Comprehensive Robustness Evaluation of File Systems with Model Checking

Jingcheng Yuan
School of Information Science
Japan Advanced Institute of Science and Technology
Tokyo, Japan
jingcheng.yuan@jaist.ac.jp

Toshiaki Aoki
School of Information Science
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
toshiaki@jaist.ac.jp

Xiaoyun Guo
School of Information Science
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
xiaoyun@jaist.ac.jp

***Abstract*—File systems are used to organize data on storage devices. The file systems may crash due to external failures, such as an unexpected power outage. Therefore, the robustness of the file system is essential. Although some existing works evaluated the robustness of file systems, they are not comprehensive enough and cost many resources. In this work, we design a file system model and verify properties related to the correctness of the file using the SPIN model checker. The robustness of the file system has been comprehensively evaluated in both single-thread and multi-thread modes. There is a critical error in the file system. By analyzing counterexamples given by model checking, we propose a mechanism to prevent it. Based on the mechanism, the robustness of the file system is effectively improved.**

Keywords—file system, robustness, power outage, model checking, SPIN, Promela

I. Introduction

File systems are widely used to organize data on storage devices. A file system separates the data into pieces that we call them files. The files make it easy to isolate and identify data. Generally, the file system organizes these files in a tree structure. The files are the leaf nodes in the tree, while the non-leaf nodes indicate directories in the file system.

Usually, a file system works as a kernel module in the operating system, as well, the operating system also organizes its system files in it. If a severe error occurs in the file system, massive data loss, even the operating system may crash. There are some reports about the file system data loss. For example, the 2009 “ext4 data loss” incident, where multiple users reported that “pretty much any file written to by any application,” becomes empty after a system crash [14], [23]. A very likely reason for the file system error is an external failure, such as unexpected power outage or data corruption in the storage device.

Like the 2009 “ext4 data loss” incident, file operations were interrupted by a power outage or a system crash. It is essential to investigate the file systems’ robustness in the external failure condition, which includes data corruption in the storage device and interrupted file operations. Due to the help of Error Correction Code (ECC) technology, most of the modern storage devices can correct the data corruption or inform the data corruption to the file system, which works on the storage device. It allows the file system to detect the data error and prevent the further inconsistent of other files. On the other hand, the interruptions of operations, like the system crash or unexpected power-outage, makes the data inconsistent directly. The file system is challenging to detect such kind of inconsistent without external tools like *fsck*. However, the file system check tools usually need to scan all the storage, which may take tens of minutes when booting up. Some file systems introduce a journal feature to speed up the checking and repairing process. However, the file system needs to write the metadata several times to confirm a consistent state. This behavior does not only reduce the performance of writing files but also consumes more storage live especially for the NAND Flash.

Moreover, there are various types of file systems used in different applications. These file systems implement different data structures and algorithms. For example, the NTFS file system and the ext serials file systems manage their files in an index table. They are widely used in Windows or Linux systems. Like the FAT/FAT-32 file system maintains a global block link table to manage its files. This file system is widely used in mobile applications because of easy to implement. However, there are fewer present works discussed the file systems’ robustness from a comprehensive viewpoint. Some researches like [3], [5], [6] proved the file system’s correctness but did not involve the external failures. Some researches like [4], [6], [7] discuss the file systems’ robustness, but they focused on recovery tools. Generally, the recovery tool depends on the individual file systems.

In this work, we tried to comprehensively investigate the file system’s robustness in the condition of an unexpected power-outage. To cover most of the various file systems, we analyzed some mainstream file systems source code. According to how the file systems map their file’s logical address to the storage’s physical storage address, we can divide them into two types, the link type, and the index type. We developed a link-type file mapping model and an index-type model. Then we checked each model in the single thread and multi-thread mode.

We used the model checking methodology to evaluate these filesystem modes. Model checking is a formal verification technique. Compare to the software testing or other verification methodologies. It can reveal even the subtle errors by exhaustively exploring all possible system states and examining all possible scenarios. In this work, we use the SPIN model checker to find out some corner-case errors. The SPIN is a tool for evaluating the correctness of concurrent software models in a rigorous and mostly automated fashion [18]. SPIN's verifier can detect errors, for example, accessing an array out of bounds, a deadlock status, or user-defined assertion. When an error is detected, SPIN reports the error trail, which is easy for analyzing and debugging. Models to be verified in SPIN are described in Promela (Process Meta Language), which supports the modeling of asynchronous distributed algorithms [18]. The Promela also supports nondeterministic selections in *do-od* and *if-fi* statements. Comparison to other model checking tools, SPIN allows us to use C codes in the Promela directly. Since most file systems are designed in C codes, embedding C in Promela makes us easy to create file system models.

Power outage failures are usually caused by external events that are uncertain. The traditional software testing is challenging to find out all potential failures which are caused by uncertain conditions. Using the SPIN, we can develop an environment model that exhaustively simulates the external conditions. The file system is positive; it is driven by the user, which inputs the file operations to the file system. To simulate and exhaust all the possible inputs, we designed a tester model in SPIN. A correct file system should hold several properties. We defined four properties that show the file systems' correctness. We separate them into critical properties and non-critical ones. Violating the critical properties may cause data corruption in the file system while violating non-critical properties not. Usually, executing the model checking once can only evaluate one property. In this work, we use the embedded C code in the SPIN to design an on-line checker and an off-line checker. Using these two checkers, we can evaluate all properties in one execution.

Using the SPIN verifier, we detected some errors in our file system models. We also provided a method to fix these errors by analyzing the counterexamples' trace generated by SPIN. We used the SPIN to evaluate that there are no critical errors in the fixed file systems. The result shows that these fixes can improve the file systems' robustness when the unexpected power outage happened.

The first contribution of this work is to evaluate the file systems' robustness comprehensively. Our comprehensive evaluation comes from the following three viewpoints. 1) We evaluate the file systems work not only in the single-thread mode but also in the multi-thread mode. 2) We evaluate the file systems against multiple properties that cover various kinds of robustness expected to them. 3) The file systems that we evaluate cover most mainstream file systems.

The second contribution is that we successfully found corner-case errors by model checking. The model checking covers all the possible input and all the conditions of an unexpected power-outage so that we can exhaustively evaluate the robustness of the file systems in the external failures. We also need to abstract the models so that the model checking can be completed within an acceptable duration.

The third contribution is to improve the file systems' robustness according to the corner-case errors. We confirmed that the corner-case errors which were found in the abstracted models happen in real file systems as well. Thus, we proposed a mechanism which does not require to use external tools like *fsck* nor additional operations like writing journals. We also verified that no critical errors did not exist anymore in the improved file systems.

In the following sections, first, we introduce some of the existing works about the file system robustness in section II. Next, in section III, we introduce the file system and present how we abstract a model from the concrete file systems. Moreover, we explain the fundamental methods of checking the file systems' model. In section IV, we show how to evaluate the file system model. We present the counterexamples we detected in both single-thread mode and multi-thread mode. As well, we introduce the mechanism of the error fixing. Finally, we give a summary of our contribution and the advantage of this work in section V.

II. Related Works

There are some researches on file system robustness, such as [3]-[8]. Some of them proved the correctness of the file system, some of the works investigated the file system's robustness, but they focus on how to recover the file system after the external failure. We think they are still not enough on the file systems' robustness with the external failure in the case of concurrent access.

Arkoudas, Zee, Kuncak & Rinard [3] present a correctness proof for the file system implementation with standard data structures and fix-sized storage. It uses the Athena theorem prover and employs a constructive approach for verification. Different from our work, this work only involves a typical case in single-thread mode. It does not deal with the issue caused by external failure or concurrency process.

Galloway, Luttgen, Muhlberg & Siminiceanu [5] use the model checking method to verify the file system model, which is abstracted for the Linux virtual file system source code. It checks some safety and liveness properties of file system APIs in the multi-thread case. However, the Linux virtual file system is treated as a file system framework in the Linux kernel. The Linux virtual file system does not involve the detail data structure of concrete file systems on the storage device. Different from our work, this work does not deal with data consistency and external failure related errors.

Yang, Sar, Twohey, Carda & Engle [7] use a symbolic execution for generating pathological test cases. Then it checks if the file system can recover from the pathological data. Yang, Twohey, Engler & Musuvathi [4] use model checking within the systematic testing of some concrete file systems. The verification system runs the Linux kernel, a file system test driver, and a permutation checker. The checker verifies that a file system can always recover by a recovery tool *fsck*. Gatla et al. [8] use some benchmark data to verify the file system recovery tool. It simulates an external failure happens in each step of the recovery process, and then checks if the recovery tool can recover again from the external failure.

The above three pieces of works investigated the file systems' robustness in external failures and recovery. However, they focus on the recovery process instead of the file system itself.

Different from these works, our work focus on the robustness of file systems' data structure. We want to find a mechanism that can keep data consistency during external failure. Because usually, the recovery tools are also stored in the file system and may be damaged in external failures. Moreover, we want to use the model checking to show the absence of errors in the robust file systems.

III. Modeling File Systems

A. Introduction of file systems

A file system is a part of an operating system in the modem computer system. It is used to organize the data on storage devices in serval files. Files are stored in directories. The directories have recursive structures, which can contain some sub-directory. It means that the file system organizes files in a tree structure. Files are the leaf nodes in the tree, while the non-leaf nodes indicate directories in the file system. Each file or directory has a unique identifier consisting of a string, which is called file name or directory name. The string, which is called a file path contains directories' names from the root directory to the file. From a logical viewpoint, a file can be treated as an array of logical blocks. The data in each file can be located by a pair of the file path and the internal-offset. In the real storage device, the file is stored in several physical blocks. Usually, these blocks are not adjacent. The file system can be treated as a mapping from the pair of the file path and logical block index to a physical block address in the storage device. The mapping is dynamic, When the user creates or appends data to a file, the file system finds unused physical blocks, assigned the blocks to the file and then update the file mapping to related the physical blocks to the logical blocks.

The algorithm of the file mapping depends on a concrete file system implementation. Among various file systems, there are two significant algorithms, link-type file mapping, and index-type file mapping. The link-type file mapping is used in FAT, FAT32, and other file systems. The data blocks of a file are represented by a chain of blocks. These data blocks are not necessarily stored adjacent to one another on the storage device. The file system maintains a global link table called a file allocation table (FAT) to manage each file's block-chain. The table contains entries for each block. The FAT does not indicate only the file block mapping, but also the blocks states such as in-use or free. Each entry contains either the number of next cluster in the file or else a marker indicating the end of the file or unused storage space. The file entry in the directory records the address of its first logical block. The file system can then traverse the FAT, looking up the physical block address of each logical block in the file [15].

Index-type block mapping is used in ext2, NTFS, and other file systems. The data blocks of a file are treated as a resizable block array. Each file maintains a local index table to manage the physical blocks. In the index table, each entry points to a physical block that the file owns. The order of the entries in the index table also indicates the data blocks' order of the file. In order that we can efficiently use the space, the index table is organized hierarchically in most of the concrete file systems. In this case, some entries point to data blocks directly, and others point to indirect index blocks. Moreover, the entries in the indirect index blocks can also point to the double indirect index block. The file system also needs to maintain a global block bitmap, in which each bit indicates a block is in-use or free.

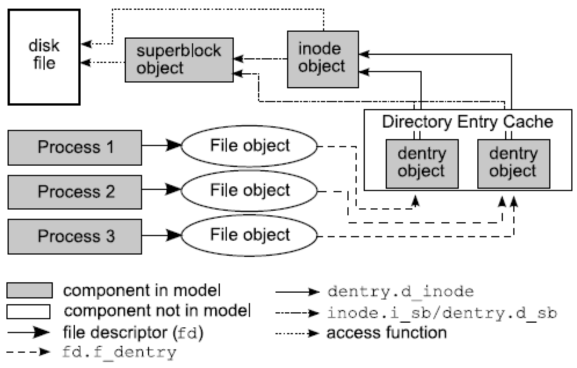

Fig. 1. Data structure of a file system.[9]

There are three necessary data structures to construct a file system, *superblock*, *dentry*, and *inode*. Fig. 1 presents an outline of the file system's data structure. The *superblock* describes the comm properties of a file system, such as its total size, mount point, and a pointer to the root directory. Usually, it is stored at the beginning of the file system, and there are several copies of the *superblock* for backup. The *dentry* objects are stored in directory files to describe the sub-directories or files which the directory contains. The *dentry* structure contains a sub-directory or file's name, a link to its parent and siblings, and some other information. It also carries a reference to its corresponding *inode*. The *inode* data structure carries information specific to a file, includes file size, file permissions, time information, and file attributes. For the link-type file mapping model, the *inode* contains the head of the block-chain. The concrete FAT file systems omit the *inode* structure and store the related information into the file's *dentry* structure. For the index-type file mapping model, the *inode* also contains the top level of the file's index table.

B. Approach to the modeling

In this work, we created a Promela model to check the file systems. Promela (Process Meta Language) is a model language that is used in the model checking tool SPIN. It supports the modeling of asynchronous distributed algorithms as non-deterministic automata [18].

The model consists of 4 sub-models, as shown in Fig. 2. The file mapping model is what we need to check. The storage model is used to simulate the storage device where the file system stores its data. The tester model is used to generates input and triggers the file system module work. We used an environmental model to simulate the unexpected power outage. We describe the detail of these sub-models in the remainder of this section.

However, to check the file system model, we still need to solve some problems. First, a concrete file system is complex and has a large scale. To check the full file system results in a

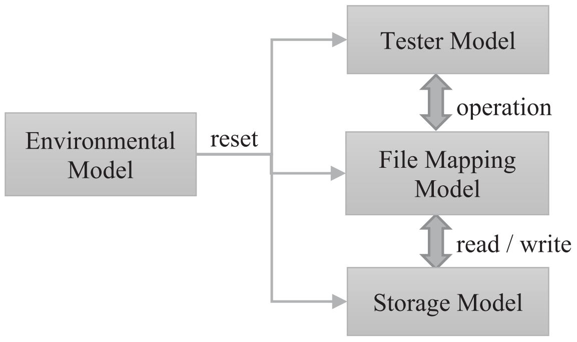

Fig. 2. Presents an overview of the models

state explosion issue in the model checking. We need to abstract it from the concrete file system. Second, the file system is a passive module in the operating system. We need to generate inputs to trigger the file system work. The inputs should be exhaustive. Third, we need to define the correctness of the file system and check these properties during the model checking. In the following sections, we describe the detail of these items.

- File mapping model

A file system is a large scale data-intensive system. Currently, a concrete file system can support gigabyte to terabyte capacity. It means a file system contains several million or billion blocks with 4KB size each. We can ignore the user data in the model checking. Even there is 1% of system data needs to be checked, we still need to travel a considerable state space.

At first, we designed a scaled file system model which contains only 128 blocks with 16 bytes each. We also designed a tool to search the file system's state space from the initial state. Then we soon faced a state explosion issue. We cannot complete the searching for an acceptable duration. Fig. 3 presents the increase in state number vs. searching depth. The x-axis indicates the searching depth, while the y-axis indicates the checked state number. The solid line indicates the actual checked state number where the dot line curve of the exponential fitting of the checked state number. This result shows that the state number increases faster than an index increase.

Regarding the reason for the state explosion, we consider it is because that the directory has a recursive structure. We checked the whole file system model, including the files and directories which can contain some sub-directories in the file system. When we expand a state, the quantity of its successor states depends on the directories and files number in the current state. For example, we create a sub-directory in each existing directory; each operation results in a new successor. In each new state, the directory number is one more than which in their parent state, so their successors also increase one. When we repeat this operation, the state number increases faster than *Exp(n)*, where *n* is the depth for searching.

As we discussed, a file system can be treated as a mapping from the pair of the file path and block offset to a physical block address. This mapping can be separated into two layers. The top one is a mapping from the file path to file, while the lower one is from the pair of the file and offset to a physical block address. The lower layer, we call it file mapping is the primary feature of file systems. Usually, the file path mapping layer is also built on the file mapping layer. File systems store the directory information in some internal files. In the robustness viewpoint, the file mapping layer is more critical. In this work, we abstract the file system as a file mapping. Then, we checked the robustness of both link-type and index-type file mapping models.

We created both link and index types file mapping models based on concrete file systems' source codes. For the link-type file mapping model, we refer to the source code of "FAT 16/32 File IO Library v2.6"[21]. Since our file system model has a fixed scale, we omitted the *superblock* and hard-coded inherent properties, such as total size, block size. We removed the directory structures in the abstract model. We omitted the *dentry* structure and designed four fixed *inodes* to describe max to 4 files. We also omitted file attributes and time information which are not interested in the robustness investigation. Each *inode* contains 2 bytes, 1 byte for the file length, and 1 byte for the start block address. We used two blocks to store a total of 4 *inodes*. In the concrete FAT file system, there are two mirrored FATs for backup. We omitted one backup table and designed one FAT. The FAT takes three blocks with a total of 12 entries to indicate the remained 11 data blocks. Hence, the abstracted link-type file mapping model has a total of 16 blocks with 4 bytes each. It can contain max to 4 files with a total of 11 blocks of user data.

We designed four operations, *CreateFile*, *DeleteFile*, *WriteFile,* and *ReadFile,* for each file system model. We abstract the operations from the source code and convert them to a state machine. Then we describe the state machine in the Promela model. Moreover, we describe the state transitions and nondeterministic selections in Promela language and described the data structures and the states' internal process in C code.

Fig. 4 presents the state machine of the *CreateFile* operation in the link-type file mapping model. When entering the operation, we check if the requested file has been created by check the file's inode. If the file does not exist, we start creating a file. We search the FAT and try to find an empty block. If there is an empty block found, we assigned the block to the file by setting the start block address field in the file's inode and mark the block as in-use in the FAT. Then, we write back the updated inode and FAT to the storage device. Finally, we update the related reference file (describe it in section III-E) and issue a FLUSH command (described in section IV-B) to close the file. In each state, we check if the file system is reset by the

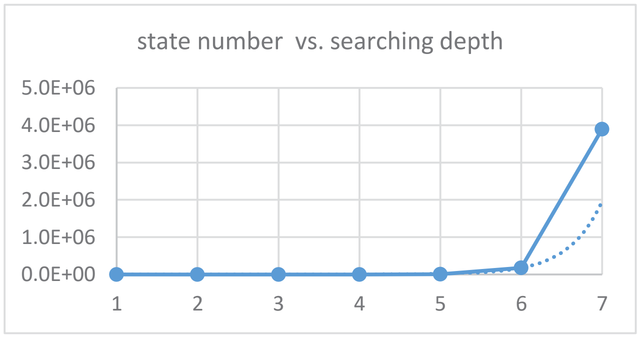

Fig. 3. State number for searching the file system model

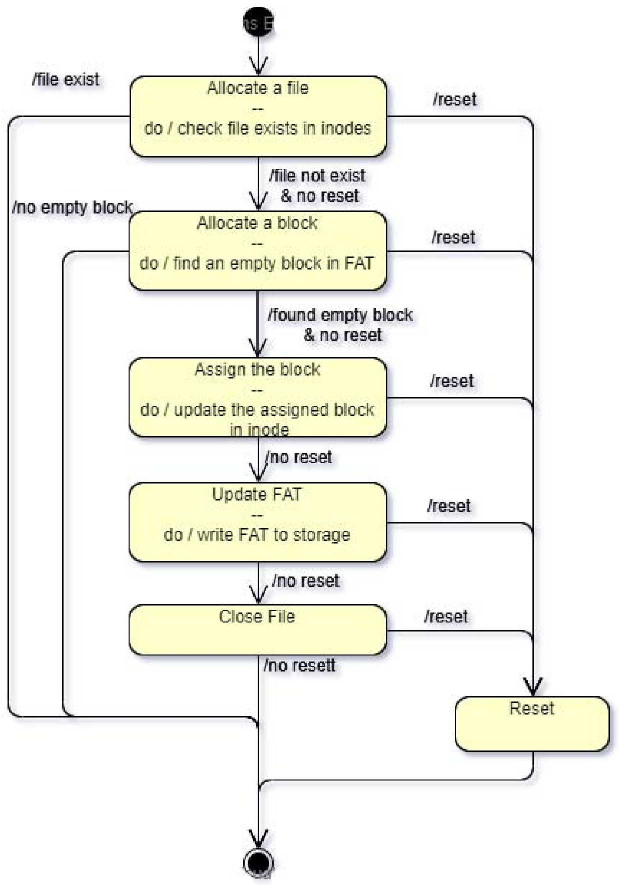

Fig. 4. State machine of CreateFile for the link-type file mapping model.

environment model. If it is reset, we discard the current operation and reset the file system.

For the index-type file mapping model, we refer to the ext2 source code from the Linux kernel. Similar to the link-type file mapping model, we omitted the *superblock* and *dentry* structure. We also designed four inodes, each of them takes one block, contains 1 byte for the file length, and 3 bytes for the block index table. The index file mapping model supports a 2-layer index table. The 3rd entry in the inode points to an indirect index block, which is dynamically assigned from the data blocks. We also designed a bitmap block to indicate on block's status, assigned or free. Each bit in the block indicates one data block. The abstracted index-type file mapping model also contains max to 4 files with a total of 11 data blocks. The data blocks are shared with file data and indirect index tables.

The index-type file mapping model has the same operations as the link-type model. These operations also have similar behaviors except assigning and reclaiming blocks. When we assign a block to a file in the *CreateFile* or *WriteFile* operations, we calculate whether the block belongs to the direct index table or the indirect index table by the block offset in the file. If the block belongs to the direct index table, we find an empty data block from the bitmap and set its entry in the index table in the file's inode. If the block belongs to the indirect index table, we need to check if the indirect index block is assigned. If not, we allocate an empty block for the indirect index block and set its entry in the inode. Then we allocate another empty block for the data block and set its entry in the indirect index block. When we reclaim blocks in the *DeleteFile* operation, besides the data blocks, we also need to reclaim the indirect index block if it exists.

- Tester model

The file system works as a passive module in the operating system on the computer. It responds to the request from client applications or user operation. To check the file system model, we should issue operations to the file system and let it execute. In most file system testing tools, they designed several operation sequences previously, which we call a test script. Then feeds these scripts to the file system and check if the file system's response expected or not.

Such previously designed test scripts are not enough for this work. It cannot exhaust the file system's state space, and it is difficult to find some corner-case error. The file system has its internal state; each file operation transfers the file system from a state to another. Even we invoke the same file operations in different orders. It makes the file system transit to different states. For example, there are two operations *WriteFile(1)* and *DeleteFile(2)*, which write data to file 1 and delete file 2, respectively. In the user's viewpoint, these two operations work for two independent files and generate the same result regardless of their order. However, inside the file system, *WriteFile(1)* may allocate a new block for file 1, and *DeleteFile(2)* reclaims blocks that are used for file 2. Invoking *DeleteFile(2)* and *WriteFile(1)* in the different order makes the file system allocates different blocks for file 1, which results in different file system's states. When an unexpected power outage happens during the two operations, different orders may result in some different subtle errors.

In this work, we searched the whole state space of the file system by exhaust all possible file operations to find out corner-case errors. In this work, we designed a tester model, which generates all possible operations for each file system state. It invokes these operations to the file system and lets the file system transit to respective successor states. Then the tester model generates and invokes all possible operations for each successor state. Repeating these steps from the initial state, we can travel the whole file system's state space. In our tester model implementation, we use Promela's nondeterministic statements *do-od* and *if-fi* to exhaust all possible operations. The tester model also supports concurrently accessing by running multi tester processes at the same time.

- Storage model

The storage model is used to store file system data like the storage device in the actual system. Our storage model is treated as a block device, which has 16 blocks with 4 bytes for each block. The storage model supports block read and block write operations. Considering the real storage device has a parity code to hold the data integrity for each block, the read and write operations in our storage model work atomically. This abstraction can help to reduce the model scale. Since the storage's density and capabilities are fixed and hard-coded, we omit the features for handshaking operations.

TABLE I. SUMMARY OF THE FILE SYSTEM PROPERTIES

Error mode		Description	Result	Error level	Recover	How to check
File system	Dead block	Block is allocated but not assigned to a file	Make valid capacity loss, no data corruption	Non-critical	Yes	Off-line
	Lost block	A block is assigned to a file but not allocated	Causes file data corruption Causes block double pointed	Critical	No	On-line and Off-line
	Double pointer	More than one pointer points to a block	Causes data corruption Causes data conflict	Critical	No	Off-line
File	Contents error	The data read from the file does not equal we written	Data corruption	Critical	No	On-line

- Environment model

The environment model is used to simulate external failure like an unexpected power outage. It issues reset signal asynchronously to the file mapping model. With the help of the SPIN model checker, it can exhaust all the state between two asynchronous sub-models. It means that we can evaluate all the conditions of when the power outage happens. All the other sub-models enter a ready state after reset and send a ready signal to the environment model. After getting the ready signal, the environment model performs an off-line check to evaluate if the file system holds all properties, and then let the file system start again. We will introduce the off-line checking in the following sections.

C. Properties of file systems

Since a file system is used to organize and store data, keeping data correct is the most critical and essential request for the file system. The data correctness means that when a user reads data, the value must equal when it was written. Besides the data correctness, the file system also needs to keep healthy during operations. We defined four properties to indicate the file system's correctness. Three of them are for the file system's data consistency, and one of them is for the file data's correctness. If the file system does not hold one of the properties, we say there is an error. If an error causes data corruption, we consider it as a critical error. Otherwise, it is a non-critical error. TABLE I gives a summary of the four properties of a file system.

We define some notations for our abstracted file systems. The sort of *Block* is an abstract type that represents the physical blocks of the storage system. We define *File* as a resizable of *Block* where *File=ArrayOf(Block)*. We also define *FS* as a resizable of *File* to represent the abstract of file systems where *FS=ArrayOf(File)*.

- No dead block

This property is a consistent request for the block assignment in the file system. The block which is marked allocated, it must be assigned to some file. This property can be described in the following formula.

$$\forall block\ .\ the\ block\ is\ allocated \Rightarrow \exists file\ in\ FS \land block\ in\ file$$

If the file system does not hold this property, there is some dead space in the file system that cannot be reclaimed. So it results in that the available area in the file system becomes less and less. Violating this property causes some blocks of waste. However, it does not cause data loss. We can reclaim the dead blocks when we detect them. Hence a "dead block" error is a non-critical error.

To check if a block is a "dead block", we need to scan all the files and check if the block is assigned. It is difficult to check this property during file operations. So we designed an off-line checking. We run the off-line checking before we mount the file system. We will describe the off-line checking in the following segment.

- No lost block

Contrary to the dead block, the lost block means a block which is assigned to a file, but it is not allocated. It can be described in the following formula.

$$\forall file\ in\ FS\ .\ block\ in\ file \Rightarrow block\ is\ allocated$$

The file system may assign the lost block to another file again. If the second file writes data to the block, the original data is overwritten and results in data corruption. Hence, the "lost block" error is critical. We detect this error both on-line and off-line.

- No double pointed block

In the file mapping model, one block can only be assigned to one file. It can be described in the following formula.

$$\forall f_i,\ f_j\ in\ FS\ .\ f_i \neq f_j \Rightarrow B_i \cap B_j = \emptyset$$

Where B_i, B_j are defined as sets of blocks in the block array of file f_i and f_j. Some file systems support symbol links or hard links. Since these features belong to the file path mapping layer, they are out of range for this thesis. If one block is assigned to more than one file, or assigned twice in the same file, the second file's data may overwrite the first file's data in the same physical block. Hence, violating this property may result in data corruption. The "double-pointed" error is critical. We can only check this property off-line because detecting the double-pointed blocks need to scan all files. It is challenging to do on-line checking.

- The correctness of the file contents

The correctness of file contents is the necessary request for a file system. Each data read from a file should be the same as what it was written.

$$\forall file\ i\ b.\ i<sizeof(file) \Rightarrow read(write(fs,\ file,\ i,\ b),\ file,\ i)=b$$

Where the function *fs'=write(fs, f, i, b)* is a write operation, which writes a block *b* to the index *i* of the file *f* in the file system *fs,* and it returns the new state of the file system *fs'*. The function *b=read(fs, file, i)* is a read operation, which returns a block from the index *i* of the file *f* in the file system *fs*. Violating this property results in a critical error. This property is checked on-line in a designed subroutine. We designed reference files to check the correctness of the file contents. A reference file is a resizable array. For the facility of implementation, we use a fix-sized array and a length field to describe the reference file. Each valid concrete file has a corresponding reference file. The reference file records the length of which the concrete file should be. When we write data to the concrete file, we write the same value to the same location in the corresponding reference file. In the subroutine, we check if the concrete file has the same length with its corresponding reference file. Then we compare the two files' contents one byte by one. If the checking subroutine finds different content, it reports violating assertion to the model checking.

- Checking properties

In the model checking, we use the assert statements to check if the properties are held. This method allows us to check all properties in a single execution of the verification. The properties are checked on-line or off-line. The on-line checking means we check the properties when the file system is mounted. For example, when we read or write a file's data block, we verify if the block is allocated. Otherwise, a "lost block" error is detected. The "lost block" property and file contents are checked on-line.

The off-line checking is invoked in the environment model. When a simulated power outage happens, the environment model reset all other models, then invoke the off-line checking before re-mount the file system. We designed an off-line checking function to checked "lost block", "dead block", and "doubled-pointed block". In the off-line checking, we scan the blocks twice. In the first scan, we check that if all blocks in each file are allocated. If not, we detect a "lost block" error. Then we mark these blocks as checked. If a block has already been marked, we detect it as a "double-pointed" error. After this scan, all allocated blocks should be marked. Then in the second scan, we check that if all the allocated blocks are marked. If not, we detect it as a "dead block" error.

Finally, we created both two types of file system models by Promela in three files in a total of 1770 lines. The storage model, which is shared with both file system models, is separated into the file "storage.pml" with 105 lines. The link-type file mapping model, which contains a copy of the environment model and the test model, is described in the file "link.pml" in 744 lines. The index-type file mapping model is described in the file "index.pml" in 921 lines. Similar to the link-type one, it also contains a copy of the environment and the tester models. TABLE II gives a summary of Promela models.

IV. Verification

We verified both types of file system models. For each type model, we verified it in both single-thread mode and multi-thread mode. We run the verification by the SPIN using DFS (depth-first searching) algorithm. For the space-efficient consideration, we chose the BITSTATE hash to compress the state. BITSTATE HASHING is a lossy compression algorithm, but it has more space-efficient than a regular hash table.

At first, we allowed the environment model to send a reset repeatedly and did not limit the reset times. This method causes the verifier to reach the max searching depth soon. As a solution, we limited the environment model sending reset at most twice. Since the file system returns to a consistent state after it recovers from the power outage, more resets do not cause new errors. These efforts allow the model-checking to complete the searching less than $3x10^8$ depth.

TABLE II. Summarize of the file system models

File	Dependency	Contents	Scale *(lines)*
storage.pml	non	Storage model	105
link.pml	storage.pml	Link-type file mapping model, Tester model, Environment model,	744
Index.pml	storage.pml	Index type file mapping model, Tester model, Environment model,	921

A. *Model checking for single-thread mode*

First, we checked the file system models in the single-thread mode. We confirmed that the model had no error in normal execution without a power outage. After we imported an unexpected power outage, we found some counterexamples. Fig. 5 shows one of the examples of the double-pointed error in the link-type file mapping model.

In the typical case, when the tester calls the file system to create a file (*1:CreateFile(0)*), the file system searches empty blocks in the FAT (*1.2:FindCluster()*). When it finds an unused block, the file system assigns the block to the file and update its *inode*. Then the file system marks the block is allocated by setting the block's entry to a particular value of end-of-chain. Finally, the file system writes back the file's inode

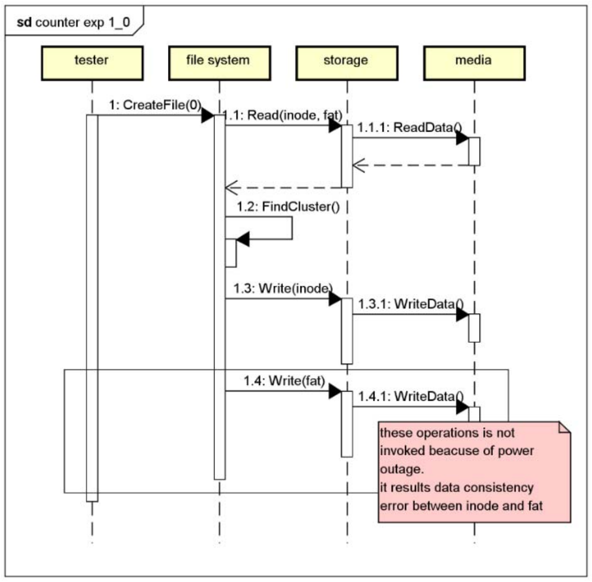

Fig. 5. Counterexample of double pointed

TABLE III. VERIFICATION RESULT WITH CACHE SOLUTION

Error mode		Error level	Link type		Index type	
			Single thread	*Multi thread*	*Single thread*	*Multi thread*
File system	Dead block	Non-critical	No error	Not Check	No error	Not Check
	Lost block	Critical	No error	Error	No error	Error
	Double pointer	Critical	No error	Error	No error	Error
File	Error content	Critical	No error	Error	No error	Error

(*1.3:Write(inode)*) and the FAT (*1.4:Write(fat)*)to the storage. The data between *inodes* and FAT should keep consistency.

In the error case, an unexpected power outage happened between writing *inode* and FAT. After the file system recovered from the power outage, the *inode* was updated, but the FAT not. It means that the block was assigned in the file viewpoint. However, the file system still considered the block as free. When the user asked the file system to create another file, the file system may assign the same block to it. So, the double-pointed error happened. If both files wrote data to the block, either data was damaged, and the data corruption happened.

Regarding the cause of the problem, we consider it is a data consistency issue. Each file operation needs to write and update some data which should keep consistency. Usually, these consistent data are not stored adjacent. So we need to update all these consistent data separated into several write commands. These sequence of write commands may be interrupted by a power outage or other external failures.

In our counterexample, create a file need to update the file's inode and the global FAT. However, file 0's inode and the FAT are separated. We invoke two write commands to update the inode and the FAT, respectively. When the simulated power outage happened between writing inode and FAT, the data consistency is broken. The brokenness of the consistency makes a conflict between the file and the file system about the assigned block. It results in the double-pointed block error.

To keep the data consistent, we consider using the volatile cache in the storage device. During the file operation, all the data written to the storage is saved in the volatile cache first. When all the consistent data are ready, the file system issues a FLUSH command to let the storage device move all cached data to the non-volatile media. Since the FLUSH command is invoked inside the storage device, we can consider it as an atomic invoking. In this case, if a power outage happened during the file operation, all update data in the volatile cache is discarded. The data in the non-volatile media can keep in a consistent state.

Following this idea, we implemented the volatile cache feature in the storage model. For convenient implementation considerations, we designed a cache which has the same capacity of the storage device. After the storage model reset (or power on), we load all data in the non-volatile media to the cache. During runtime, all the data is written to and read from the cache. If the file system issues a FLUSH command, we copy all data from the cache to the non-volatile media. If any power outage happened, the data in the cache is restored by the non-volatile data. It means all the data written after the last FLASH command is discarded. We checked both types of the file mapping models using the new storage model in the SPIN; we cannot detect any error in the single thread mode even power outage happened. TABLE III presents the checking result using this mechanism.

B. Model checking for multi-thread mode

When we evaluate the file system model in the multi-thread mode, we still found some critical errors. The above mechanism does not affect the multi-thread mode. Fig. 6 presents a counterexample of a lost block error. In this counterexample, we have two threads, and we are invoking create file operation in the thread 1 (*1:CeateFile(0)*)and invoking delete file operation in another thread (*2:DeleteFile(1)*). In the create operation, the file system loads the FAT (*1.1:Read(inode, fat)*), finds a free block (*1.2:FindCluster()*), updates the inode and FAT, writes them back to the storage (*1.3:Write(inode)* and *1.4:Write(fat)*) and issues FLUSH command (*1.5:Flush()*) to close the file. In the delete operation, the file system reclaims all the blocks used

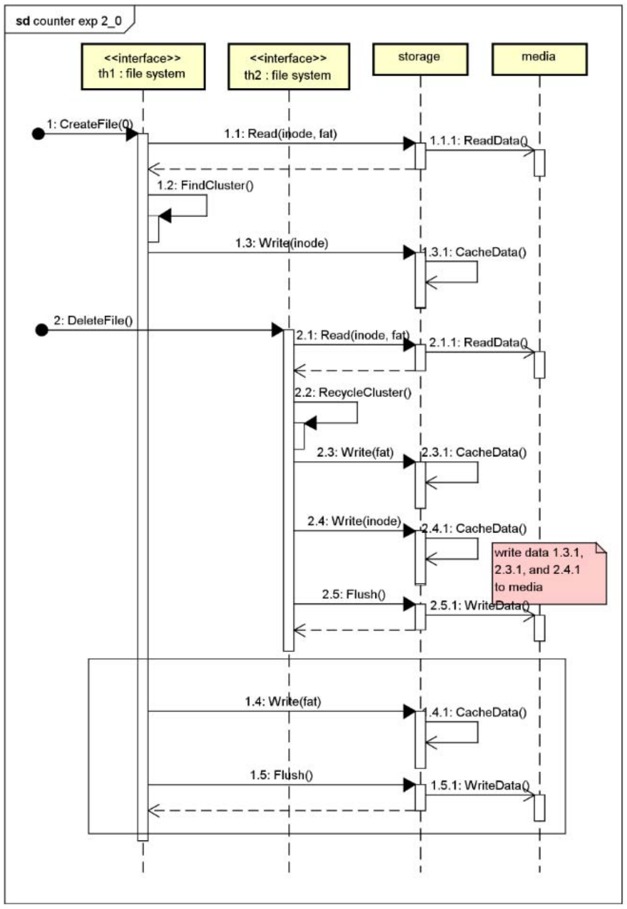

Fig. 6. Counterexample for multi thread mode

by the file (*2.2:RecycleCluster()*), updates the FAT and inode, writes them to the storage(*2.3:Write(fat)* and *2.4:Write(inode)*), and finally issues the FLUSH command (*2.5:Flush()*). In the user's viewpoint, these two operations are invoked concurrently. However, because the storage does not support concurrent operations, the file system needs to serialize the storage commands and issues them alternately to the storage device. When the *DeleteFile* operation issues the FLUSH command, it let all the cache data store to the non-volatile media, including the data updated by thread 1 (the inode of file 0 in this counterexample). Because the storage's cache does not distinguish the data from different threads. This behavior makes the thread 1's data break (the inode has stored, but the fat has not) if a power outage happens just after the FLUSH command from the thread 2 (*2.5:Flush()*). Hence, to improve the file system's robustness for multi-thread access, it is necessary to find a mechanism that does not depend on the storage cache.

According to the counterexamples, the error always happens in the case of allocating a new block. When we allocate a new block and assign it to a file, we need to update several entries in the mapping table and the block allocation table. Usually, the issue happens between storing the upper layer mapping table and low layer mapping table, or between the mapping table and the allocation table. In the link-type file mapping model, the upper layer is the start block entry in the inode; the lower layer is the block entries in the FAT. The block allocation table shares the FAT with the mapping table. In the example, when the write sequence is broken after updating the inode or before updating the FAT by an external power outage or the FLUSH command for other threads, the data consistency is broken, and then some critical errors may happen.

C. *Improvement of robustness*

We found that it is difficult to avoid all the errors, but we can avoid critical errors by adjusting the order of writing data in each file operation. In the file system, the file block mapping is a hierarchy structure. It is layered by the pointer order. In the link-type file mapping model, the start block entry in the inode points to the block-chain in the FAT and the entry in the block-chain points to the data block. The top layer is the start block entry in the inode, the second layer is a block-chain, and the lowest layer is the data block. In the index-type file mapping model, the entry of the index table in the inode points to an indirect index table, the entry in the indirect index table points to a double indirect index table, and so on until it points to a data block. The top layer is the index table in the inode, then the indirect index tables and the lowest layer is the data blocks.

According to the block and pointer dependency, we proposed a mechanism to improve the file systems' consistency during the power outage. We suggest an updating order that is writing a pointed-to block to disk before the entry that points to it when allocating a new block, and that reinitializing or reusing a pointed-to block to disk after removing the entry that points to it when reclaiming a block for reusing. Following this rule, we can ensure that the entries never point to an invalid or conflicted block. This mechanism can help the file system to avoid all critical errors; even the operations are broken by an unexpected power outage.

For the practice of our models, we modified both types of file mapping models as the following mechanism. For the link-type file mapping model, the block allocation table shares the FAT with the mapping table. In the create file operation, when we assign a new block to a new file, first, we mark a free block to allocated state in the FAT and write the FAT to the storage. Secondly, we let the start block filed in the inode point to the block, which is allocated and write the inode to the storage. In the delete file operation, when we reclaim the blocks of a file, first, we clear the start block field in the inode first and write it to the storage. Then, clear the entry from the blockchain's head to the tail in the FAT, and write the FAT to the storage.

For the index-type file mapping model, the block allocation is stored in the bitmap. In the create file operation, first, we allocate a data block by marking it as used in the bitmap. Secondly, if a new indirect index block is necessary, we allocate another block by marking it in the bitmap, then let the entry in the indirect index block point to the data block allocated, and write the indirect index block to the storage. Finally, let the entry in the inode point to the indirect index block and write the inode to the storage. In the delete file operation, first, we clear the index table in the inode and write it to the storage. Then we clear the indirect index table if it exists. Finally, we mark the related blocks as free in the bitmap and write the bitmap to the storage. In this case, it needs to take some additional memory to remember the upper layer index table temporally.

We evaluated both file system models with the above mechanism. First, we checked all of the four properties. We detected the dead block error in both file system models. Since the "dead block" error is a non-critical error, we disabled aborting from the "dead block" error in the off-line checking and verified the model again. The result did not show any errors.

Finally, we run the verifier four times to cover both types of file mapping models not only in single-thread mode but also in multi-thread mode. TABLE IV summarizes the verification result for both types of file mapping models. The result shows that we cannot detect the critical errors in our file system models. According to that, we ensure that our mechanism can improve the file systems' robustness that preventing the file systems from critical errors, even the operations are interrupted by an unexpected power outage.

TABLE IV. VERIFICATION RESULT OF THE FILE SYSTEMS

Error mode		**Error level**	**Link type**		**Index type**	
			Single thread	***Multi thread***	***Single thread***	***Multi thread***
file system	dead block	non-critical	no error	repaired	no error	repaired
	Lost block	critical	no error	no error	no error	no error
	double pointer	critical	no error	no error	no error	no error
file	error content	critical	no error	no error	no error	no error

```
01:  static FL_FILE* _create_file(const char *filename)
02:  {
       ... Sanity check and initialize ...
       ... Open the parent directory ...
03:    // Create the file space for the file (at least one clusters worth!)
04:    file->startcluster = 0;
05:    if (!fatfs_allocate_free_space(&_fs, 1, &file->startcluster, 1))
06:    {
07:      _free_file(file);
08:      return NULL;
09:    }
       ... file name processing...
10:    // Add file to disk
11:    if (!fatfs_add_file_entry(
12:      &_fs, file->parentcluster, (char*)file->filename,
13:      (char*)file->shortfilename, file->startcluster, 0, 0))
14:    {
15:      // Delete allocated space
16:      fatfs_free_cluster_chain(&_fs, file->startcluster);
17:      _free_file(file);
18:      return NULL;
19:    }
       ... set general attributes ...
20:    fatfs_fat_purge(&_fs);
21:    return file;
22:  }
```

Fig. 7 The original source code of _create_file() in the Ultra-Embedded FAT IO

D. *Evaluate the concrete file system*

Following the counterexample trace, we find the same error in the referenced source code of "Ultra-Embedded FAT IO Library[21]".

In the source code of the *_create_file()* function, we found that write order issue (See Fig. 7). In this function, the file system allocates a free cluster by calling the function *fatfs_allocate_free_space()* (line 05). In this function, it only updates the FAT table in the memory, instead write the FAT table back to the storage. Then the file system saves the file entry, which includes the start cluster into its parent directory in the function of *fatfs_add_file_entry()* (line 11). Finally, the file system writes back the FAT table to the storage in the function of *fatfs_fat_purge()* (line 20). These operations violate the rule that "write the pointed-to object before the object that points to it".

We did a power-outage simulation test of creating a file. We call the *_create_file()* function to create a test file in the root directory, we write about two sectors of data to the file and then close the file. We recorded the write command sequence of these operations according to the log of the storage simulator (See Fig. 8). We found that the file system issued four write commands to the storage. We define these commands as W_1, W_2, W_3, W_4. The 1st write (W_1, LBA 0x32, the location of the root dir) is used to save the file entry to its parent directory (the root in this test). The 2nd write (W_2, LBA 0x08, the location of the FAT) is used to update the FAT tab. Then the file system writes the user data by the 3rd write command (W_3, LBA 0x52, the location of the file). Finally, it writes the file entry again to update the file length by the 4th write command (W_4, LBA 0x32).

In order to simulate the power-outage after each writing command, we prepare a set of disk images. The original disk image is called as img_0. Then, we invoke the 1st write command W_1 to the img_0 and get a new disk image img_1. Then we invoke the W_2 on the img_1 and get img_2. And so on, we can get img_3 and img_4. These images img_1, .., img_4 present the disk status when the power-outage happens after command W_1, .., W_4, respectively. We run the *fsck* on these images to check if any errors happened after power-outage.

As a result, we detected a lost block error on img_1. It means if a power-outage happens between the 1st and the 2nd write command (we suppose that the write command is atomic and cannot be broken by the power outage), a consistent critical error happens in the file system. The consistent error happens because the file entry in the root directory is stored, but the FAT table not. The failure mode matches the counterexample in the model checking.

According to the method which has been verified in the above section, for the link-type file system, it is necessary to store the block allocation information to the file allocation table first, then update the file entry in the root dir. In this issue, we added to line 11 (See Fig. 10) to make the file system write back the FAT table before it stores the file entry.

We redo the above power-outage test on the fixed file system. Fig. 9 shows the write command log of creating a file with the

```
W1:  write lba=00000032, secs=1   // write root dir
W2:  write lba=00000008, secs=1   // write FAT
W3:  write lba=00000052, secs=2   // write file
W4:  write lba=00000032, secs=1   // write root dir
```

Fig. 8 Write command log for the wrong order.

```
W1:  write lba=00000008, secs=1   // write FAT
W2:  write lba=00000032, secs=1   // write root dir
W3:  write lba=00000052, secs=2   // write file
W4:  write lba=00000032, secs=1   // write root dir
```

Fig. 9 Write command log for the correct order

```
01: static FL_FILE* _create_file(const char *filename)
02: {
        ... Sanity check and initialize ...
        ... Open the parent directory ...
03:   // Create the file space for the file (at least one clusters worth!)
04:   file->startcluster = 0;
05:   if (!fatfs_allocate_free_space(&_fs, 1, &file->startcluster, 1))
06:   {
07:     _free_file(file);
08:     return NULL;
09:   }
10:   //<SPOR> save fat to fix spor issue
11:   fatfs_fat_purge(&_fs);
        ... file name processing...
12:   // Add file to disk
13:   if (!fatfs_add_file_entry(
14:     &_fs, file->parentcluster, (char*)file->filename,
15:     (char*)file->shortfilename, file->startcluster, 0, 0))
16:   {
17:     // Delete allocated space
18:     fatfs_free_cluster_chain(&_fs, file->startcluster);
19:     _free_file(file);
20:     return NULL;
21:   }
        ... set general attributes ...
22:   fatfs_fat_purge(&_fs);
23:   return file;
24: }
```

Fig. 8 The fixed _create_file() in the Ultra-Embedded FAT IO

fix. It shows writing the FAT table (W_1, LBA 0x08) before writing the file entry (W_2, LBA 0x32). Because the FAT table is not changed after line 11, the function *fatfs_fat_purge()* in line 22 does not write FAT again. In this case, we can only detect a dead block error in the img_1, and there is no data loss or other critical error detected. This result matches the conclusion in the above model checking.

V. Evaluation

In this work, we designed two file system models from the mainstream real file systems to comprehensively evaluate the file systems' robustness. With the help of the model checking method on these models, we detected some critical errors when the unexpected power outage happens on the file systems. And then, we provided a mechanism to improved the file systems' robustness. Finally, we verified the absence of critical errors on the improved file systems. TABLE V presents the verification time for each mode. It takes about 4.9 hours to run four verifications on our desktop PC, with Intel Core i7-8700K CPU, 16GB memory, and Windows 10 64bit.

The first advantage is that our models cover most of the mainstream file systems and exhaustive conditions like both single-thread and multi-thread mode, and multiple properties. We divided the mainstream file systems into link-type and index-type according to their file mapping algorithm and designed two models to cover these file systems. Comparing the two kinds of file systems, TABLE III and TABLE IV show that the link-type and the index-type file systems have the same robustness. Regarding the implementation of these two models, TABLE II shows that it costs 744 lines to implement the link-type model, while to implement the index type in the same scale, it needs 921 lines. It means that implementing the link type file system is more comfortable than implementing the index type one. However, the index-type file system is more efficient in the multi-thread because each file manages the file mapping data in local. These behaviors result that more mobile applications like USB Memory use the link-type file system, and more desktop applications like Windows or Linux use the index-type file system.

Comparing the verification result, TABLE V shows that the single-thread model has a smaller state space and more comfortable to check than the multi-thread mode. The single-thread model keeps data consistent easier than the multi-thread model when the unexpected power outage happens. The volatile cache improves the single-thread model's robustness. However, it has no help for the multi-thread model. The method of appropriate write-command order improves the robustness for both single-thread and multi-thread.

The second advantage of our model is that we can find corner-case errors in the file systems within an acceptable duration on the desktop PC. It is difficult to find the corner-case errors caused by power outage without checking the whole file systems' states. We abstract the file mapping model from the file system and scale down the model size. This method helps to avoid the state explosion when we exhaust the model. In the file system model, we also design the on-line checker and off-line checker, which allow us to check all the file system properties in one execution. The checkers helps to reduce the total execution time for the comprehensive model checking.

The third advantage is that the model helps us to find a mechanism that improves the file systems' robustness. We recall the corner-case errors in the real file system, and fixed these errors and improved the file systems' robustness by analyzing the trace of the counterexamples. Finally, we verify that the improved file systems hold all critical properties, even the unexpected power outage happens. Moreover, the mechanism of the improvement covers most of the mainstream file systems,

TABLE V. SUMMARY OF THE VERIFICATION RESULT

Model		Depth	States	Transitions	Mem (GB)	Time (hrs)
Link type	single-th	763408	0.75×10^{10}	1.71×10^{10}	4.67	0.82
	multi-th	2652768	1.20×10^{10}	3.83×10^{10}	4.73	1.81
Index type	single-th	3449	0.56×10^{10}	1.15×10^{10}	4.64	0.60
	multi-th	37686	1.10×10^{10}	2.86×10^{10}	4.65	1.66

and it does not depend on external tools nor additional journal operations.

However, our work is subjected to the following threats to validity. 1) We defined four properties of file systems' correctness by our experiment. In our experiment, we found that if these four properties are held, the file system should be no error on data integrity. These properties may not be sufficient for evaluating the file system. To minimize the risk of missing properties, we designed a reference file system, which can be used to check the files' data integrity. 2) There may be some mistakes during the abstracting model from the real file system. To minimize the risk, we performed a special designed test on the real file systems to reproduce the counterexample in the model checking. Then we verified the issues were fixed in the real file systems by the solutions.

There is also a limitation to this work. Our model does not present the directory part of the file systems, so it cannot detect the errors in the directory data structures. Different file systems implements manage the directory data structures in various algorithms. Usually, the directory has a tree structure. To check the directory mapping model requests too massive resources. We will try to check the file system, including the directory mapping model in the future works.

VI. SUMMARY

In this work, we comprehensively evaluated the file systems' robustness in the presence of the unexpected power outage. We designed models from both link type and index type file systems. We also defined the properties of the file systems' correctness and verified the models by model checking. As a result, we found a critical error that can be encountered in a real file system. In the verification, we obtained a counterexample, which makes it possible to find a root cause of the error and how it should be fixed. According to this analysis of the error, we proposed a mechanism to improve the file systems' robustness.

We adopted the two-layer mapping in modeling file systems, which allows us to abstract concrete systems and avoid the state explosion in the model checking. We think that it could be useful for designing a new file system such as a NAND FLASH file system. We also found a mechanism to update a dynamic mapping system so that it can prevent data inconsistency issues. This mechanism could be useful for the other mapping based storage systems such as the address mapping of SSD.

We would like to continue the investigation of the file systems' robustness, including the file path mapping, which is out of scope at this moment. Taking it into account may cause the state explosion, but proposing a method to mitigate it is our future work.

REFERENCES

[1] G. J. Holzmann, *The SPIN Model Checker Primer and Reference Manual*, Addison Wesley, 2004.

[2] C. Baier and J. Katoen. *Principles of Model Checking*, The MIT Press, 2008.

[3] K. Arkoudas, K. Zee, V. Kuncak, and M. C. Rinard. "On Verifying a File System Implementation", in *International Conference on Formal Engineering Methods*, vol. 3308 of LNCS, p. 373-390, 2004.

[4] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. "Using model checking to find serious file system errors", in *USENIX Symposium of Operation System Design and Implementation*, p. 273-288, 2004.

[5] A. Galloway, G. Luttgen, J. T. Muhlberg, and R. Siminiceanu, "Model-Checking the Linux Virtual File System", in *Verification, Model Checking and Abastract Interpretation*, vol. 5403 of LNCS, p. 74-88, 2009.

[6] A. Galloway, J. T. Muhlberg, R. Siminiceanu, and G. Lutgen. "Model-checking part of a Linux file system", Tech. Report. YCS-2007-423, U. of York, UK, 2007

[7] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engle. "Automatically Generating Malicious Disks using Symbolic Execution", in *IEEE Symposiumo on Security and Privacy*, p. 243-257, 2006.

[8] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu. "Towards Robust File System Checkers", in *USENIX Conference of File and Sotrage Technologies*, p. 105–121, 2018.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*, O'Reilly, 2002.

[10] E. M. Clarke, T. A. Henzinger, H. Veith and R. Bloem, *Handbook of Model Checking*, Springer International Publishing AG, 2018.

[11] C. Cadar, P. Twohey, V. Ganesh, and D. Engler, "EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution", in *Conference on Computer and Communications Security,* p. 322-335, Oct. 2006.

[12] M. Kim, Y. Choi, Y. Kim, and H. Kim, "Formal Verification of a Flash Memory Device Driver – an Experience Report", in *Spin Workshop*, LNCS 5156, p. 144-159, LA, USA, 2008.

[13] M. Kim and Y. Kim, "Concolic Testing of the Mult-sector Read Operation for Flash Memory File System", in *Brazilian Symposium on Formal Methods*, LNCS 5902, p. 251-265, Gramado, Brazil, 2009

[14] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak and X. Wang, "Specifying and Checking File System Crash-Consistency Models", in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 83-98, NY, USA, 2016

[15] "The Ext2 file system," *sourceforge.net*, [Online]. Available: http://e2fsprogs.sourceforge.net/ext2.html

[16] "File Allocation Table," *wikipedia.org*, [Online]. Available: https://en.wikipedia.org/wiki/File_Allocation_Table

[17] "File System," *wikipedia.org*, [Online]. Available: https://en.wikipedia.org/wiki/File_system

[18] "SPIN model checker," *wikipedia.org*, [Online]. Available: https://en.wikipedia.org/wiki/SPIN_model_checker

[19] "Database transaction," *wikipedia.org*, [Online]. Available: https://en.wikipedia.org/wiki/Database_transaction#Transactional_filesystems

[20] "Commit (data management)," *wikipedia.org*, [Online]. Available: https://en.wikipedia.org/wiki/Commit_(data_management)

[21] "Ultra-Embedded FAT16/32 File IO Library," *ultra-embedded.com*, [Online]. Available: http://ultra-embedded.com/fat_filelib/

[22] "Dokan," *dokan-dev.github.io*, [Online]. Available: https://dokan-dev.github.io

[23] J. Corbet. "ext4 and data loss" *lwn.net*, [Online]. Available: http://lwn.net/Articles/322823