Spiffy: Enabling File-System Aware Storage Applications

Kuei (Jack) Sun, Daniel Fryer, Joseph Chu, Matthew Lakier, Angela Demke Brown, and Ashvin Goel

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 system crash, data 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

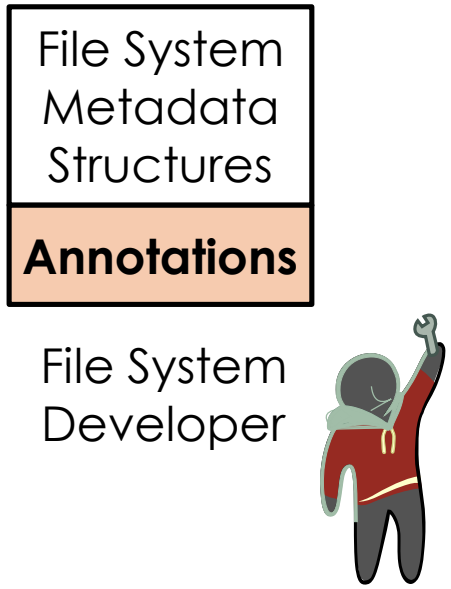
- Improve robustness of these applications
  - Enable correct traversal of file system metadata
  - Ensure type safe access to file system structures
    - Helps detect corruption for both read and write
    - Helps reduce error propagation, and further corruption

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

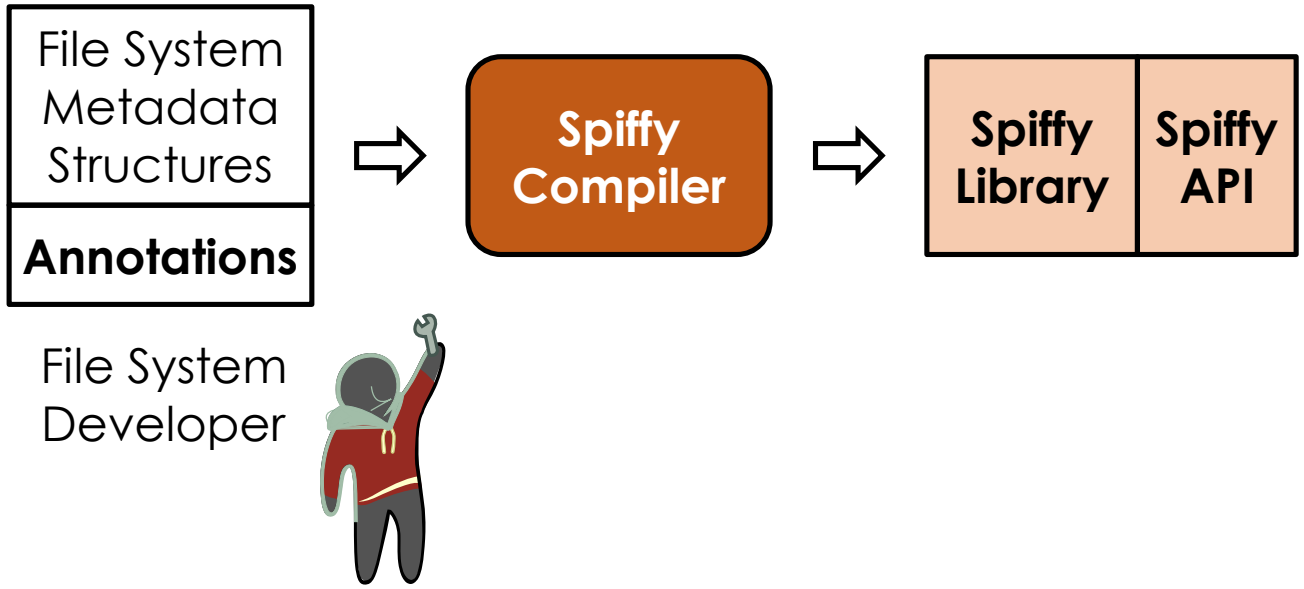
Specifying Format

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



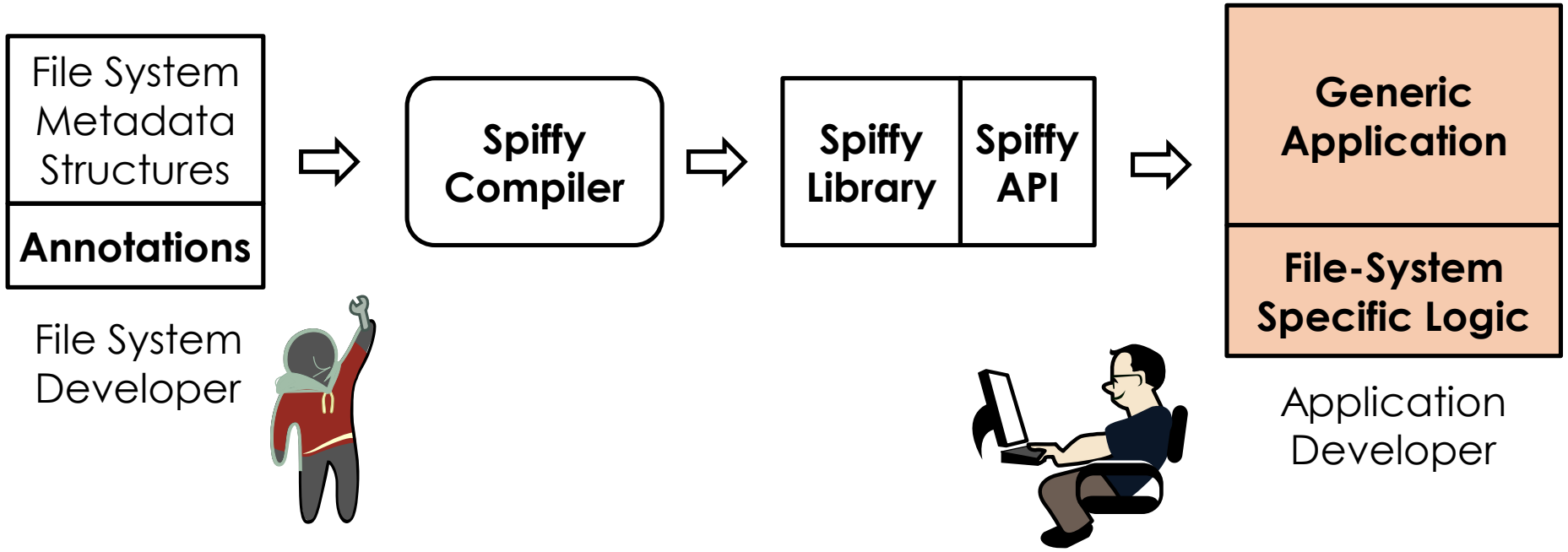
Generating Library

- 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



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 32-bit little endian pointer to type “bar_block”
  - However, its hard to deduce this type information

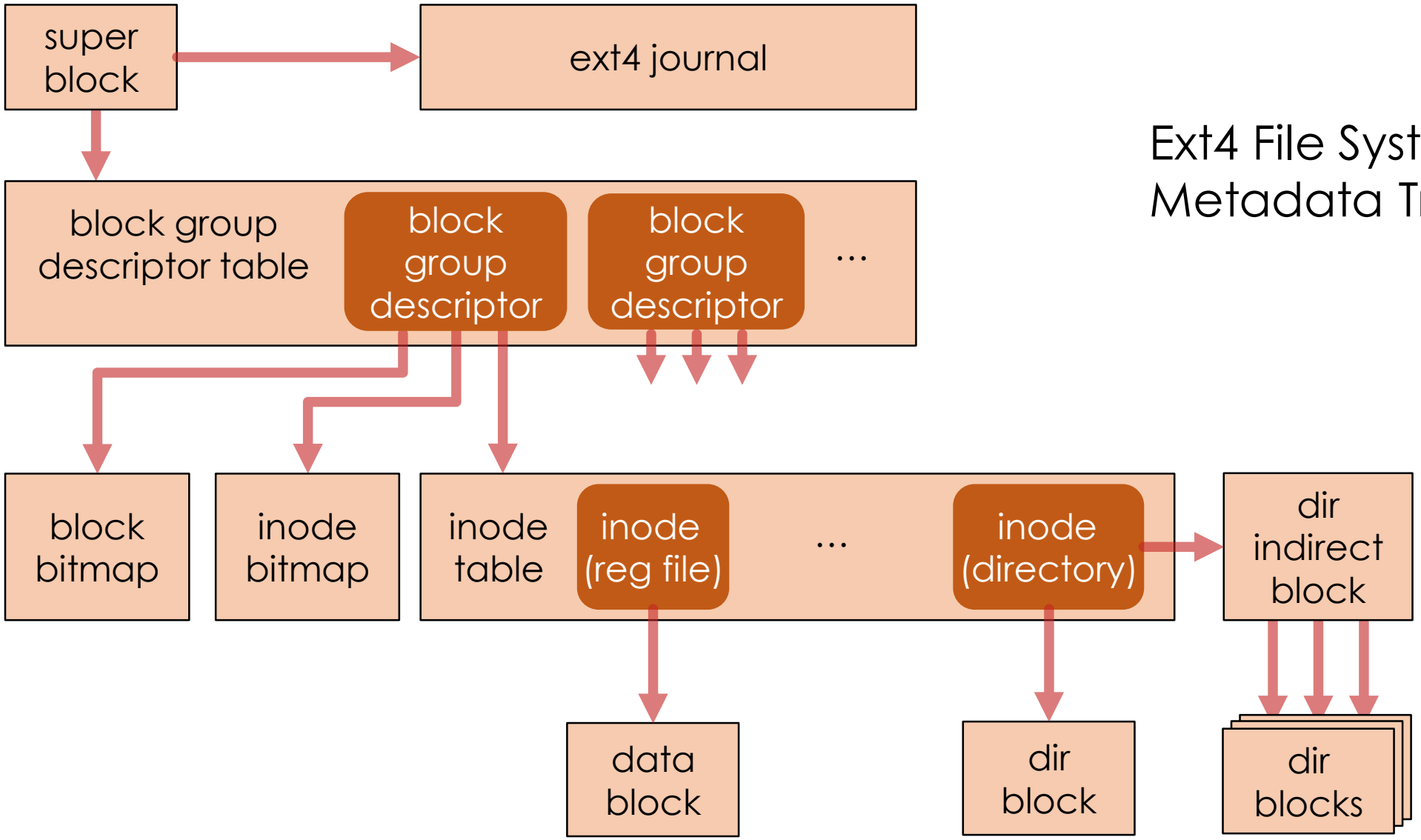
Need for Annotations

- Solution
  - Annotate structures to supply missing information

```
FSSTRUCT() foo {
    __le32 size;

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

Pointer Annotations



Ext4 File System Metadata Tree

Pointer Address Space

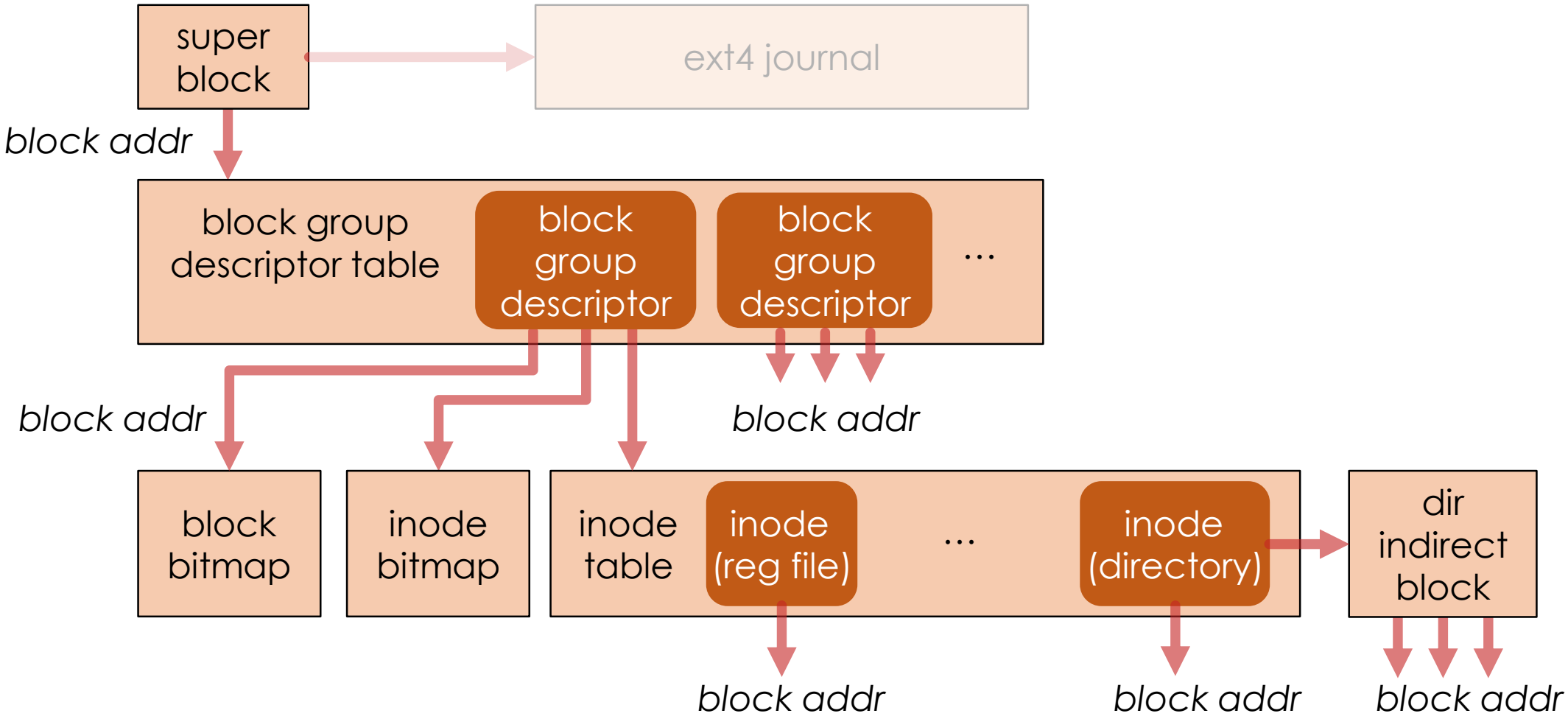
- Challenge: File system pointer can store different types of logical addresses
  - Need different mappings to obtain physical address
- Solution: Pointer annotations specify an *address space* that indicates how the address should be mapped to physical location

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

- Examples: Block, File, F2FS NID address spaces

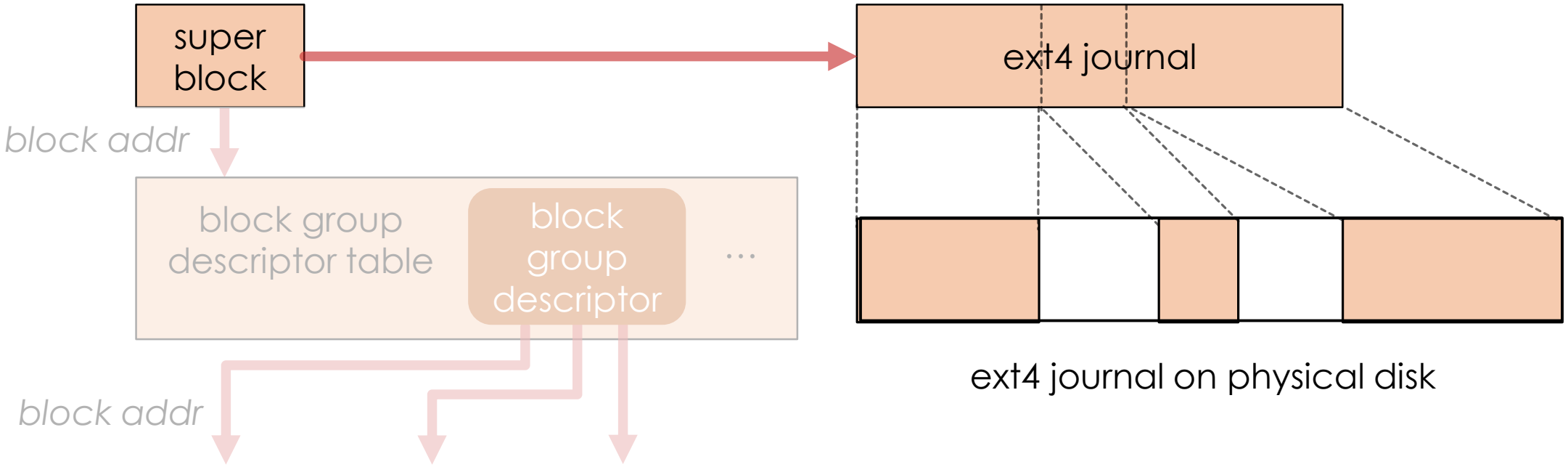
Block Address Space

- Block address is the block number in the file system



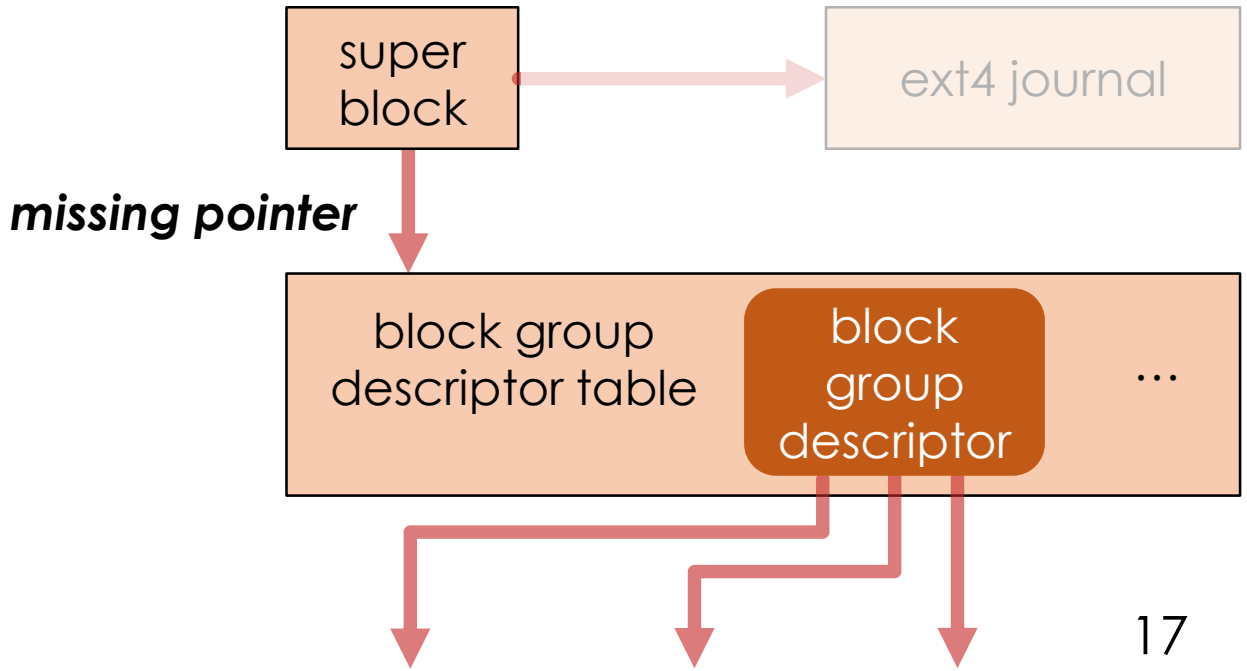
File Address Space

- 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



Missing Pointer

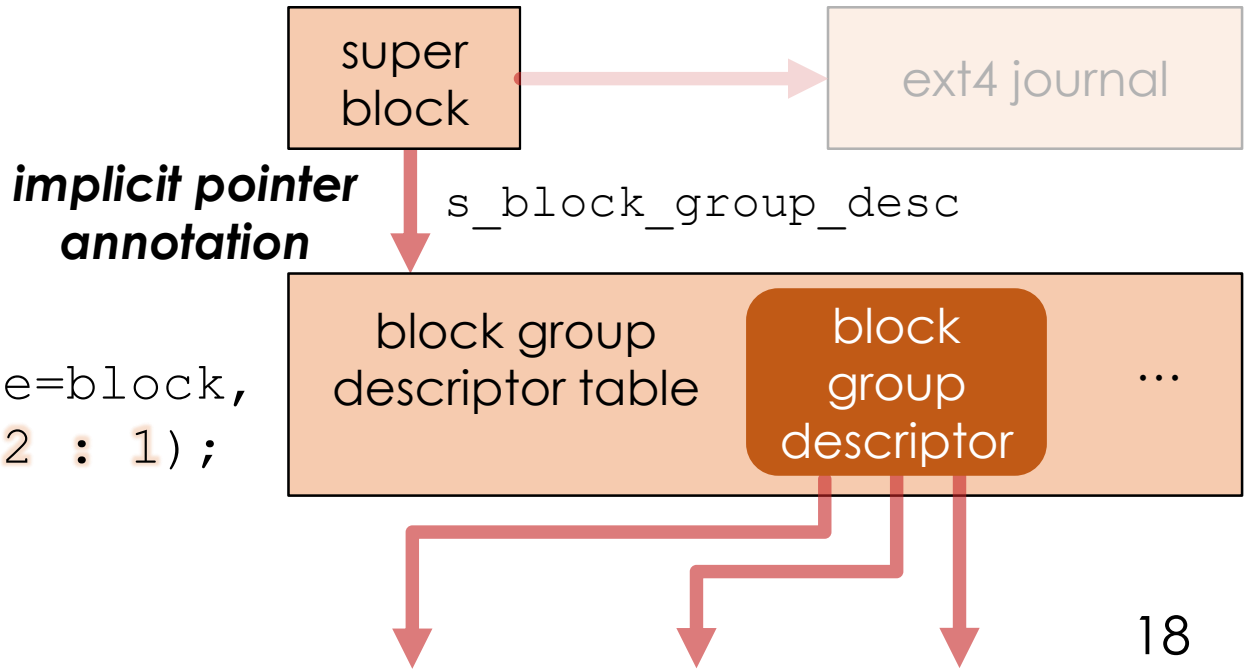
- Locations of some structures are implicit in the code
- E.g. Ext4 block group descriptor table is the next block following the super block
  - Ext4 super block does not have a field that points to descriptor table
  - Pointer required for file system traversal



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

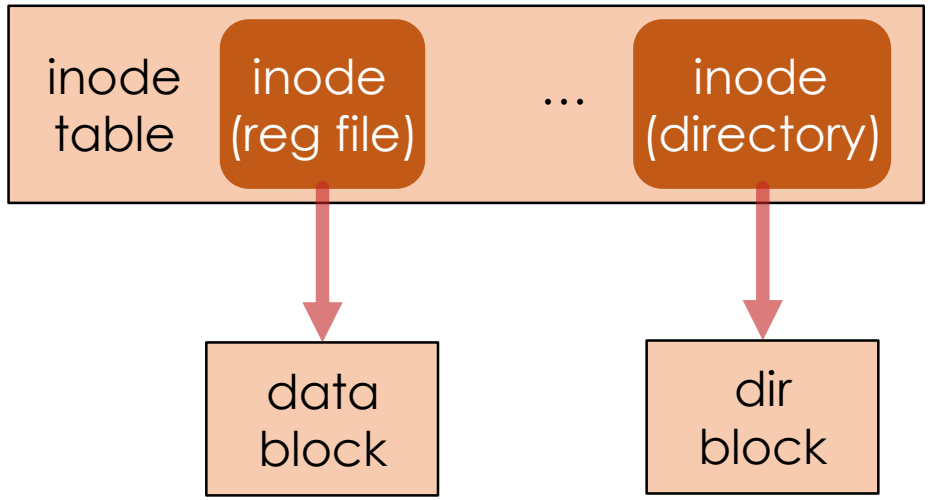
```
FSSUPER(...) ext4_super_block {
   __le32 s_log_block_size;
   ...
   POINTER(name=s_block_group_desc,
   type=ext4_group_desc_table, addrspace=block,
   expr=(self.s_log_block_size == 0) ? 2 : 1);
};
```



Context-Sensitive Types

- A pointer may point 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(addrspace=block, type=dir_block,
          when=self.i_mode & S_IFDIR)
  POINTER(addrspace=block, type=data_block,
          when=self.i_mode & S_IFREG)
  __le32 i_block[EXT3_NDIR_BLOCKS];
   …
};
```



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 parent

- 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;
}
```

Evaluation: Annotation Effort

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

- Lines of code required to correctly annotate modern file systems
  - Need to declare some structures
    - E.g. Ext4 indirect block is 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, the first 12 are direct block pointers, while the 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) {
1:  Ext4 ext4(dev);
2:  /* read super block into memory */
3:  Ext4::SuperBlock * sb = ext4.fetch_super();
4:  if (sb == nullptr) return -1;
5:  dev.set_block_size(1024 << sb->s_log_block_size);
6:  /* traverse file system and find/process all block bitmaps */
7:  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) {
1:  Ext4 ext4(dev);
2:  /* read super block into memory */
3:  Ext4::SuperBlock * sb = ext4.fetch_super();
4:  if (sb == nullptr) return -1;
5:  dev.set_block_size(1024 << sb->s_log_block_size);
6:  /* traverse file system and find/process all block bitmaps */
7:  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) {
1:  Ext4 ext4(dev);
2:  /* read super block into memory */
3:  Ext4::SuperBlock * sb = ext4.fetch_super();
4:  if (sb == nullptr) return -1;
5:  dev.set_block_size(1024 << sb->s_log_block_size);
6:  /* traverse file system and find/process all block bitmaps */
7:  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) {
1:  Btrfs btrfs(dev);
2:  /* read super block into memory */
3:  Btrfs::SuperBlock * sb = btrfs.fetch_super();
4:  if (sb == nullptr) return -1;
5:  dev.set_block_size(sb->sectorsize);
6:  /* traverse file system and find/process all extent items */
7:  return sb->process_by_type(EXTENT_ITEM,
                               process_extent_item, &vec);
  }
```

Spiffy Applications

- File System Free Space Tool
- Type-Specific File System Corruptor
- File System Conversion Tool
- File-system aware block layer cache

Type-Specific File System Corruptor

- Finds and corrupts a field in a specified structure
- Works for all annotated file systems
  - 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

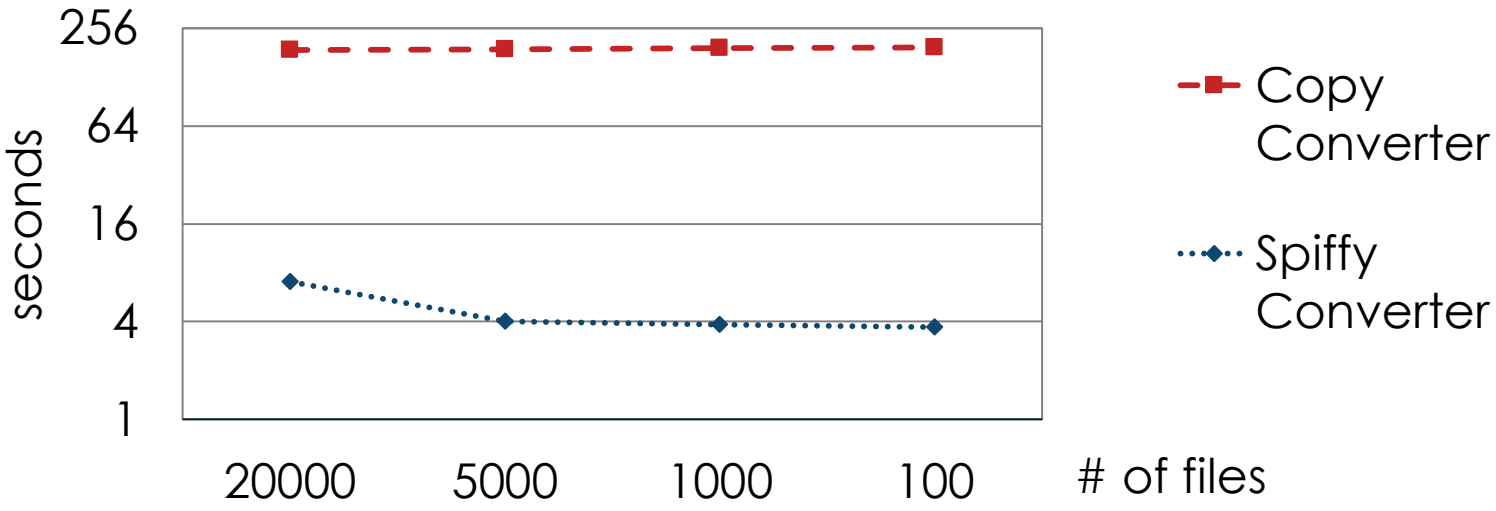
File System Conversion Tool

- Converts one file system to another while minimizing 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 copy-based converter vs. Spiffy 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 a rich set of caching policies that require file-system specific information at the block layer
  - Cache file system metadata
    - Requires knowing whether a block is data or metadata
  - Cache small files, cache a specific user's files
    - Requires knowing the file to which a block belongs, and the file's size or owner
- 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
  - Requires little file-system specific code

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

Spiffy: Enabling File-System Aware Storage Applications

Presented by Kuei (Jack) Sun

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(bool alloc=true)`	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
	`Container * parse_by_type(int type, ..., const char * buf, size_t len)`	parses the buffer *buf* as metadata container *type*
Application Developer		
`Visitor`	`virtual int visit(Entity & e)=0;`	visits an entity and possibly processes it

File-System Agnostic Traversal

```
EntVisitor ev;
PtrVisitor pv;

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

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

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

- Simple example to traverse the entire file system
- No file-system specific code required (except for bootstrap)

File-System Agnostic Traversal

```
EntVisitor ev;
PtrVisitor pv;

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

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

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

- Traversal begins from super block
- Uses two mutually recursive visitors to traverse the whole file system

File-System Agnostic Traversal

```
EntVisitor ev;
PtrVisitor pv;

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

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

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

- EntVisitor prints name of entity *e*, then calls `PtrVisitor::visit` on all pointer fields within entity *e*

File-System Agnostic Traversal

```
EntVisitor ev;
PtrVisitor pv;

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

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

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

- PtrVisitor fetches the pointed-to Container from disk, then calls `EntVisitor::visit` on the pointed-to Container

Evaluation: Programming Effort

Applications	Generic (LOC)	File-System Specific (LOC)
XML Dump Tool	565	40~50
Free Space Display Tool	271	76~194 (F2FS)
Type-Specific Corruptor	455	< 30
Ext4 to F2FS Converter	504	218 (Ext4), 1760 (F2FS)
Runtime Interpretation	2158	111 (Ext4), 134 (Btrfs)
Differentiated Storage		
- Block Layer Cache	10518	N/A
- Preferential Caching	289	N/A

- Programming effort reduced for read-only file system applications
  - Both online and offline

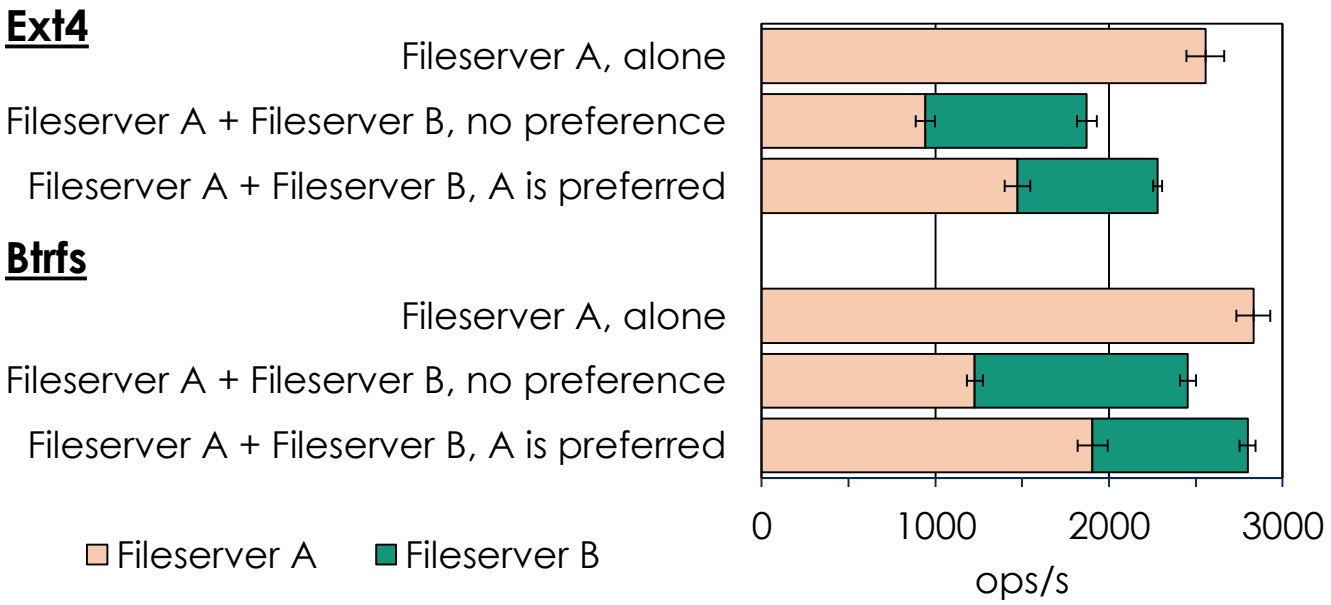
Evaluation: Corruption Experiment

- Run Spiffy dump tool and existing dump tools on corrupt images
  - Corrupt images generated by type-specific corruptor
- Spiffy is robust against corruption, found crashes on existing tools

Tool	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
	checkpoint	`cp_blkaddr`	double free error during error handling
	summary	`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

Evaluation: Preferential Caching

- 2 identical fileservers running on block layer cache

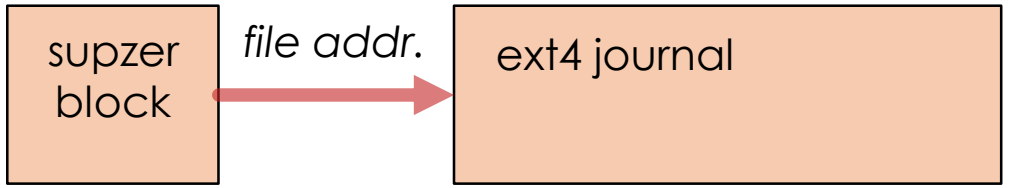


- Overall performance improves with preferential caching

FSSUPER Annotation

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

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



F2FS NID Address Space

- NID address is an index into the node address table for an F2FS metadata block