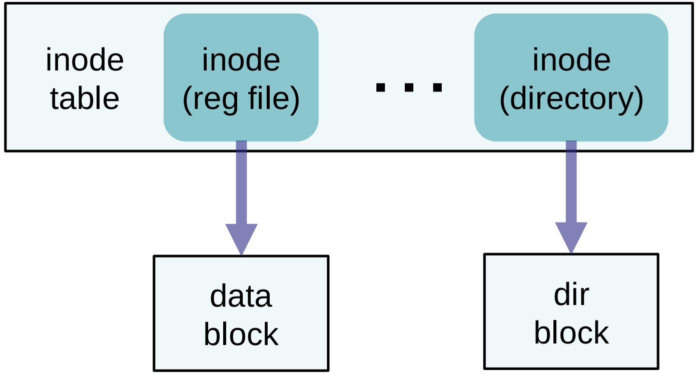
```
FSSTRUCT(...) ext4_inode {
  __le16 i_mode;
   …
  POINTER(aspc=block, type=dir_block,
          when=self.i_mode & S_IFDIR)
  POINTER(aspc=block, type=data_block,
          when=self.i_mode & S_IFREG)
  __le32 i_block[EXT3_NDIR_BLOCKS];
   …
};
```



Context-Sensitive Types

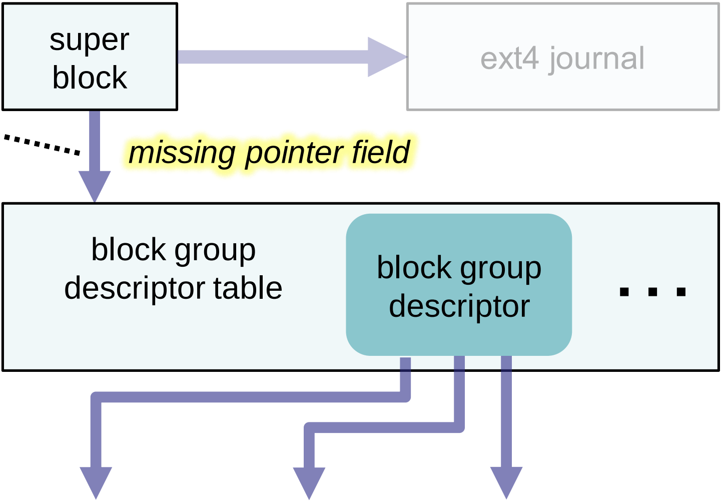
- A field may refer to different types of metadata
  - Pointers in inode structure can point to directory or data blocks
- Supported by specifying *when* condition in pointer annotation

```
FSSTRUCT(...) ext4_inode {
  __le16 i_mode;
   …
  POINTER(aspc=block, type=dir_block,
          when=self.i_mode & S_IFDIR)
  POINTER(aspc=block, type=data_block,
          when=self.i_mode & S_IFREG)
  __le32 i_block[EXT3_NDIR_BLOCKS];
   …
};
```



Missing Pointer

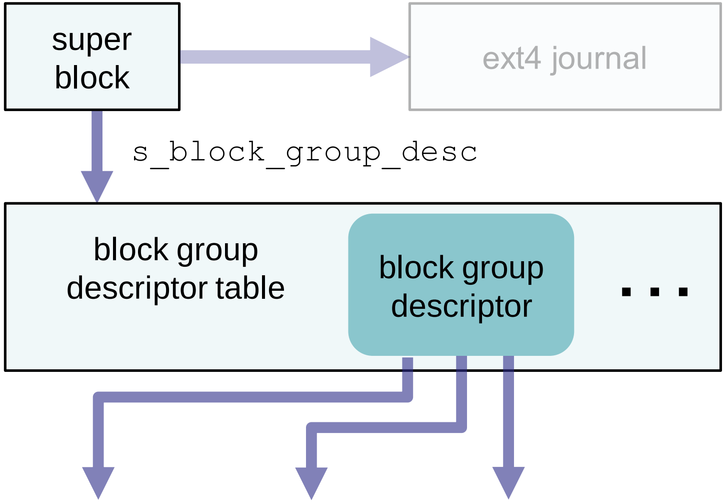
- Locations of some structures are implicit in the code
- E.g. Ext4 block group descriptor table is the next block following the super block
  - Ext4 super block does not have a field that points to descriptor table
  - Pointer required for file system traversal



Implicit Pointer

- Solution: Implicit pointer annotation
  - *name* creates a logical pointer field that can be dereferenced
  - *expr* is a C expression that specifies how to calculate the field value
    - Expression can reference other fields in the structure

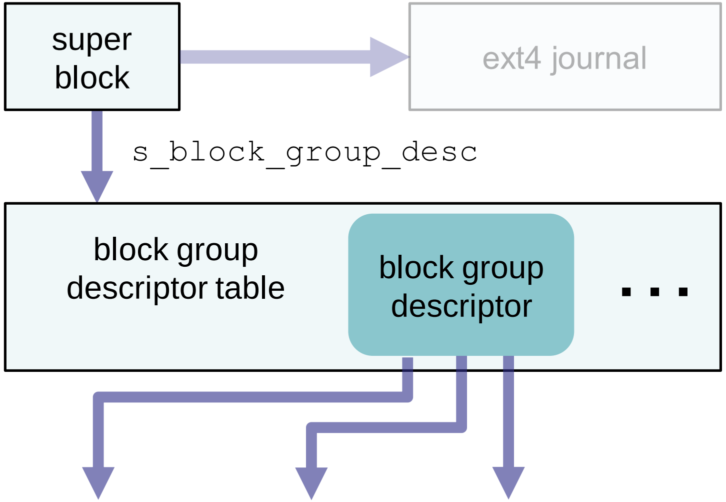
```
FSSUPER(...) ext4_super_block {
  __le32 s_log_block_size;
  ...
  POINTER(name=s_block_group_desc,
  type=ext4_group_desc_table, aspc=block,
  expr=(self.s_log_block_size == 0) ? 2 : 1);
};
```



Implicit Pointer

- Solution: Implicit pointer annotation
  - *name* creates a logical pointer field that can be dereferenced
  - *expr* is a C expression that specifies how to calculate the field value
    - Expression can reference other fields in the structure

```
FSSUPER(...) ext4_super_block {
  __le32 s_log_block_size;
  ...
  POINTER(name=s_block_group_desc,
  type=ext4_group_desc_table, aspc=block,
  expr=(self.s_log_block_size == 0) ? 2 : 1);
};
```



Vector Types

- Spiffy allows specifying vector types via VECTOR annotation
- A vector contains a sequence of elements of the same type
- The size of the vector can be specified using
    1) number of elements
    2) sentinel value
    3) total vector size

Vector Types

- Spiffy allows specifying vector types via VECTOR annotation
- A vector contains a sequence of elements of the same type
- The size of the vector can be specified using
  1) **number of elements**
  2) sentinel value
  3) total vector size

```
VECTOR(name=inode_block, type=struct ext4_inode,
          count=BLOCK_SIZE/sb.s_inode_size);
```

Vector Types

- Spiffy allows specifying vector types via VECTOR annotation
- A vector contains a sequence of elements of the same type
- The size of the vector can be specified using
  1) number of elements
  2) sentinel value
  3) **total vector size**

```
FSSTRUCT() directory_indirect_ptr {
    POINTER(aspc=block, type=dir_block)
    __le32 ind_block_nr;
};
```

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

Vector Types

- Spiffy allows specifying vector types via VECTOR annotation
- A vector contains a sequence of elements of the same type
- The size of the vector can be specified using
  1) number of elements
  2) sentinel value
  3) **total vector size**

```
FSSTRUCT() directory_indirect_ptr {
    POINTER(aspc=block, type=dir_block)
    __le32 ind_block_nr;
};
```

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

Vector Types

- Spiffy allows specifying vector types via VECTOR annotation
- A vector contains a sequence of elements of the same type
- The size of the vector can be specified using
  1) number of elements
  2) sentinel value
  3) **total vector size**

```
FSSTRUCT() directory_indirect_ptr {
    POINTER(aspc=block, type=dir_block)
    __le32 ind_block_nr;
};
```

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

Check Annotations

```
FSSUPER(…) ext4_super_block {
  __le32 s_log_block_size;
  __le16 s_magic;
  …
  CHECK(expr=self.s_log_block_size <= 6);
  CHECK(expr=self.s_magic == 0xef53);
};
```

Generated Code for ext4_super_block

```
int Ext4SuperBlock::parse(const char * & buf, unsigned & len) {
  int ret;
  if ((ret = s_log_block_size.parse(buf, len)) < 0) return ret;
  …
  if (!(s_log_block_size <= 6)) return ERR_CORRUPT;
  if (!(s_magic == 0xef53)) return ERR_CORRUPT;
  return 0;
}
```

Check Annotations

```
FSSUPER(…) ext4_super_block {
  __le32 s_log_block_size;
  __le16 s_magic;
  …
  CHECK(expr=self.s_log_block_size <= 6);
  CHECK(expr=self.s_magic == 0xef53);
};
```

Generated Code for ext4_super_block

```
int Ext4SuperBlock::parse(const char * & buf, unsigned & len) {
  int ret;
  if ((ret = s_log_block_size.parse(buf, len)) < 0) return ret;
  …
  if (!(s_log_block_size <= 6)) return ERR_CORRUPT;
  if (!(s_magic == 0xef53)) return ERR_CORRUPT;
  return 0;
}
```

Check Annotations

```
FSSUPER(…) ext4_super_block {
  __le32 s_log_block_size;
  __le16 s_magic;
  …
  CHECK(expr=self.s_log_block_size <= 6);
  CHECK(expr=self.s_magic == 0xef53);
};
```

Generated Code for ext4_super_block

```
int Ext4SuperBlock::parse(const char * & buf, unsigned & len) {
  int ret;
  if ((ret = s_log_block_size.parse(buf, len)) < 0) return ret;
  …
  if (!(s_log_block_size <= 6)) return ERR_CORRUPT;
  if (!(s_magic == 0xef53)) return ERR_CORRUPT;
  return 0;
}
```

Generating Spiffy Library

- C++ classes are generated for all annotated structures and their fields
  - Enables type-safe parsing and serialization
  - Allows introspection of type, size, name, and parents

Evaluation: Annotation Effort

File System	Line Count	Annotated
Ext4	491	113
Btrfs	556	151
F2FS	462	127

- Lines of code required to correctly annotate file systems
  - Need to declare some structures
    - E.g. Ext4 indirect block assumed to be an array of 4-byte pointers
  - Changed some structures for clarity
    - E.g. block pointers in Ext4 inode is an array of 15 pointers: first 12 are direct pointers, last 3 are indirect pointers of different types

Building Applications

- Example: File System Free Space Tool
  - Plots histogram of size of free extents
  - Application requires knowledge of how file system tracks block allocation
- Manually
  - Write code to traverse file system and access relevant metadata
    - Often through trial-and-error
  - Write code to process relevant metadata
- Spiffy framework
  - Simplifies the traversal and helps make it more robust
  - Application program focuses on processing relevant metadata

Manually-Written Application

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  /* ext4 super block is 1024 bytes away from start */
  struct ext4_super_block * sb = dev.read(1024, SB_SIZE);
  int blk_size = 1024 << sb->s_log_block_size;
  dev.set_block_size(blk_size);
  /* block group descriptors start at block 2 or 1 */
  int bg_blknr = (sb->s_log_block_size == 0) 2 : 1;
  int bg_ngrps = ceil(sb->s_blocks_count, sb->s_blocks_per_group);
  int bg_nblks = ceil(bg_ngrps*sizeof(struct ext4_group_desc), blk_size);
  /* read all of the block group descriptors into memory */
  struct ext4_group_desc * gd = dev.read_block(bg_blknr, bg_nblks);
  for (int i = 0; i < bg_ngrps; ++i) {
    char * buf = dev.read_block(gd[i]->bg_block_bitmap);
    int ret = process_block_bitmap(buf, vec);
    …
  }
  …
}
```

LOTS of boilerplate code to walk through the intermediate structures

Manually-Written Application

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  /* ext4 super block is 1024 bytes away from start */
  struct ext4_super_block * sb = dev.read(1024, SB_SIZE);
  int blk_size = 1024 << sb->s_log_block_size;
  dev.set_block_size(blk_size);
  /* block group descriptors start at block 2 or 1 */
  int bg_blknr = (sb->s_log_block_size == 0) 2 : 1;
  int bg_ngrps = ceil(sb->s_blocks_count, sb->s_blocks_per_group);
  int bg_nblks = ceil(bg_ngrps*sizeof(struct ext4_group_desc), blk_size);
  /* read all of the block group descriptors into memory */
  struct ext4_group_desc * gd = dev.read_block(bg_blknr, bg_nblks);
  for (int i = 0; i < bg_ngrps; ++i) {
    char * buf = dev.read_block(gd[i]->bg_block_bitmap);
    int ret = process_block_bitmap(buf, vec);
    …
  }
  …
}
```

Ideally, we would only have to write this function

Manually-Written Application

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  /* ext4 super block is 1024 bytes away from start */
  struct ext4_super_block * sb = dev.read(1024, SB_SIZE);
  int blk_size = 1024 << sb->s_log_block_size;
  dev.set_block_size(blk_size);
  /* block group descriptors start at block 2 or 1 */
  int bg_blknr = (sb->s_log_block_size == 0) 2 : 1;
  int bg_ngrps = ceil(sb->s_blocks_count, sb->s_blocks_per_group);
  int bg_nblks = ceil(bg_ngrps*sizeof(struct ext4_group_desc), blk_size);
  /* read all of the block group descriptors into memory */
  struct ext4_group_desc * gd = dev.read_block(bg_blknr, bg_nblks);
  for (int i = 0; i < bg_ngrps; ++i) {
    char * buf = dev.read_block(gd[i]->bg_block_bitmap);
    int ret = process_block_bitmap(buf, vec);
    …
  }
  …
}
```

No sanity checks! Value may be out-of-bound or invalid, which can cause crashes or garbage output

Application Using Spiffy Library

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  Ext4 ext4(dev);
  /* read super block into memory */
  Ext4::SuperBlock * sb = ext4.fetch_super();
  if (sb == nullptr) return -1;
  dev.set_block_size(1024 << sb->s_log_block_size);
  /* traverse file system and find/process all block bitmaps */
  return sb->process_by_type(BLOCK_BITMAP,
                             process_block_bitmap, &vec);
}
```

Returns *nullptr* if super block is corrupted

Application Using Spiffy Library

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  Ext4 ext4(dev);
  /* read super block into memory */
  Ext4::SuperBlock * sb = ext4.fetch_super();
  if (sb == nullptr) return -1;
  dev.set_block_size(1024 << sb->s_log_block_size);
  /* traverse file system and find/process all block bitmaps */
  return sb->process_by_type(BLOCK_BITMAP,
                             process_block_bitmap, &vec);
}
```

THAT'S IT

Application Using Spiffy Library

```
int process_ext4(vector<Extent> & vec, Device & dev) {
  Ext4 ext4(dev);
  /* read super block into memory */
  Ext4::SuperBlock * sb = ext4.fetch_super();
  if (sb == nullptr) return -1;
  dev.set_block_size(1024 << sb->s_log_block_size);
  /* traverse file system and find/process all block bitmaps */
  return sb->process_by_type(BLOCK_BITMAP,
                             process_block_bitmap, &vec);
}
```

- Advantages
  - simplifies file system traversal, reduces need to know format details
  - library parsing routines have automatically generated sanity checks

Spiffy Application for Btrfs

```
int process_btrfs(vector<Extent> & vec, Device & dev) {
  Btrfs btrfs(dev);
  /* read super block into memory */
  Btrfs::SuperBlock * sb = btrfs.fetch_super();
  if (sb == nullptr) return -1;
  dev.set_block_size(sb->sectorsize);
  /* traverse file system and find/process all extent items */
  return sb->process_by_type(EXTENT_ITEM,
                             process_extent_item, &vec);
}
```

Spiffy Applications

	Read-Only	**Read-Write**
Offline (Userspace)	• File System Free Space Tool • File System Dump Tool	• Type-Specific File System Corruptor • File System Conversion Tool
Online (Kernel)	• File-System Aware Block Layer Cache • Runtime File Systems Checker	

File System Dump Tool

- Helps debug file system implementation
- Parses all metadata and exports them in XML format

```
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Container * sup = fs.fetch_super();
  if (sup != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

```
EntVisitor ev;
PtrVisitor pv;

int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}

int PtrVisitor::visit(Entity & p) {
  Container * tmp;
  tmp = p.to_pointer()->fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
```

File System Dump Tool

- Helps debug file system implementation
- Parses all metadata and exports them in XML format

```
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Container * sup = fs.fetch_super();
  if (sup != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

```
EntVisitor ev;
PtrVisitor pv;

int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}

int PtrVisitor::visit(Entity & p) {
  Container * tmp;
  tmp = p.to_pointer()->fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
```

File System Dump Tool

- Helps debug file system implementation
- Parses all metadata and exports them in XML format

```
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Container * sup = fs.fetch_super();
  if (sup != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

```
EntVisitor ev;
PtrVisitor pv;

int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}

int PtrVisitor::visit(Entity & p) {
  Container * tmp;
  tmp = p.to_pointer()->fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
```

File System Dump Tool

- Helps debug file system implementation
- Parses all metadata and exports them in XML format

```
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Container * sup = fs.fetch_super();
  if (sup != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

```
EntVisitor ev;
PtrVisitor pv;

int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}

int PtrVisitor::visit(Entity & p) {
  Container * tmp;
  tmp = p.to_pointer()->fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
```

File System Dump Tool

- Provides API to filter out fields and structures
  - Helps reduce and declutter the output
  - E.g. Ext4 dump tool does not export unallocated inode
- Works for all annotated file systems
  - Generic Application Code: 482 LOC
  - File-System Specific Code: 30 to 60 LOC each

Type-Specific File System Corruptor

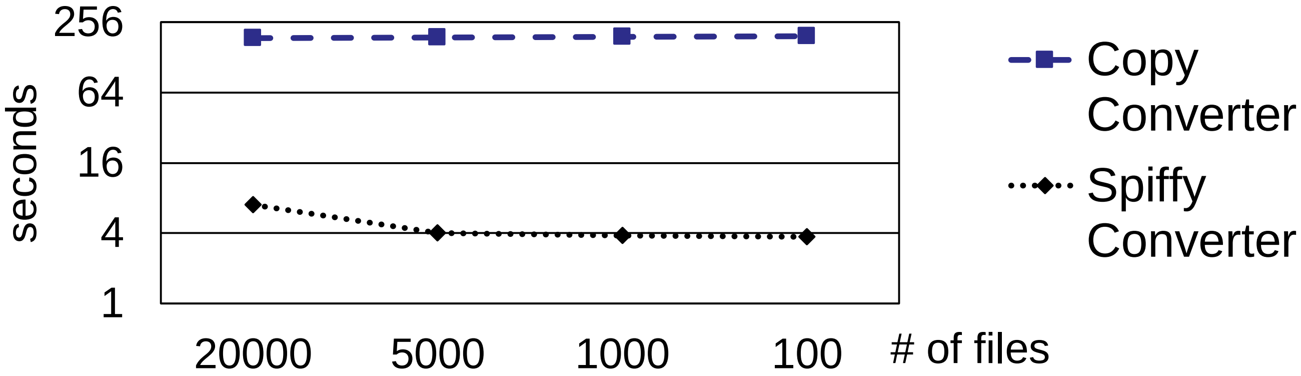
- Helps test robustness of file systems and their tools
- Finds and corrupts a field in a specified structure
  - Generic Application Code: 455 LOC
  - File-System Specific Code: < 30 LOC each
- Corruption Experiment
  - Ran existing tools on corrupt file system image
  - Discovered 1 crash bug in dumpe2fs (Ext4)
  - Discovered 5 crash bugs in dump.f2fs (F2FS)
  - None in our Spiffy dump tool on Ext4, Btrfs and F2FS

File System Conversion Tool

- Converts from one file system to another
  - In-place conversion, no secondary device needed
  - Minimizes copying data blocks
- Currently, converts from Ext4 to F2FS
  - Generic application code: 504 LOC
  - Ext4 specific code (source file system): 218 LOC
  - F2FS specific code (destination file system): 1760 LOC

Evaluation: Ext4 to F2FS Converter

- Compare Spiffy converter versus copy-based converter
  - Copy converter copies data to local disk, reformat, then copies back
- Converts 64GB file system with 16GB of data on SSD



- Copy converter 30~50 times slower

File-system Aware Block Layer Cache

- Supports block caching policies that use file-system specific information
  - Implemented at the block layer
  - Requires no changes to the file system!
- Identifies and interprets blocks as they are read or written
  - Identifies the types of blocks
  - Interprets their contents to extract file-system specific information
- Block caching policies
  - Cache file system metadata
    - When a block is accessed, Spiffy helps determine whether block is data/metadata
  - Cache small files, cache a specific user’s files
    - When a block is accessed, Spiffy helps determine the file to which block belongs

Runtime File System Checker

- Checks whether file system writes would cause file system inconsistency on disk
  - Identifies and interprets blocks as they are read or written
  - At commit time, compares old and new versions of modified blocks
  - Generates logical changes to file system metadata
  - Checks changes against file-system specific consistency rules
- Evaluation
  - Ext4 manual differencing: 2099 lines of code
  - Ext4 Spiffy differencing: 1059 lines of code

Demo of Spiffy Applications

- Type-Specific File System Corruptor
- File System Dump Tool
- And more … (time permitting)

Conclusion

- Spiffy framework
  - Annotation language for specifying file system format
  - Enables generating a library for traversing file system metadata
- Simplifies development of file-system aware applications
  - Reduces file-system specific code
  - Enables code reuse across file systems
- Enables writing robust applications
  - Provides type-safe parsing and serialization of metadata
  - Helps detect file system corruption

Find Out More

- FAST 2018 Paper
  - https://www.usenix.org/system/files/conference/fast18/fast18-sun.pdf
- GitHub repository
  - https://github.com/jacksun007/spiffy

SDC 18

September 24-27, 2018
Santa Clara, CA

www.storagedeveloper.org

Spiffy: Enabling File-System Aware Storage Applications

Kuei (Jack) Sun
University of Toronto

Spiffy API (C++)

Base Class	Member Functions	Description
Spiffy File System Library		
`Entity`	`int process_fields(Visitor & v)`	allows *v* to visit all fields of this object
	`int process_pointer(Visitor & v)`	allows *v* to visit all pointers of this object
	`int process_by_type(int t, Visitor & v)`	allows *v* to visit all structures of type *t* that is reachable from this object
	`get_name(), get_size(), etc.`	allows for type introspection
`Container`	`int save()`	serializes and persists the container
`Pointer`	`Container * fetch()`	retrieves pointed-to container from disk
`FileSystem`	`FileSystem(IO & io)`	instantiates a new file system object
	`Container * fetch_super()`	retrieves the super block from disk
Application Developer		
`Visitor`	`virtual int visit(Entity & e)=0;`	visits an entity and possibly processes it