Annotation for Automation: Rapid Generation of File System Tools

Kuei (Jack) Sun, Daniel Fryer, Angela Demke Brown, Ashvin Goel
University of Toronto

Abstract

Today file system tools and file-system aware storage applications are tightly coupled with file system implementations. Developing these applications is challenging because it requires detailed knowledge of the file system format, and the code for interpreting file system metadata has to be written manually. This code is complex and file-system specific, and so the application requires significant re-engineering to support different file systems.

We propose a file system annotation language for specifying a file system's on-disk metadata format. File system developers are asked to annotate the data structure definitions of a file system's metadata. The annotated code is parsed and used by tool-specific code templates to create interpretation routines (e.g., a metadata parser) for the desired file system tool. The benefit is that different tools can reuse the interpretation routines, and they are much less dependent on file system formats and implementations. We show the feasibility of this approach by implementing a compiler that generates a runtime metadata interpreter for an annotated toy file system. The generated code has low overhead (roughly 3%) compared to a hand-written version of the same application.

1. Introduction

There are several file system and storage applications that require access to file system metadata. For example, offline file system tools access file system metadata to check and repair file system consistency [11] and recover deleted files [4]. Online applications improve storage performance or reliability by taking advantage of file system metadata at the block layer. For example, differentiated storage services [12] improve performance by preferentially caching metadata blocks. I/O shepherding [7] improves reliability by using file structure information to implement checksumming and replication. Similarly, the Recon system [6] improves reliability by verifying the correctness of metadata operations at runtime.

All of these applications require the ability to recognize or interpret file system metadata structures, in either an offline or online context. Currently these applications must interpret metadata on an ad-hoc basis, either by implementing their own methods or by leveraging libraries provided by the file system developer. For example, applications can use the libext2fs library for offline interpretation of Linux ext2/3 file system images, but the library does not support online interpretation. Similar libraries for other file systems, where they exist, provide a different interface and hence these applications have to be significantly rewritten for each file system.

PLOS'13, November 03 - 06 2013, Farmington, PA, USA.
Copyright © 2013 ACM 978-1-4503-2460-1/13/11…$15.00.
http://dx.doi.org/10.1145/2525528.2525529

Our aim is to allow developers to focus on the structure and policies of their applications, rather than the intricacies of any particular file system. To reduce the burden of developing applications that access file system metadata, we envision the separation of 1) the low-level details of a file system's format, 2) the structure of the application, and 3) any file-system or application-specific policies applied by the tool. Our goal is to enable file system developers to annotate the file system format on disk so that a metadata parser can be generated automatically, allowing applications to focus on their structure and policies. We motivate our approach by describing three use cases.

File System Checker A file system checker must be able to access metadata objects to perform diagnostics and/or repair. Developers nowadays build the consistency checker for each file system from scratch, which requires significant effort. In our approach, the consistency checkers for different file systems would use generated metadata parsers that share a common API, and will only need to implement their file-system specific rules.

The SQCK file system checker [8] implements all the file system consistency rules as SQL queries on a database containing the file system's metadata. The database is populated by a file-system specific parser that iterates over the file system metadata and generates rows to be inserted into corresponding tables. With proper annotations, a shared code template can generate the parsing routines for different file systems.

Type-Specific Storage Policies Mesnier et al. [12] describe a differentiated storage service in which different classes of data are handled with different storage system policies for improved performance. In their prototype, the file system source code was modified to identify the class (e.g. inode, directory) of each block request issued to the block layer, where the policies are implemented. Instead of requiring application developers to modify each file system, a runtime metadata interpreter can identify block classes by interpreting file system I/O at the block layer. The ability to automatically create file system interpreters allows a developer to quickly prototype the storage service for different file systems.

Runtime Verification The purpose of runtime verification is to protect metadata from buggy file system operations [6]. This application requires a framework for interpreting metadata at runtime, similar to the previous example, and in addition it needs to be able to compare the state of a file system before and after a transaction. The logical differences between the states are checked against a set of invariants to determine if a consistency violation has occurred. To generate the logical differences, the metadata is compared field-by-field, based on its parsed structure. In our approach, code templates are used to generate the functions that traverse and compare the changed structures. These templates use the annotated pointer relationships between objects to identify the corresponding structures in the old and new metadata trees.

Our annotation-based approach has several benefits:

- It enables rapid-prototyping of applications for a file system that may not have an existing parsing library.

- The generated code does not modify the file system source code, eliminating the chance of introducing file system bugs.
- The annotated structure definitions provide a concise and clear documentation of the file system's format, helping with the development of additional debugging and analysis tools.
- The generated routines can be reused across applications, further simplifying application development. For example, with n file systems and k file system tools, currently $n \times k$ programs needs to be written. Our approach decouples file systems from file system tools, reducing the effort to $n+k$ (n annotated file systems and k code-template based tools).
- Since file system formats are known to be stable over time, there is minimal cost for maintaining annotations. When format changes do occur, the specification needs modifications, which is easier than modifying all the related tools.

We evaluate the feasibility of our approach by designing a file system annotation language and a compiler that parses annotated data structures, defined in C. To facilitate evaluation, we use a simplified user-space implementation of an ext3-like file system named TestFS [16] and add annotations to the file system's metadata definitions. We use the annotated definitions to generate code that interprets metadata below the file system (i.e., at the block layer) to verify the correctness of the file system operations performed by TestFS. Our results show that the generated code has minimal performance overhead.

In Section 2, we present an overview of our file system annotation language. Section 3 describes the implementation of our system. Section 4 evaluates the performance of our implementation. Finally, we present related work in Section 5 and discuss our conclusions and future work in Section 6.

2. Annotating File System Structures

The purpose of our file system annotation language is to specify the relationships between metadata objects on disk so that they can be traversed without using hand-written code. Ideally, these relationships could be extracted from the file system code. Although the C header files of a file system contain the structural definitions for each metadata type, they are incomplete descriptions of the file system format because information may be hidden within the code of the file system. For example, in the Linux ext3 file system, the `i_dtime` field of an inode structure can mean either the deletion time of the inode, or the inode number of the next inode in the orphan list [14]. Our approach helps clarify these relationships.

After a file system developer annotates their file system's data structures, we use a compiler to parse the annotated structures and produce an intermediate representation (IR) of the file system's format, as shown in Figure 1. An application is developed by using a template-based code generator that operates on the file system IR and generates file-system specific accessor functions and parsing routines. This code forms the substrate on which high-level policies can be specified, such as file system invariants to be checked, or storage policies on specific files or types of metadata.

Analogous to source code that is parsed based on a grammar, a file system image can be parsed if there is a suitable "grammar" describing the format. Whereas parsing source code results in an abstract syntax tree, parsing a file system helps identify distinct file system objects such as inodes, directory entries, and their fields. The main difference between syntactic analysis of source code versus metadata parsing is that a source code parser processes a character stream sequentially, whereas file system metadata forms a directed acyclic graph that requires a graph traversal.

The metadata parser starts the graph traversal at the root of the tree (i.e., the super block), parses the block to find any pointers

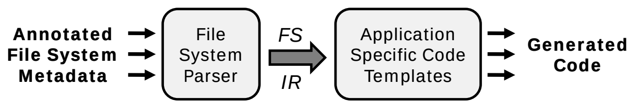
Annotated File System Metadata
File System Parser
FS
IR
Application Specific Code Templates
Generated Code

Figure 1. Annotating file system structures

to file system objects in the block, and then continues parsing the blocks containing these objects. The parser requires the type of an object to be known before it can be parsed. This type information is available in the parent block containing the pointer to the object, because the file system also needs the same information. However, this type information is implicit in the file system code. For example, the pointer relationship between two file system data structures may be implicit, as shown in Figure 2(b). We explain Figure 2 and how our annotations deal with the problem in the next subsection.

Similarly, the placement of structures on disk (e.g., an instance of structure B optionally follows structure A) may be implicit in the code that operates on them. The meaning of fields might be context sensitive (e.g., untagged unions), and some structures may not be declared at all (e.g., treating a buffer as an array of `int`s or `char`s).

2.1 Annotation Language

Our file system annotation language specifies the information needed by the parser to traverse the file system as a graph. Table 1 shows the complete list of annotations that are provided for expressing the missing type information. The annotations are written as keywords, followed by arguments. The value of each argument is an arbitrary C expression. Our language is designed to be compatible with C because the annotations are defined in the format of C preprocessor macros, which can be ignored during regular compilation.[1] In the following paragraphs, we describe each annotation in detail. Then in Section 2.2, we provide examples of their usage.

The `FSSTRUCT` annotation specifies that a data structure definition is used for file system metadata on disk. It distinguishes file system metadata from in-memory file system structures, and signals the start of an annotated block to our compiler.

To identify the root of the file system metadata tree, typically described by the super block structure, we use the `SUPER` annotation. The `location` parameter gives the super block's placement relative to the beginning of the partition, in bytes. We allow the C expressions in the arguments of other annotations to reference fields in the super block through a global variable `sb`.

The `PROPERTY` annotation allows developers to specify information on how to parse the data structure. For example, the size of an ext3 inode is specified by the super block's `s_inode_size` field, and not by `sizeof(struct ext3_inode)`. Therefore, the annotation for the ext3 inode `size` parameter is `sb.s_inode_size`. Another example is ext3's block group descriptor, which is located in the block right after the super block. Its `location` parameter is `sb.s_first_data_block+1`.

Pointers make up the edges of the metadata graph. Typically, pointer information is missing from file system metadata definitions. For example, with in-memory data structures, we know that if `struct foo` has a member variable of type `struct bar *`, then `struct foo` points to `struct bar`, as shown in Figure 2(a). However, file system developers may declare a similar on-disk data structure as shown in Figure 2(b), which tells us that a pointer value is stored as a little-endian 32-bit integer. It also suggests that the value stored is a block number, and that the pointed-to block is of type *bar*, but that conjecture is based on the variable naming conventions. We introduce the `POINTER` annotation to explicitly state the relationship between the `foo` and the `bar` types. It specifies that a field is a pointer to a metadata object of the specified type.

[1] With the exception of `FSSTRUCT`, which evaluates to `struct`.

Annotation	Description	Arguments	Meaning
FSSTRUCT	This data definition is for a file system metadata.	N/A	N/A
SUPER	This file system metadata is a super block.	repr	The default pointer representation. If `repr=byte`, pointers are absolute byte locations on disk. If `repr=block`, the pointer values need to be multiplied by `blocksize`.
		blocksize	The size of the block when `repr=block`.
		null	The default value of a null pointer.
		location	The location of the super block as an offset in bytes, from the start of the file system image.
PROPERTY	Specifies any object-level property about this type of file system metadata.	size	The actual size of the metadata object (default is `sizeof(struct` *foo*`)`).
		location	The location of this metadata object, expressed in the default pointer representation.
POINTER	This field is to be interpreted as a pointer to a specified metadata object.	repr,null	Overrides the default representation specified in the SUPER annotation.
		type	The type of the pointed-to structure or array.
		when	A precondition that must be satisfied before interpreting the pointer.
VECTOR	1. Defines a variable-length field that appends to this metadata object. 2. Defines an array type.	name	The symbolic name of the vector field or the array type.
		type	The type of the vector elements.
		size	An expression for calculating the actual size of the array.
		sentinel	The sentinel value which specifies the end of a linked list.

Table 1. File system annotation language

```
struct foo {
  struct bar *a;
};
```

(a)

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

(b)

Figure 2. (a) In-memory pointer representation (b) On-disk pointer representation

File system code commonly uses variable-sized arrays and accesses them using pointer arithmetic. We use the VECTOR annotation to express such arrays because they cannot be expressed in standard C structure definitions[2]. The VECTOR annotation helps specify the name, type and length of an array, and can be placed inside or outside struct definitions. Intuitively (in C terminology), when placed inside, it defines an implied field of a struct, and when placed outside, it defines an implied typedef type. These implied types are available to other annotations and to the code templates, without affecting the original file system code.

The language allows the keyword arguments to be arbitrary C expressions. Currently, we support using constants in the expressions, and accessing fields of the super block (referenced as `sb`) and the current metadata object (referenced as `self`) within the expression (e.g., `self.foo_ptr`), which is sufficient for our current target file system, described in Section 2.2. We plan to extend the language to allow expressing other variables (e.g., objects in parent block) and dereferencing pointer annotated fields (e.g., `self.foo_ptr.foo_field`). In addition, we expect that the set of keyword arguments shown in Table 1 may need to be extended to support more complex file systems such as btrfs.

2.2 Example: TestFS

Figure 3 shows the format of the TestFS file system, a simplified variant of the Linux ext3 file system. We have chosen to annotate

[2] C allows naming one variable-sized array located at the end of a structure, but the size of this array cannot be specified. This feature is known as flexible array member.

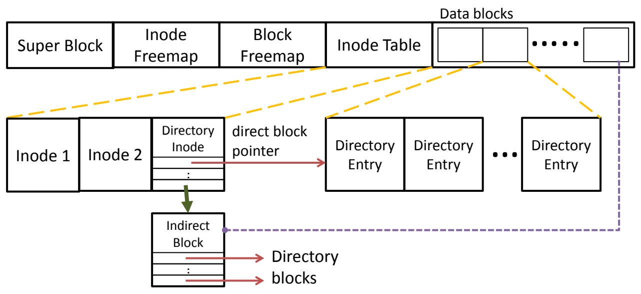

Figure 3. TestFS metadata layout. Directory blocks and indirect blocks are placed within the region designated for data blocks.

TestFS because it simplifies the discussion of annotations while being sufficiently rich to reveal their expressiveness.

The super block holds pointers to three types of metadata: inode table, inode freemap, and block freemap. Inodes in the inode table are metadata objects that represent either files or directories. File inodes hold pointers to file data blocks and directory inodes to directory blocks. A file data block is not a metadata block and thus is not interpreted. However, directory blocks are metadata and are interpreted. An inode may also have a pointer to an indirect block, which has a set of pointers to either directory blocks or data blocks. The inode freemap tracks all used and unused inodes, whereas the block freemap tracks all available blocks.

Figure 4 shows the annotated data structure definition of the super block. As an example, the `inode_freemap_start` field is specified as a pointer of type `inode_freemap`. Notice that the super block holds a pointer to data blocks (i.e., `data_blocks_start`), but this field is not annotated because the blocks are not metadata. Metadata blocks in ext3-like file systems often contain a homogeneous set of objects. The VECTOR annotations, shown in Figure 5, help specify these arrays of objects. For example, the `inode_freemap` vector type is specified as a bit vector.

Figure 6 shows the annotations on the inode structure. The `i_block_nr` field declares an array of direct block pointers. This field has a POINTER annotation with the `when` argument describing

```
FSSTRUCT super_block {
  SUPER(blocksize=BLOCK_SIZE,
    location=0, null=0, repr=block);

  POINTER(type=inode_freemap)
  const u32 inode_freemap_start;
  POINTER(type=block_freemap)
  const u32 block_freemap_start;
  POINTER(type=inode_table)
  const u32 inode_blocks_start;

  const u32 data_blocks_start;
  u32 modification_time;
};
```

Figure 4. Annotated TestFS super block

```
VECTOR(name=inode_freemap, type=bitmap,
  size=INODE_FREEMAP_SIZE*BLOCK_SIZE);
VECTOR(name=block_freemap, type=bitmap,
  size=BLOCK_FREEMAP_SIZE*BLOCK_SIZE);
VECTOR(name=inode_table, type=struct inode,
  size=BLOCK_SIZE*NR_INODE_BLOCKS);

VECTOR(name=dir_block, type=struct dirent,
  size=BLOCK_SIZE, sentinel=self.d_inode_nr==0);

VECTOR(name=data_indirect_block, type=u32,
  size=BLOCK_SIZE);
VECTOR(name=dir_indirect_block,
  type=struct dir_indirect, size=BLOCK_SIZE);
```

Figure 5. Annotated TestFS metadata blocks

```
FSSTRUCT inode {
  u32 i_type;
  u32 i_mod_time;
  u32 i_size;

  POINTER(type=dir_block,
      when=self.i_type == I_DIR)
  u32 i_block_nr[NR_DIRECT_BLOCKS];

  POINTER(type=data_indirect_block,
      when=self.i_type != I_DIR)
  POINTER(type=dir_indirect_block,
      when=self.i_type == I_DIR)
  u32 i_indirect;
};
```

Figure 6. Annotated TestFS inode

```
FSSTRUCT dirent {
  u32 d_name_len;
  s32 d_inode_nr;
  VECTOR(name=d_name, type=u8,
         size=self.d_name_len);
};
```

(a)

```
FSSTRUCT dir_indirect {
  POINTER(type=dir_block)
  u32 ind_block_nr;
};
```

(b)

Figure 7. (a) Annotated directory entry (b) Annotated directory indirect pointer.

when the value can be interpreted as a pointer of the specified type. When an inode is not a directory (i.e., `self.i_type != I_DIR`), the direct pointers do not point to metadata, and are therefore not annotated. Otherwise, the direct pointers point to the `dir_block` type, specified in Figure 5. The `i_indirect` field can point to either a data or directory indirect block, and so the field has two `POINTER` annotations. The referenced types are vectors, as defined in Figure 5.

In TestFS, directory blocks contain variable-length directory entries. Figure 7(a) shows the annotation for a directory entry. The `VECTOR` annotation, declared here within a structure (unlike the definitions in Figure 5), is used as a replacement for C's flexible array member so that the length of the array can be specified. It assigns the label `d_name` to this field so that it can be symbolically referred to just like the `d_name_len` and `d_inode_nr` fields. Within a directory block, the file system sets the `d_inode_nr` field of the last directory entry to 0. In Figure 5, the `sentinel` argument in the `VECTOR` annotation of `dir_block` is used by the code templates to end the parsing of directory entries, so that the parser does not attempt to interpret unused bytes of data.

A file system may not have definitions for all of its data structures. For example, since the indirect blocks of TestFS are just an array of four byte pointers, the TestFS developer chose to omit its definition. However, these implicit data structures need to be defined so that TestFS can be specified unambiguously. Figure 7(b) shows the newly defined structure with the appropriate annotation.

3. Implementation

We have implemented a compiler called `jdc` (Jack and Daniel's Compiler) that enables parsing the annotations described in Table 1. It uses Python Lex-Yacc (PLY 3.4) [3] as its parser generator and lexical analyzer. The grammar, written in Yacc, is based on the ANSI C grammar with the addition of annotations. The compiler only parses the `VECTOR` annotations and data structure definitions tagged by the `FSSTRUCT` annotation in C code and keeps all arbitrary C expressions in text form (e.g., in the `when` argument of the `POINTER` annotation). These expressions are assumed to be syntactically and semantically correct. We also assume that file system developers use types of known size (e.g., `u32` instead of `long`) in structure definitions to avoid portability concerns.

The compiler is invoked with a set of header files that contain metadata definitions (e.g., `jdc -name testfs super.h extra.h dir.h inode.h`). It generates the file system's internal representation in the form of a symbol table, which contains 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, semantic analysis is performed to detect errors such as duplicate declarations or missing arguments. Finally, the symbol table and compiler options are exported for use by code templates.

3.1 Code Generation

We generate file-system specific metadata interpretation code using the powerful Django template language [9], which was originally designed to generate dynamic HTML content. The code generator works by embedding Django's template filters and tags directly into C source code. Figure 8 shows an example template. As a primer, the set of double brackets (e.g., `{{ fs_name }}`) allows text substitution for the string value of the variable. The statements inside a set of brackets followed by percent signs (e.g., `{% for st in structs %}`) are used to specify control logic.

Figure 9 shows a parse function for a TestFS inode object generated by one iteration of the `{% for st in structs %}` loop shown in Figure 8. This function prints each field of the inode

```
{% for st in structs %}
static void
{{ fs_name }}_{{ st.ident }}_parse(FILE * fsimg, const {{ st.type }} * self, int size) {
  {% for fd in st.fields %}
    {% include "print_field.c" with field=fd struct=st %}
  {% endfor %}
  {% for fd in st.fields %}
    {% if fd.pointer %}
      {% include "chase_pointer.c" with field=fd struct=st %}
    {% endif %}
  {% endfor %}
}
{% endfor %}
```

Figure 8. An Django template for generating the metadata dump tool.

```
static void testfs_inode_parse(FILE * fsimg, const struct inode * self, int size) {
  printf("i_type = %u\n", self->i_type);
  printf("i_mod_time = %u\n", self->i_mod_time);
  printf("i_size = %u\n", self->i_size);
  for ( int i = 0; i < NR_DIRECT_BLOCKS; i++ )
    printf("i_block_nr[%d] = %u\n", i, self->i_block_nr[i]);
  printf("i_indirect = %u\n", self->i_indirect);

  for ( int i = 0; i < NR_DIRECT_BLOCKS; i++ ) {
    if ( self->i_block_nr[i] != 0 ) {
      if ( self->i_type == I_DIR ) {
        const char * blkdata = read_block(fsimg, self->i_block_nr[i]);
        testfs_dir_block_parse(fsimg, (const char *)blkdata, BLOCK_SIZE);
      }
    }
  }
  if ( self->i_indirect != 0 ) {
    if ( self->i_type != I_DIR ) {
      const char * blkdata = read_block(fsimg, self->i_indirect);
      testfs_data_indirect_block_parse(fsimg, (const char *)blkdata, BLOCK_SIZE);
    } else if ( self->i_type == I_DIR ) {
      const char * blkdata = read_block(fsimg, self->i_indirect);
      testfs_dir_indirect_block_parse(fsimg, (const char *)blkdata, BLOCK_SIZE);
    }
  }
}
```

Figure 9. A parse function generated by the template shown in Figure 8.

object using the `print_field.c` template that contains code for printing the value of arbitrary types, including arrays. When a field is annotated as a pointer, then the `chase_pointer.c` template generates code that reads the referenced block and invokes the correct parse function depending on the type of the referenced block. Next, we describe the code templates for three applications that we have written:

File System Dump Tool The offline dump tool, partly shown in Figure 8, parses metadata in a file system image. The template generates parsing functions for each type of file system metadata. The parsing functions work by iterating through each metadata block and displaying all the fields of each metadata object. Whenever a pointer is encountered, the parser chases that pointer in depth-first order by invoking the parsing function for that pointer. We plan to build a file system checker based on this dump tool.

Runtime Metadata Interpretation This template generates interpretation functions for each type of file system metadata that references other metadata. For metadata leaf nodes (e.g., containing a directory entry), no interpretation functions are generated. When the application intercepts file I/O at the block layer, it uses the interpretation functions to iterate over and record the types of all pointers in the block, so that a later file system I/O for the pointed-to blocks can be interpreted.

Metadata Differencing A metadata difference engine [6, 16] takes the old and new versions of a metadata object and identifies changes made by a write operation. The template generates comparison functions for each type of file system metadata block. Currently, this code template relies on handwritten functions to associate metadata objects with a specific identity in order to match old and new versions of an object. Additionally, journaling file systems do not update blocks in place, requiring us to interpret the journal before metadata can be interpreted. We plan to fix these limitations by introducing annotations to specify the identity of metadata objects and the format of journals.

4. Evaluation

We reimplemented the Recon runtime verification system [6] by replacing various modules with generated code, and comparing the performance of hand-written versus generated code. Our code template generates routines for interpreting metadata at runtime and for performing the metadata differencing operations. The benchmark consists of 250,000 file system operations, including creating

	Handwritten	Generated	Overhead
User	35.9±0.1s	37.2±0.1s	3.6%
System	22.5±0.1s	23.0±0.1s	2.2%
Sleep	546.3±8.2s	550.8±8.1s	0.8%
Total	604.7±8.3s	611.0±8.2s	1.0%

Table 2. Performance of the Recon runtime verification system using hand-written vs. annotation-generated code

files and directories, deleting files and directories, writing random data to files, and changing the current working directory. Our test machine has an Intel Xeon X430 quad-core processor at 2.4 GHz with 4GB of RAM; it is running Debian Squeeze Linux version 6.0, kernel version 2.6.32. Since TestFS uses a regular file as its storage device, we open this file with the O_SYNC flag to mimic the characteristics of a write-heavy file system by forcing writes to disk. We ran a total of 22 tests, and removed the fastest and the slowest run from each set. We also verified that the outputs of both implementations are identical.

Table 2 shows that the generated code has minimal overhead. The difference in sleep time is insignificant. Since Recon for TestFS runs in userspace, we expect the modest increase in user time. We are surprised by the increase in system time, but we have not investigated this effect.

5. Related Work

There have been several prior works on specifying binary serialization formats [10, 15, 17]. However, developers have control over the serialization format of their protocols whereas file system formats are fixed by the file system developers and not the application developers. Therefore, parsers generated by these languages cannot interpret file system I/O, which perform a graph traversal rather than a sequential scan. Our annotation language overcomes this limitation by making the pointer information explicit, which defines how metadata objects reference each other. The PADS project [5] has similar goals to this work, but it also assumes that all objects are accessed sequentially.

There have been several attempts to extend C to express more semantic information [1, 13, 18]. CCured [13] enables type and memory safety, and the Deputy Type System [18] prevents array out-of-bound errors. Both projects annotate existing source code, perform static analysis, and add runtime checks, but they are designed for in-memory data structures.

Amani et al. [2] tackles the challenge of fully verifying the correctness of a file system's implementation. Their approach uses static analysis and requires building a new file system from scratch. In contrast, our approach generates runtime verification frameworks for existing file systems.

6. Conclusions and Future Work

We have developed a file system annotation language for the on-disk format of file system data structures. The annotated structures can be compiled to generate application-specific tools for rapid prototyping. We have shown that the generated code incurs minimal performance overhead over hand-optimized code. Overall, we believe this technique will enable many interesting applications that are based upon parsing of file system metadata.

In the future, we plan to annotate real Linux file systems such as ext3 and btrfs. We plan to complete the implementation of the compiler to support all the annotations. Then we plan to build several file-system tools and applications that can use these annotations. We are also looking to extend the language so that metadata invariants can be expressed or derived from the specification.

References

[1] Sparse - a Semantic Parser for C. `https://sparse.wiki.kernel.org/index.php/Main_Page`.

[2] S. Amani, L. Ryzhyk, and T. Murray. Towards a fully verified file system, 2012. EuroSys Doctoral Workshop 2012.

[3] D. Beazley. Ply (python lex-yacc), 2013. `http://www.dabeaz.com/ply/`.

[4] B. Buckeye and K. Liston. Recovering deleted files in linux. *Retrieved February*, 2006.

[5] K. Fisher and D. Walker. The pads project: an overview. In *Proceedings of the 14th International Conference on Database Theory*, pages 11–17. ACM, 2011.

[6] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage*, 8(4):15:1–15:29, Dec. 2012.

[7] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.

[8] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, Dec. 2008.

[9] A. Holovaty and J. Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[10] G. Inc. Protocol buffers - google developers. `https://developers.google.com/protocol-buffers/`.

[11] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2013.

[12] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 57–70, 2011.

[13] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[14] Ryoichi Kato. Ext3 orphaned inode problem. `http://tree.celinuxforum.org/CelfPubWiki/Ext3OrphanedInodeProblem`, July 2007.

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

[16] J. Sun, D. Fryer, A. Goel, and A. D. Brown. Expressing invariants for protecting file-system integrity. In *Proceedings of the Workshop on Programming Languages and Operating Systems (PLOS)*, 2011.

[17] T. Weigert and P. Dietz. Automated generation of marshaling code from high-level specifications. In *SDL 2003: System Design*, pages 374–386. Springer, 2003.

[18] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60. USENIX Association, 2006.