Doctor's Thesis

Verification of Flash File System Crash Consistency Based on Model Checking

Jingcheg Yuan

Supervisor Toshiaki Aoki

Division of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

March, 2026

Abstract

File systems are fundamental to computing, managing data on storage devices, yet they face the persistent challenge of maintaining crash consistency, ensuring data integrity after an unexpected system failure. Operations like file creation or modification require updating multiple, non-adjacent on-disk data structures, and a crash during this process can leave the file system in an inconsistent state, leading to data corruption or loss. While techniques such as journaling improve robustness, verifying crash consistency remains difficult due to system complexity and the vast number of potential failure scenarios. Existing testing methods, including black-box tools like CrashMonkey, suffer from limited coverage and efficiency, often missing subtle corner-case errors. Formal verification with model checking offers exhaustive exploration but has been hampered by state-space explosion and the impracticality of modeling full-stack file systems in languages such as Promela. This thesis introduces a comprehensive methodology and a practical tool, MC^3 (Model Checking for Crash Consistency), for the automated, exhaustive verification of file system crash consistency. Our approach employs a state space search that exhaustively explores file system states while checking correctness properties on-the-fly. To overcome state explosion, we developed two key innovations: an object-based workload generation algorithm that systematically produces all valid file system operations, and a novel directory tree isomorphism detection technique that encodes and eliminates redundant states, drastically reducing the search space. We first applied this methodology to block-based file systems (e.g., FAT, ext2), identified the root causes of consistency violations and the non-atomicity of multi-block metadata updates, and proposed a write-ordering mechanism to prevent critical errors. We then implemented MC^3 to verify file system models written in C/C++. Applying MC^3 to a model of the Flash-Friendly File System (F2FS), a complex log-structured file system (LFS), revealed a previously unknown crash consistency bug where metadata rollback combined with block reuse during garbage collection causes silent file data loss, a vulnerability likely common to many LFS designs. Our evaluation shows that MC^3 significantly outperforms CrashMonkey in testing efficiency and coverage, enabling the discovery of deep design-level bugs on a standard desktop PC within a practical time frame, thereby providing a powerful and scalable verification framework to enhance the reliability of future storage systems.

Keywords: Model Checking, File System, Crash Consistency, SPIN, F2FS

Contents

List of Figures

List of Tables

Chapter 1

Introduction

File systems rely on a combination of data structures, allocation mechanisms, metadata, and user data to manage files on storage devices. For operations such as file creation, updates to several on-disk data structures are mandatory [7]. For instance, in the FAT file system, creating an empty file entails modifying the parent directory's entries and the file allocation tables (FAT) associated with both the parent directory and the new file. Notably, these data structures are rarely stored contiguously on disk and lack atomicity during updates. Should a file operation be interrupted (e.g., by a system failure), partial persistence of these updates, where some changes are retained and others discarded, can lead to file system inconsistencies.

File systems are ubiquitous for organizing data on storage media by partitioning it into discrete units called files. This file-based partitioning simplifies data isolation and identification, with most file systems adopting a tree-like hierarchy in which files serve as leaf nodes and directories serve as non-leaf nodes that organize and link files. Typically implemented as a kernel module, a file system also manages the operating system's own system files. A critical file system error can result in catastrophic data loss and even render the operating system inoperable. Numerous incidents of file system-induced data loss have been documented, such as the 2009 ext4 data loss event, where multiple users reported that "pretty much any file written to by any application" would be emptied following a system crash [8, 9]. Sudden power outages or system crashes can cause a file system to enter a transient, inconsistent state. Upon reboot, the file system initiates automatic repair procedures to resolve these inconsistencies. However, if the repair fails, a crash consistency flaw may emerge, leading to data corruption, file loss, or an unbootable system.

Designing and verifying a robust, crash-consistent file system is a non-trivial task. First, file systems are large-scale and complex, with myriad operational combinations that are difficult to simulate exhaustively in testing.

Second, a file system crash can occur at any time under various conditions, making it challenging to account for all possible scenarios during design and to simulate all conditions during testing. Third, the cost of file system crash testing is substantial. Traditional power-off testing requires a cold boot and a subsequent file system repair, and each test cycle often takes around 5 minutes. Furthermore, if a crash-consistency issue occurs, it can corrupt the tested system, rendering it unbootable and requiring reinstallation, severely impacting testing efficiency. While kernel testing on a virtual machine can avoid permanent system damage, it reduces execution speed compared to a physical machine. Many file system developers use xfstester [10] for testing, but its crash-consistency test suite accounts for only 5% of its total tests. This highlights that file system developers lack effective methods for detecting and verifying crash consistency.

CrashMonkey proposed a bounded black-box testing (B3) method for testing file system crash consistency. This method automatically generates test workloads and performs black-box testing on an actual system. However, CrashMonkey is inherently designed for testing with extremely short workloads, which imposes strict limitations on both the number of files and their sizes. Consequently, such workloads fail to effectively verify the file system's crash consistency behavior under boundary conditions. Furthermore, CrashMonkey exhibits low test efficiency and requires substantial computational resources, making it practically infeasible to increase test depth and coverage further.

Model checking, a rigorous formal verification method, distinguishes itself from black-box testing and other testing paradigms by its ability to uncover nuanced flaws. It achieves this by comprehensively enumerating all feasible system states and systematically assessing every potential crash scenario. These capabilities enable the detection of even insidious errors often missed by alternative approaches. However, applying model checking to file system testing faces numerous challenges: (1) How to create a model to cover the diversity of file system types? (2) How to traverse the whole file system state space? (3) How to comprehensively simulate crash conditions? (4) How to define and check the file system's correctness? Furthermore, a file system has a large state space, which makes it difficult for model checking to cover the entire state space.

To address these gaps, this study proposes a comprehensive and efficient model checking methodology for verifying file system crash consistency through exhaustive state-space exploration and property checking. We proposed an on-the-fly workload generation approach that enables traversal of the entire file system state space. We also defined file system correctness properties that can be verified by searching the file system state space.

Our study comprises two phases. First, we demonstrated the feasibility of verifying file system crash consistency via model checking with the SPIN tool [1, 11]. We constructed models of the core components, a file-mapping model of two distinct file systems in Promela. Based on model checking results, we systematically verified that crash consistency violations arise from the non-atomicity of metadata updates. Furthermore, we demonstrated that critical errors can be prevented by enforcing specific ordering constraints on metadata write operations.

Subsequently, we proposed a comprehensive approach to file system verification that addresses the scalability limitations inherent in SPIN and Promela modeling. We designed a C/C++-native model interface to align with real-world file system implementations; a file system state-encoding algorithm to track visited states; a multi-threaded DFS algorithm to improve search efficiency; and an I/O rollback approach to simulate crash scenarios exhaustively. As well, we designed the MC^3 (Model Checking on Crash Consistency) tool to implement our proposal, and used the MC^3 tool to detect potential crash consistency issues in LFS.

In summary, the primary contributions of this thesis are:

1. A comprehensive evaluation approach of the file system by SPIN. This portion was published in Yuan et al. [1]. The key design includes:
    - File mapping models in Promela to cover mainstream block file systems.
    - On-the-fly workload generation algorithms to traverse the file system stats space exhaustively.
    - Verifiable to define the file system correctness.
2. The full-stack file system checking approach and the implementation of MC^3 to make checking full-stack and log-structured file systems possible. The key design includes.
    - A C/C++ native model with a POSIX-compatible interface to align with concrete file systems.
    - A file system state encoding scheme to track visited file system states
    - Multi-thread DFS algorithm to improve the searching speed.
    - IO rollback approach to exhaustively simulate crash scenarios.
3. Root Cause Analysis and Solution for Block-based File Systems: We identified the root cause of crash consistency violations in block-based

file systems (non-atomicity of metadata updates) and proposed and verified a low-cost solution based on write-ordering.

4. Discovery of Latent LFS Errors: We discovered a latent crash consistency bug in Log-structured File Systems (LFS), specifically affecting metadata rollback and garbage collection.

However, our model still has certain limitations. Overall, the two-phase verification approach is targeted at file system models. Abstraction from real-world file systems to file system models, whether Promela-based or C/C++-based, inevitably introduces discrepancies. For instance, implementation-specific information or code may be omitted during abstraction. This represents a trade-off between coverage and granularity in file system checking: if bugs exist in the omitted implementation code, they cannot be detected by model checking.

Additionally, regarding the SPIN-based model checking approach in the first phase, although we explored the entire state space of the file-mapping model, this model is extremely limited in scale. It fails to incorporate common file system functionalities, such as managing directory trees and garbage collection (GC). In the second phase, we adopted a C/C++ file system model that enables our model to scale to full-stack file systems, including GC functionality, and aligns it more closely with real-world file system implementations. However, the increased scale of the full-stack model precludes exhaustive exploration of the entire file system state space; instead, we can only explore it as comprehensively as possible within a given search depth. On another front, the current version of the MC^3 tool cannot directly verify multi-threaded file system tasks. It can only simulate such scenarios by opening multiple files simultaneously. We aim to address these limitations in future research.

Our paper is organized as follows: Chapter 2 reviews related work on file system crash consistency. Chapter 3 provides background knowledge on file systems, crash consistency, and model checking. Chapter 4 details the file system model checking by the SPIN tool. Chapter 5 details the full-stack file system checking approach and the MC^3 tool. Chapter 6 provides a conclusion.

Chapter 2

Related Works

Model Checking on Crash Consistency (MC^3) is a tool for comprehensive crash testing of file system models. It can automatically perform thorough, full-featured checks on file systems, including crash consistency verification under various conditions. Here, we introduce the development of file system verification techniques that preceded MC^3.

2.1 File Systems Crash Consistency

As file systems have evolved, they have integrated an expanding array of features, thereby increasing complexity. This progression has heightened the focus on robustness, particularly in scenarios involving system crashes. The earliest strategy for addressing crash-related inconsistencies was passive post-failure checking and repair, exemplified by legacy file systems like Windows' FAT and Linux's ext2. File system check utilities, such as fsck [12], execute during system reboot to identify and resolve inconsistencies [13, 14]. Despite its straightforward implementation, this method requires a full disk scan before it can resolve any issues. While various optimizations have been proposed to shorten the execution time of these checkers [15, 16, 17], the overhead remains prohibitive for large-scale storage systems.

Concurrently, file systems have bolstered crash resilience through advancements in their underlying data structures and algorithms. A pivotal breakthrough in this regard was the adoption of journaling mechanisms. Journaling file systems [18] first record a log of metadata, and in some cases, user data, to a dedicated storage region; once all log entries are securely persisted, the corresponding changes are applied to their original on-disk locations. This approach is integrated into major file systems like NTFS and ext3. Nevertheless, logging metadata and user data introduces substantial

overhead for file operations, degrading write performance and increasing the volume of auxiliary data written to storage. Additionally, NAND Flash, the primary storage medium in solid-state drives (SSDs), has finite endurance, meaning the extra write operations from journaling shorten the SSD's operational lifespan [19, 20].

2.2 Checking File System

2.2.1 Black Box Test

Although file systems have incorporated methods such as journaling and fsck to improve robustness, testing and verification remain essential for detecting defects. A robust file system is not only difficult to design but also difficult to verify and test.

Black-box testing is a common method in software testing. Linux provides a file system black-box testing tool called xfstests [10]. It was initially used to test the XFS file system and was later ported to Linux alongside XFS. It now supports testing most file systems on Linux. xfstests predefines several test workloads and tests a file system by executing them. xfstests can simulate crash testing by forcing PC restarts. xfstests relies on pre-designed test workloads, typically based on experience, derived from actual usage loads, or reproductions of known issues. Such a design has certain biases and makes it difficult to cover all scenarios. It is particularly challenging to discover unknown corner case issues.

CrashMonkey addressed this problem to some extent. Mohan et al. proposed the Bounded Black-box Crash Testing (B3) [21] to check file system crash consistency. They also developed two tools, CrashMonkey and ACE [22], to implement the B3 testing method. ACE automatically generates comprehensive test workloads under given boundary conditions. CrashMonkey runs these test workloads, simulates crash conditions, and checks whether the post-crash image can be restored to a normal state. Although CrashMonkey improved the coverage of test workloads, it still cannot cover all test conditions.

The advantage of black-box testing is that it doesn't require modifying the system under test and can detect errors in real systems, including coding issues. However, this behavior also becomes a shortcoming. Actual file systems typically run in the operating system kernel. Performing crash testing on actual file systems not only consumes significant time due to frequent crashes and system restarts, but also risks the system being unable to recover after crashing [23]. One solution is to run the testing on virtual machines;

however, under the same hardware conditions, virtual machines typically run much slower than actual systems. Another problem is that black-box tests cannot guarantee that all workloads are exhausted. Although ACE attempts to enumerate all possible workloads, it is practically impossible to exhaust all paths due to limitations on the length of workloads.

2.2.2 Theorem Proving

Formal verification analyzes the source code of a software system to prove the correctness of a specific property through logical reasoning, theorem proving, or model checking. This method effectively avoids situations in which insufficient workloads result in a lack of counterexamples to errors. One method of formal verification is theorem proving, which uses higher-order logic to model and specify the properties a program must satisfy. Then, using machine-assisted proof methods, it demonstrates, step by step, that the program satisfies the required properties.

Arkoudas et al. leveraged theorem proving to verify the correctness of a file system implementation built on standard data structures and fixed-size storage [24]. Using the Athena theorem prover, they adopted a constructive approach to formal verification. Separately, Joshi et al. [25] proposed a verifiable file system amenable to automated proof. While both studies focus on defining and validating the correctness of basic file system functionalities, their notion of "correctness" does not encompass crash consistency, a critical property for real-world deployment.

In a subsequent study, Chen et al. [26] developed the specification of a verifiable journaling file system and used Crash Hoare Logic (CHL) to formally prove its correctness, specifically ensuring that the file system can recover properly after a crash. They later implemented this specification in the FSCQ file system. Originating from Hoare logic [27], a formal system with rigorous logical rules for verifying program correctness, CHL extends the original framework to account for crash scenarios, logical address spaces, and recovery execution semantics, thereby enabling the analysis of file system crash events and recovery workflows. Nevertheless, the underlying Hoare logic framework is not effective for evaluating concurrent programs.

Another relevant work by Ernst et al. [28] centered on defining a specification for flash-based file systems. To validate this specification, they employed Abstract State Machines (ASMs), which were extended to handle crash-related scenarios.

Bornholt et al. [8] proposed a crash consistency model for file systems, drawing an analogy to memory consistency models. They introduced a formal specification, a set of test cases known as Litmus Tests, and a development

toolkit called Ferrite.

Using theorem-proving methods with interactive proof assistants, such as Coq and Isabelle, to verify file-system correctness presents its own set of challenges. The main drawback is the low level of automation. Building a proof requires substantial specialized knowledge and a significant time investment. Additionally, the scale of theorem proving remains small. Although the theorem-proving approach is used for file system verification, it can only cover partial aspects of the file system, such as fixed-size storage and basic read/write operations, and it lacks support for concurrency.

2.2.3 Model Checking

Model checking is another method of formal verification, offering a higher degree of automation compared to theorem proving. Instead of converting a program's code into a theorem that must be proven, model checking requires abstracting the system's model, which demands less specialized knowledge. A key advantage of model checking is that when it finds a bug, it provides a counterexample, which makes it much easier to analyze and fix the problem. This makes model checking a more practical and suitable approach for complex systems like file systems. Modeling an existing system is far more feasible and operational than converting it into a theorem to be proven.

Compared to black-box testing, model checking provides broader coverage. Model checking exhausts the entire state space and guarantees the absence of errors in the software system under test. Several well-known model checking tools have been used to identify design flaws in software systems, as demonstrated in [29, 30, 31].

To verify file systems, Galloway et al. [32] successfully applied model checking to verify a Linux file system. They verified certain safety and liveness properties of file system application program interfaces (APIs) in the multi-threaded case. This work does not check data consistency under crash conditions. Yang et al. [33] employed symbolic execution to generate a pathological test case. They subsequently checked if the file system could cover the pathological data. The checking tool must be run on a real file system and requires significant modifications to its source code to expose choice points. This not only reduces the checker's efficiency but also demands a substantial amount of specialized professional effort.

In this research, we first used SPIN to systematically and comprehensively evaluate the robustness of the file mapping model, a crucial component of file systems, and to investigate the consistency of various file systems under crash conditions[1, 11]. This work revealed several pervasive crash consistency bugs and proposed solutions. However, a key limitation of using Promela for file

system modeling is that we can only construct a simplified model of the file system's core components. Consequently, the model check can only be performed on a very small-scale file system.

Subsequently, we designed MC^3, a file system verification tool capable of verifying file system models written in C (or any language that adheres to a specific interface). MC^3 operates in user space and employs an object-based workload generator to explore the entire state space of a file system exhaustively. It then checks all states, including the crash states that may be generated during state transitions. Furthermore, by encoding the file system's state, MC^3 can exclude symmetric states, thereby optimizing the search. Compared to tools that run on real file systems, such as CrashMonkey and ACE [22], our MC^3 demonstrates superior testing efficiency, enabling it to perform more operations per unit time. Our tool also detects corner-case errors that CrashMonkey failed to discover.

A notable recent advancement in file system model checking is PerSeVerE [34], which formalizes ext4's semantics and verifies both consistency and persistency of file I/O programs by integrating ext4's behaviors with C/C++ weak memory models. Its key innovations include: an axiomatic model that captures ext4's non-POSIX behaviors and distinguishes consistency from persistence; an extended DPOR algorithm with a "recovery observer" that reduces the state space to 2^N by focusing on observable disk locations; and support for ext4's journaling modes and configurable block/-sector sizes. Built on GenMC, PerSeVerE detected crash consistency bugs in `vim`, `emacs`, and `nano` and handles concurrent file I/O, addressing a limitation of earlier theorem-proving approaches. However, PerSeVerE's focus on ext4-specific semantics limits its direct applicability to other file systems without significant reconfiguration, and it relies on explicit modeling of file system operations rather than direct reuse of kernel code, distinguishing it from MC^3's interface-compatible model design.

Chapter 3

Preliminaries

3.1 Storage Device

File systems store data on nonvolatile storage devices, such as hard disk drives (HDDs)[35], solid-state drives (SSDs)[36], and USB memory. In mainstream computer systems, these devices typically function as block devices, and most file systems are designed accordingly.

Traditional storage devices primarily use magnetic storage media, such as HDDs and floppy disks. Because magnetic media support direct in-place updates, these devices are designed as block devices. A block storage device functions as an array of fixed-size blocks that can be updated in place, with each block serving as the minimum access unit. The host can read from or write to the storage device using block addresses. In the AT Attachment (ATA) specification [37], these blocks are referred to as sectors, every 512 bytes in size, and their addresses are denoted as logical block addresses (LBAs).

In recent years, SSDs have increasingly adopted nonvolatile semiconductor storage media, particularly NAND flash memory. Compared to magnetic storage devices, NAND flash memory offers several advantages: lightweight construction, compact size, low power consumption, and fast read/write performance. Without moving parts, semiconductor storage provides higher reliability than magnetic storage. As a result, NAND flash-based storage devices are widely used in mobile devices, including digital cameras, smartphones, and laptops. With declining per-gigabyte costs, NAND flash-based SSDs are increasingly replacing HDDs as the primary storage solution in personal computers.

NAND flash memory and SSDs differ from traditional magnetic storage media and HDDs in their asymmetric read and write characteristics and

their inability to be updated in place. Data must first be erased before writing, and it must be written sequentially to an empty block. Attempting to overwrite a non-empty page risks data corruption. The read, write, and erase operations also use different units: a page (16 KB or 32 KB) is the smallest unit for reading and writing, whereas a block (several or tens of megabytes), composed of multiple pages, is the smallest erasable unit.

To address these limitations, SSDs employ a flash translation layer (FTL)[38]. The FTL maps logical block addresses (LBAs) from the host to physical addresses in the NAND flash. It maintains a dynamic mapping table on the flash memory, updating it with each write operation. For reads, the FTL looks up the physical address, retrieves the data, and returns it to the host. For writes, it checks for existing data at the logical address, reads and merges it if present, allocates a new physical page, writes the new or merged data, and updates the mapping to link the logical address to the new physical location. As out-of-place updates consume empty blocks and reduce free space, the FTL addresses this by performing garbage collection (GC), which collects valid pages and writes them to a new block. Once rewritten, the original pages are marked invalid, and blocks containing only invalid pages are erased for reuse.

Benefiting from advances in error-detection and error-correction technologies, most storage devices employ Error Correction Code (ECC) to protect block data. The ECC module can correct bit errors within its correction threshold and transmit the corrected data to the host. When the number of bit errors exceeds this threshold, the storage device detects the anomaly and reports the fault to the host. When the host reads block data from the storage device, it will receive either fully correct data or a fault notification; partially correct data is never returned, even if read or write operations are interrupted by external failures. For this reason, the read and write operations on block storage devices are treated as atomic operations in the present study.

3.2 File System

A file system manages the organization and retrieval of data on storage devices. It arranges files and directories in a tree structure, where directories serve as non-leaf nodes and files as leaf nodes. Since a file has a dynamic size and its data is typically scattered across non-adjacent storage blocks, the file system must efficiently manage data storage and retrieval. While various types of file systems exist, including network file systems, this study focuses on disk file systems, those that store data on local devices, such as HDDs

or SSDs. A disk file system operates through three essential components: an on-disk data structure, a file system access interface, and an algorithm for managing directory and file data. The following sections detail these components and their operations.

3.2.1 Data Structure of File Systems

In most file systems, data access is divided into fixed-size blocks of 512 bytes or 4KB. A file system typically contains several key components: a superblock that stores essential system information, a block allocation table that tracks which storage blocks are in use, a file mapping that links blocks to specific files, and data blocks that store the contents of directories or files. The block allocation and mapping information is referred to as metadata, distinct from the actual user data.

The `superblock` stores critical metadata about the file system, including the physical device on which it resides, its total capacity, the mount point, and a pointer to the file system's root directory. Typically, the superblock is placed in the first physical data block of the storage device. To enhance data reliability, some file systems replicate the superblock across multiple locations for redundancy.

3.2.2 Operations of File Systems

A file system provides external interfaces for user and application interaction. POSIX defines standard operations such as create, open, close, read, and write. Here, we introduce the major operations, which we will later abstract in our models.

To use a file system, users must first initialize it on a storage device with `mkfs` and then mount it before use. When finished or before shutting down, users must unmount the file system, which flushes all unsaved metadata and user data to storage. A shutdown without unmounting the file system, called a crash, may damage data consistency and cause the file system to crash. In this study, we use model checking to assess whether a file system remains robust upon a crash. Modern operating systems automatically mount storage devices when they are attached and unmount them when they are detached or when the system is shutting down.

To store data, users create files in the file system. The creation process initializes the inode of a file and links it to its parent directory. To access existing files, users must first open them. After creating or opening a file, the system returns a handle that users can use to read or write data. While read operations don't modify storage data, write operations can either overwrite

existing data or append new data to the file. Writing requires the system to assign physical blocks and update both user data and metadata. Users must close files when finished to flush unsaved data and release resources. When a file is no longer needed, users can remove it to free its space.

For better file management, users can create directories using the createdir or mkdir operations. These directories can contain both files and subdirectories. Empty directories can be deleted using rmdir. The move function allows users to relocate files or directories between directories or rename them.

3.2.3 Types of File System

A file that functions as a resizable byte array is distributed across multiple discrete blocks on a storage device. File systems employ algorithms to map a file's logical address to the storage device's physical block address (LBA), with each file system implementing this mapping differently. Mainstream file systems are divided into two categories based on their data management approach: block-based and log-structured systems. Block-based file systems, designed for block storage devices, store most metadata and file data in fixed physical locations after allocation. When updating data, these systems modify it in the same physical location, a process known as an in-place update. In contrast, log-structured file systems [18] treat physical blocks as a sequential log, allowing data to be appended only at the end. When updating data or metadata, these systems append new data to the end of the log and update the links of the parent blocks, ensuring that all data writes occur sequentially at the head of the log.

Based on the file mapping algorithm, file systems can be further divided into link and index types. Link-type systems use a global link table to manage physical block usage and the mapping between logical and physical blocks. In contrast, index-type systems employ an array for each file's logical-to-physical mapping and a bitmap to manage block allocation. Since the link table must be stored across multiple contiguous blocks, it is not suitable for out-of-place updates; therefore, most log-structured file systems map files using the index type.

Link Type File System

The link-type file system is used by the FAT series of file systems, including FAT16, FAT32, and FATex. Files are distributed across multiple non-adjacent physical blocks on storage devices. Treating each file as a blockchain (Figure 3.1), the link-type file system uses a global link table to manage the

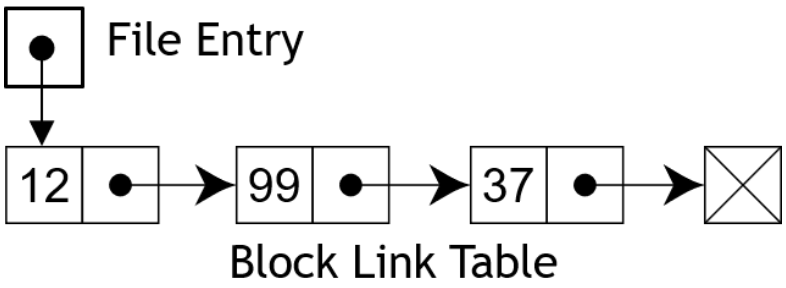


Figure 3.1: Link Type File Mapping[1]

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	file name (8.3)											Attr	Reserved			
0x10	CrtDateTime				FstClusHi		ModDateTime				FstClusLo		FileSize			

Figure 3.2: dentry structure in the FAT file system[1]

logical-to-physical address mapping for all files. This table contains an entry for each block, where each entry either records the number of the next block in the file or carries a marker indicating the end of the file, unused space, or special reserved areas. The address of the first block of a file is stored in its directory entry (dentry), enabling the file system to traverse from the first logical block to the last by querying the file allocation table. This table also manages block usage within the file system.

In FAT file systems, the global link table, also known as the File Allocation Table (FAT), is statically allocated during formatting. Acting as a linked list composed of block entries, the table either stores the number of the subsequent block or marks file endings and unused space. Moreover, it serves as a block allocation table, utilizing specific values to denote unassigned blocks.

FAT file systems use the file allocation table for block mapping and don't employ an inode structure. Instead, file information is stored in the dentry structure of the parent directory. Each file or directory has exactly one dentry that points to it; therefore, FAT systems cannot support hard links. Figure 3.2 shows the FAT32 dentry structure: bytes 00h-0Ah store the filename and extension in 8.3 format; byte 0Bh contains attribute flags; bytes 10h-13h store creation time; bytes 16h-19h contain modification time; bytes 14h, 15h, 1Ah, and 1Bh hold the first block address; and bytes 1Ch-1Fh store the file size. Using the first address in the dentry, we can locate the blockchain's head in the FAT table and access all blocks by traversing the list.

When creating a new file in the FAT file system, the system first locates an

empty directory entry slot or allocates a new one in the parent directory file, then populates the file's directory entry with the file name, attributes, and other relevant information. Next, the file system identifies an empty block from the File Allocation Table (FAT), marks it as a tail block in the FAT, and stores its address in the first cluster field of the file's directory entry. When data is added to the file, the file system locates additional empty blocks from the FAT, inserts them into the block chain, and updates the FAT accordingly. When reading data from a file, the file system retrieves the first block address from the file's directory entry. By calculating the desired offset within the file, the system determines which logical block to read. It then traverses the blockchain from the first block to the end, identifies the physical block address corresponding to the offset, and reads the data from the storage device.

While the link-type file system is simple to implement and popular in embedded systems and mobile devices, such as digital cameras and USB drives, it has limitations. Random access performance is poor since accessing specific data requires traversing the file blocks from the beginning. Additionally, parallel file access is inefficient because all files share a single global link table.

Index Type File System

The index-type file system treats a file as an array of blocks. Each file maintains an array of entries corresponding to its logical blocks. Typically, this array, referred to as the index table, is stored in the file's inode. In most practical file systems, the index table is hierarchically organized to optimize space utilization, with a variable number of layers. The initial entries of this table directly point to data blocks, while intermediate entries point to indirect index blocks that contain pointers to data blocks. The remaining entries point to multi-layered index tables. Figure 3.3 illustrates the hierarchical index block structure.

Index-type file systems utilize index tables, significantly improving the efficiency of random file access. Additionally, each file maintains its own index table, thereby increasing parallelism in file processing. These two factors lead to a substantial performance improvement in index-type file systems relative to link-type file systems. Consequently, the index-type has been widely adopted by mainstream file systems, such as the ext series in Linux and NTFS in Windows.

In the index-type file system, a directory is treated as a special file, with its data managed in the same manner as a normal file. A directory file contains a set of directory entries (dentry), where each dentry represents a

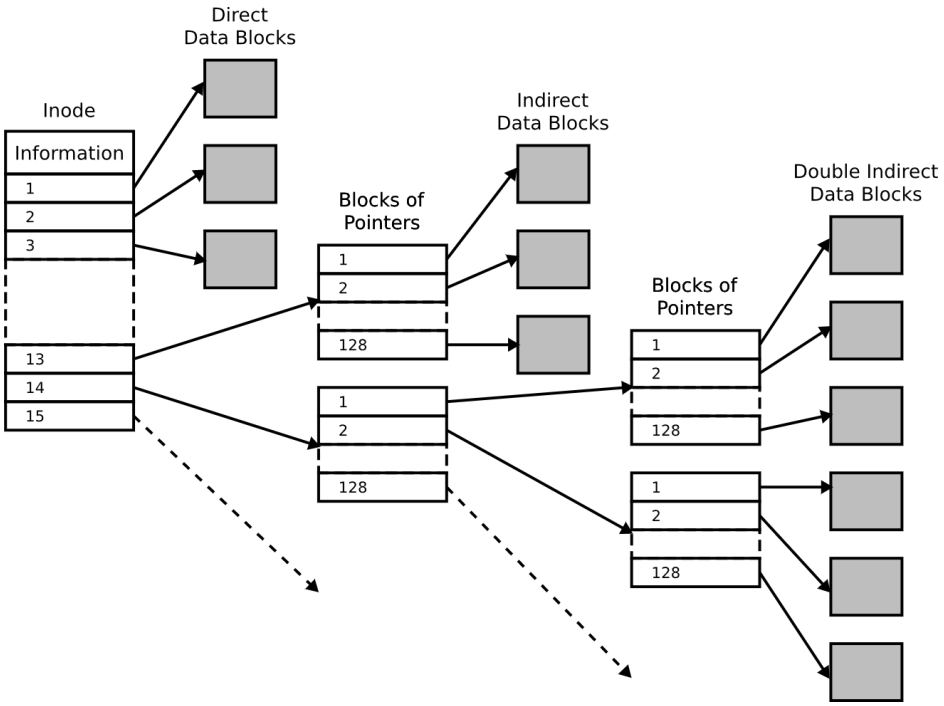


Figure 3.3: Index-type file mapping[1]

sub-item, whether a sub-directory, sub-file, symbolic link, etc.

Figure 3.4 depicts the on-disk data structures of the ext3 file system. Each inode in ext3 features a fixed 128-byte size and holds metadata for a file or directory, such as the file/directory type, size, and modification time (corresponding to bytes 0x00 - 0x27 in Figure 3.4(a)). Within each inode, there are 15 index table entries (bytes 0x28 - 0x63 in Figure 3.4(a)), each pointing to a physical block. Specifically, entries 1 to 12 establish direct pointers to physical blocks, entry 13 points to an indirect block, entry 14 to a double indirect block, and entry 15 to a triple indirect block. As shown in Figure 3.4(b), the dentry structure of the ext3 file system is presented: each sub-item is associated with a dentry that contains a variable-length filename, file length, and a pointer to its corresponding inode.

Index-type file systems use a block bitmap to track the usage status (in use or free) of physical blocks. The ext3 file system uses this bitmap to accelerate block allocation and reclamation. Nevertheless, maintaining data synchronization between the bitmap and the index table is vital, as such synchronization can be disrupted by a system crash. Our model incorporates the bitmap block and verifies data consistency between the bitmap and the index table.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	mode		uid		file size				access time				create time			
0x10	modify time				delete time				gid		link count		block number			
0x20	flags				reserved 1				index table							
0x30	index table															
0x40																
0x50																
0x60	index table				generation				file acl				dir acl			
0x70	faddress				reserved 2											

(a) *inode* of ext2 file system

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00	inode				length		nlen	type								
0x10	file name															
…																

(b) *dentry* of ext file system

Figure 3.4: Data structure of Ext3 file system[1]

Flash File Systems and LFS

Block-based file systems such as NTFS, FAT [39], and ext [40] are optimized for block storage devices, organizing metadata in fixed locations to facilitate indexing. File operations in these file systems update multi-metadata components alongside file data, resulting in frequent small, scattered in-place writes.

On flash-based storage, these scattered writes generate numerous fragments and invalid data across NAND flash blocks. Due to differing write and erase units, these invalid pages remain until garbage collection reclaims the blocks. Frequent writes also cause repeated FTL mapping updates, degrading write performance and NAND flash lifespan.

To address these issues, flash-based file systems like Reliable Flash File System (RFFS) [41, 42], Journaling Flash File System (JFFS) [43], and Yet Another Flash File System (YAFFS) [44] are developed for row NAND or NOR flash memory.

These flash file systems predominantly adopt the Log-structured File System (LFS) architecture [18]. LFS treats the entire file system as a log structure, where all write operations are appended sequentially to the end of the log. When a user updates part of a file, the new data is appended to the end

of the log, and the file index table is modified to point the logical block to the new physical address of the corresponding data block. The original data block is then marked as invalid. The append-only nature of LFS aligns well with the characteristics of NAND flash-based storage devices [45].

The out-of-place update mechanism employed by the LFS always appends new data, leaving old data as invalid. Continuous updates eventually consume the empty blocks in the storage. Consequently, LFS must reclaim invalid data blocks through a process known as garbage collection (GC). When the storage device approaches full capacity, the GC mechanism is triggered to reclaim invalid data blocks for reuse. Typically, garbage collection involves sliding all valid data blocks to the head of the log in the same order, overwriting invalid data blocks, and freeing space at the end of the log.

However, to improve data-copying efficiency, LFS uses segment cleaning to recycle blocks, rather than sliding valid data directly. LFS divides the entire storage space into multiple segments, each containing a fixed number of contiguous blocks. These segments form a linked list that logically represents the log structure. When data is written to the log, LFS selects an empty segment from the segment pool as the active segment, places it at the end of the list, and writes data sequentially into it. Once the current active segment is full, a new empty segment is chosen as the active one. The GC process is initiated when the number of empty segments falls below a certain threshold.

In the segment cleaning mechanism, there are three kinds of blocks. An empty block indicates a physical block that has never been written to since it was first cleaned. Empty blocks may be contained in empty segments or be a part of active segments. A valid block indicates that it has been written and contains the latest version of user data or metadata, whereas an invalid block indicates that it is non-empty and that its data has been updated to another physical block. It is important to note that in an actual block device, there is no electronic status indicating whether a block is empty. A block can be read at any time, and there is no block erase command in the block device interface. The concept of an "empty block" is specific to segment/block management in LFS.

The segment cleaning GC process involves four phases:

1. Selecting a victim segment from the log for reclamation;
2. Copying valid data blocks from the victim segment to the active segment;
3. Updating pointers in the file index table to point to the new location;
4. Erasing the victim segment and placing it in the pool of empty segments;

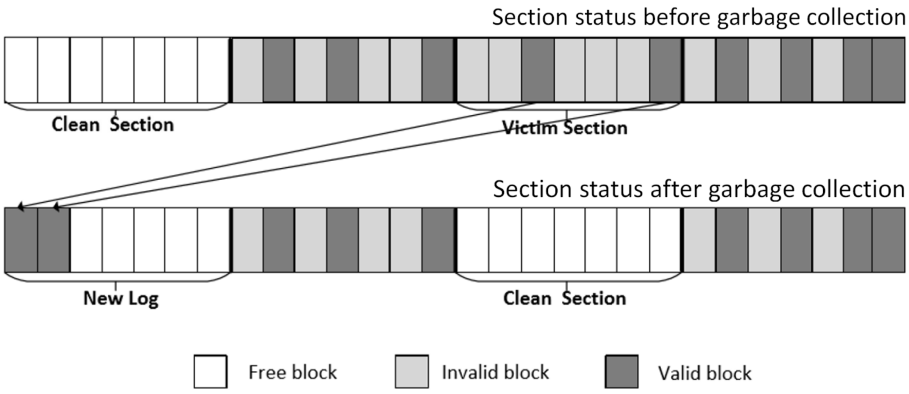


Figure 3.5: Process of garbage collection [2]

To maximize garbage-collection efficiency, segments with the fewest valid data blocks are typically selected as the victim segment for reclamation. Figure 3.5 illustrates the garbage collection process, showing valid blocks copied from the victim segment to the active segment, followed by cleaning the victim segment.

While traditional LFS aligns well with NAND flash-based storage characteristics, several issues require improvement:

1. Wandering Tree Problem [4]: When certain blocks in a file are updated, changing the physical addresses corresponding to these logical blocks, the file index table must be updated. This update propagates recursively through the inode structures of enclosing directories up to the root directory. Consequently, updating a data block in a file often results in rewriting several data blocks, which adversely affects file system efficiency, especially when writing small, non-contiguous data blocks across different files. Additionally, it accelerates wear on NAND flash-based storage.

2. High Reading Cost [5]: Loading the file system incurs significant time and memory overhead. Since the locations of file inodes and index nodes in LFS are not fixed, reading a specific logical block requires accessing multiple inodes or index nodes to retrieve the corresponding physical address. Moreover, some flash file systems, such as YAFFS, store physical-to-logical mapping information in the spare area of data blocks rather than maintaining file index nodes, necessitating nearly full-disk scans to locate specific data blocks. These factors result in longer file read times. Some flash file systems mitigate this by caching portions of the entire file system's metadata, which requires substantial

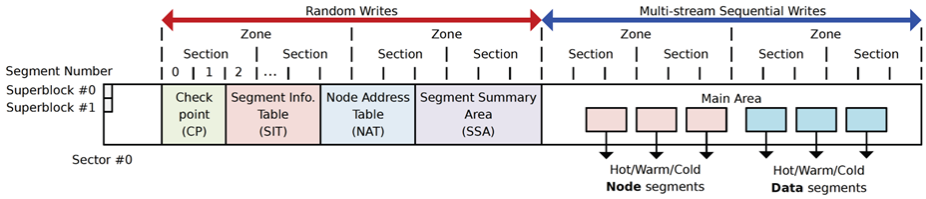


Figure 3.6: On disk data structure of F2FS[3]

memory.

3. Inefficiency in Garbage Collection (GC): Although LFS's segment cleaning mechanism improves garbage collection, its efficiency is suboptimal. In particular, in file systems with a mix of cold and hot data, a significant fraction of cold data is repeatedly moved, thereby wasting NAND bandwidth and reducing lifespan.

Flash Friendly File System (F2FS)

To address these issues, the Flash-Friendly File System (F2FS) [3] was proposed. The following sections introduce the structure and algorithms of F2FS and elucidate how F2FS optimizes LFS performance.

F2FS enhances performance and longevity for modern NAND flash-based storage devices through a generic block device interface. Figure 3.6 demonstrates the on-disk data structure of F2FS. Unlike LFS's fully append-only approach, F2FS reserves a small in-place area for file system metadata to reduce high indexing costs (red arrow range in Figure 3.6). It introduces a node address table (NAT) (blue area in Figure 3.6) to solve the wandering tree issue by breaking the update chain from leaf node to root block. Additionally, F2FS uses multiple log areas to separate different data types, improving garbage collection efficiency.

Wandering Tree Problem Most file systems manage data in tree structures. Directories and files are organized hierarchically, with files as leaf nodes and directories as non-leaf nodes. Furthermore, the file data is structured as a tree. The ext3 file system, for instance, uses a layered index table within its inodes to efficiently manage file data. The inode is treated as the root of the file data index tree; all data blocks are leaves, and indirect or double-indirect blocks are the non-leaf nodes of the tree.

In block-based file systems, data updates affect only the related block due to in-place updating. However, this design presents challenges for LFS. Since LFS doesn't allow in-place updates, any file modification requires the system

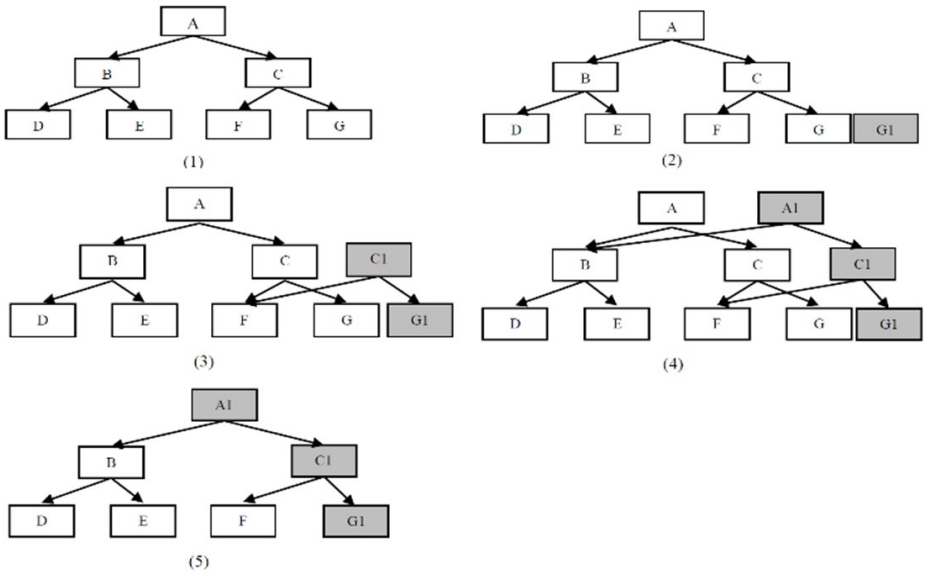


Figure 3.7: Snowball effect of wandering tree problem[4]

to rewrite certain blocks to new physical locations. This change in physical addresses necessitates updating the index table that points to these logical blocks. Consequently, the inode pointing to the index block must be updated, which in turn requires replacing the folder containing the inode. This updating process cascades through the file system hierarchy until it reaches the root directory. This phenomenon, known as the "snowball wandering tree" issue [4], significantly increases write operations when updating small file data. It adversely affects write performance and NAND flash lifespan, especially when writing small amounts of data across different files. Figure 3.7 illustrates this issue: when block G is updated to G1 (sub-fig (2)), its parent block C must also be updated to point to G's new location G1 (sub-fig (3)). This update then propagates up to the root node A (sub-fig (4)).

To mitigate the snowball effect of the wandering tree, F2FS implements a node address table (NAT) in its metadata area, which can be updated in-place. Each node, including inodes, direct nodes, and indirect nodes, is assigned a unique ID (NID). The NAT maps a node ID to its corresponding physical block address. In the parent index table, only the node ID is recorded, not the physical block addresses of the child nodes. When a node is written to a new physical block, only the address mapping in the NAT needs to be updated, eliminating the need to update any nodes that refer to the updated index node. This feature effectively breaks the update chain from the leaf node to the root, thereby alleviating the snowball effect of the wandering tree and its impact on the file system.

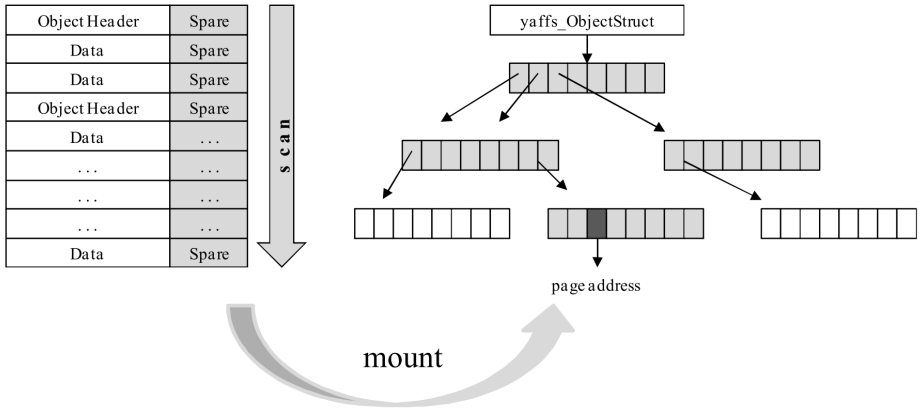


Figure 3.8: The process of mounting YAFFS[5]

High Indexing Cost A file system maps logical file addresses to physical addresses on the storage device [1]. When users want to access data in a file, they must specify the file and the offset within it. The file system then computes the physical location of the data on the storage device using the index table in the file's inode. This mechanism can be considered an address mapping from logical addresses in the file to physical addresses on the storage device. Block-based file systems only need to map logical to physical addresses. However, in LFS, scenarios such as garbage collection or power failure recovery require a reverse mapping from physical to logical addresses. During garbage collection, a file data block must be moved from one physical block to another to reclaim the original segment and update the reference in its parent index table. This process necessitates a reverse mapping from physical to logical addresses. Updating the logical to the physical map may cause the wandering tree issue. Some file systems, such as YAFFS, maintain only a physical-to-logical mapping to avoid wandering tree issues. Consequently, the YAFFS file system scans the entire storage to build a logical-to-physical mapping in running memory, incurring significant time and memory overhead [5, 46]. Figure 3.8 demonstrates the process of mounting YAFFS. It scans all the blocks in the storage to construct the directory tree and file mapping in memory during mounting.

F2FS maintains both logical-to-physical and physical-to-logical mappings to reduce indexing time. It stores a physical-to-logical mapping in its in-place updated metadata area, called the Segment Summary Area (SSA). The SSA is divided into multiple summary blocks, with the number of blocks corresponding to the number of segments in the file system. Each summary block represents a segment and is further divided into three parts: 512 summary

entries, a set of journals, and a footer. Each summary entry corresponds to a physical block in the segment and records the parent node ID (NID) and the offset within that parent node. Using the SSA, F2FS reduces garbage-collection loading and indexing times.

Inefficiency in Garbage Collection F2FS's garbage collection uses an optimized segment-cleaning mechanism derived from traditional LFS. It implements both background and foreground garbage collection. Background garbage collection activates during system idle time, employing a cost-benefit strategy to select victim segments and minimize unnecessary data movement. F2FS leverages Linux's page cache for data movement during background garbage collection, reading blocks to be moved and marking them as dirty. When Linux page management flushes these dirty pages, F2FS writes them to the log's end and invalidates the source blocks. While this approach reduces write counts, it risks data integrity during crashes.

Foreground garbage collection is triggered when insufficient segments are available for continuous writing. It employs a greedy strategy, selecting source segments with the fewest valid blocks to maximize the number of reclaimed segments and minimize block movement. Our focus is on foreground garbage collection because it temporarily blocks user operations while copying blocks, thereby impacting user bandwidth and potentially causing performance degradation or momentary stuttering.

F2FS employs a multi-head log mechanism to optimize garbage collection efficiency. File systems contain node blocks (for inode, index nodes, metadata) and data blocks (for directory or file content). Data blocks are classified as cold data, which refers to rarely modified file data, and hot data, which refers to frequently modified file data. Traditional LFS uses a single, large log that mixes different block types within a single segment. This approach is inefficient during garbage collection because moving valid blocks, regardless of their update frequency, from the victim segment to the active segment consumes storage device bandwidth. Separating cold and hot blocks into distinct segments improves garbage-collection efficiency. Hot blocks have a shorter lifespan, so hot segments require fewer block copies during garbage collection. Conversely, cold segments rarely need reclaiming as they contain few valid blocks.

F2FS categorizes blocks into two main types: data blocks and node blocks. Data blocks contain file data or directory entries, while node blocks store inodes, index nodes, or indirect index nodes. F2FS further classifies blocks as hot, warm, or cold within each main type based on their update frequency. F2FS maintains multiple log structures, each with an active seg-

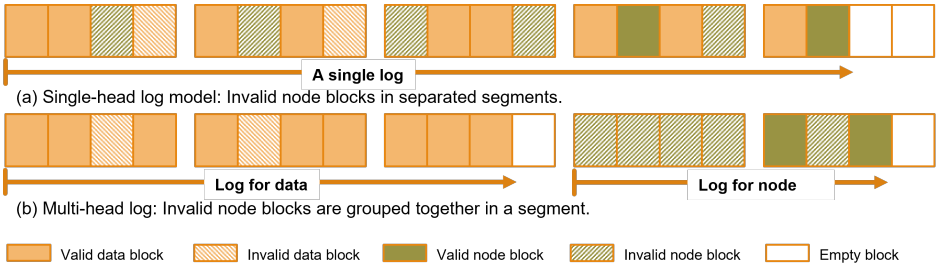


Figure 3.9: Difference between single-head log and multi-head log[6]

ment for appending new data. Different types of data are written to distinct log structures, effectively segregating various data types into separate segments. F2FS can manage up to six log threads, covering hot, warm, and cold nodes, as well as hot, warm, and cold data.

Figure 3.9 shows the difference between single-head and multi-head log. A simulation result shows that separating nodes and data into different logs can significantly improve garbage collection efficiency [6].

3.2.4 File System Crash Consistency

File systems maintain numerous metadata and data structures to manage disk space and organize files. The consistency of these data structures is critical, as a loss of consistency can lead to data corruption, file loss, and even an unbootable system. However, these metadata are typically not stored adjacent to one another. To persist data, a file system must send multiple I/O commands to the storage. If an unexpected power loss or an operating system crash interrupts these related I/O commands, this can result in temporary data inconsistencies. Upon the next system boot, the OS generally executes an `fsck` program to check for and attempt to repair these inconsistencies. In some cases, these inconsistencies are unrecoverable, resulting in a crash consistency error. The root cause of such unrecoverable errors may be design-related, such as an incorrect write order of different metadata [1], or an implementation bug, such as a developer forgetting to persist a critical piece of data.

For instance, in the ext file system, allocating and using a data block involves at least two data structures: a block allocation bitmap that records whether a data block is in use, and a pointer in a file's index table that indicates which file and position is using this block. If a crash occurs and the bitmap is updated but the index table is not, the file system will contain a block that is in use but not allocated to any file. If this condition persists (assuming no similar, repeated occurrences), the only consequence is a loss of

disk space, with no severe data loss. Such issues can also be easily repaired by re-marking the block as unused in the bitmap. The converse, however, presents a more severe issue: if the index table is updated but the bitmap is not, a block will be in use by a file but not marked as allocated. If this issue is not promptly resolved, a subsequent operation may re-allocate this block to another file, leading to data loss because two files share the same data block. This type of error is notoriously difficult to repair [1].

Checking the file system crash consistency issue is a non-trivial challenge. There is no strict, universally agreed-upon definition for consistency. While it is generally accepted that all data synced before a crash must remain intact, the definition of unsynced data, which may be lost, is vague. This poses significant challenges for automated verification. Building upon model checking techniques, we argue that a file system can be considered crash-consistent only if it meets the following three conditions:

1. After crash recovery, all metadata is consistent or can be successfully repaired. This judgment is deferred to the file system's developers, and we verify it by invoking the file system's `fsck` utility.

2. The file system's directory structure and file data must be consistent with one of the valid states from the last sync up to the point of the crash.

3. Subsequent file operations can proceed normally. This is a key differentiator from existing crash consistency-checking tools, which often terminate upon a crash.

The LFS is well-suited to the characteristics of increasingly prevalent flash storage devices. Flash file systems based on log-structured architecture significantly enhance performance on flash storage. F2FS optimizes the log-structured file system by using an in-place area to store newly added metadata, including the Segment Information Table (SIT) for segment and block management, the Node Address Table (NAT) to address the wandering tree problem, and the Segment Summary Area (SSA) to improve garbage collection efficiency. These improvements enhance F2FS performance. However, whether these newly introduced metadata remain consistent during crash recovery and whether they may cause crash consistency errors remains an open question. In Chapter 5, this paper will investigate this issue.

3.3 Model Checking Techniques

Several methodologies can be employed to verify the correctness of a software system. Software testing is valuable for finding bugs in software development. It executes the concrete software system with designed input sequences, called test vectors, and then checks whether the system's output matches the expected results. Since software testing runs on the actual system, it can handle large-scale software systems without requiring abstraction from the concrete system. As a black-box testing method, it also doesn't require testers to have extensive knowledge of the system's internals.

Software testing is valuable for finding bugs in software development. It can handle large-scale software systems without requiring model abstraction from the concrete system. This advantage helps avoid missing errors that might be hidden during the abstraction process. As a black-box testing method, it also doesn't require testers to have extensive knowledge of the system's internals.

However, software testing is not suitable for this work for three main reasons.

1. It cannot exhaust all possible execution paths in the system. While it can reveal the presence of bugs, it struggles to prove their absence, particularly in concurrent systems or external conditions. The interaction between parallel processes is a major source of complexity and errors in system design. These errors are subtle and contingent, depending on thread interactions and external events, making them difficult to reproduce. Our goal is to study file system robustness, which requires handling concurrent processing and external events. Software testing may miss errors in these corner cases.

2. Software testing is inefficient for kernel testing. Since the file system operates as a kernel module, any kernel error typically causes a system crash. To reproduce an error, the system must restart and execute from its initial state, which is impractical for detecting file system errors.

3. While software testing can identify system errors, it provides limited help in analyzing them. It cannot reveal the execution path from the initial to the error states. Error analysis requires adding logs and attempting to reproduce the error path. However, reproducing identical execution paths is challenging because system output depends not only on inputs but also on unpredictable factors, such as the state of other processes in the concurrent system and the timing of the external environment. Since we need to analyze file system errors thoroughly,

software testing proves inadequate for this research.

3.3.1 Model Checking

Model checking is an automated, formal verification technique for ascertaining the correctness of finite-state concurrent systems, such as those found in digital circuits, communication protocols, and embedded software. Its conceptual framework operates on three core pillars: a model, a specification, and a verification algorithm.

The model, M, is a precise mathematical abstraction of the system under verification, typically formalized as a state-transition graph, such as a Kripke structure or a labeled transition system. This model captures all possible behavioral paths and configurations that the system can enter.

The specification defines the desired system properties, expressed as formal logic formulae. These are predominantly written in temporal logics such as Computation Tree Logic (CTL) or Linear Temporal Logic (LTL), which enable the expression of constraints over time and execution sequences. Specifications generally fall into two categories: safety properties (asserting that "nothing bad ever happens," e.g., the absence of deadlock) and liveness properties (asserting that "something good eventually happens," e.g., eventual response to a request).

The core of the method is the model checker, an algorithmic engine that performs an exhaustive state-space exploration. It systematically verifies whether the model M satisfies the specification ϕ, formally denoted as $M \models \phi$. The verification involves traversing all reachable states of the model and checking the validity of the specification formulae against each state and path.

If the model satisfies all properties, the verification concludes successfully. Crucially, if a property is violated, the model checker not only returns a negative result; it also generates a concrete counterexample. This counterexample is an execution trace demonstrating a precise sequence of states leading to the specification violation, providing invaluable, actionable feedback for debugging.

3.3.2 Explicit State Model Checking

Among the various model checking approaches, explicit-state model checking [47] is particularly well-suited for verifying asynchronous software systems, such as the concurrent logic in file systems. As the name implies, this method explicitly enumerates and stores each system state during verification. To mitigate the fundamental challenge of state space explosion,

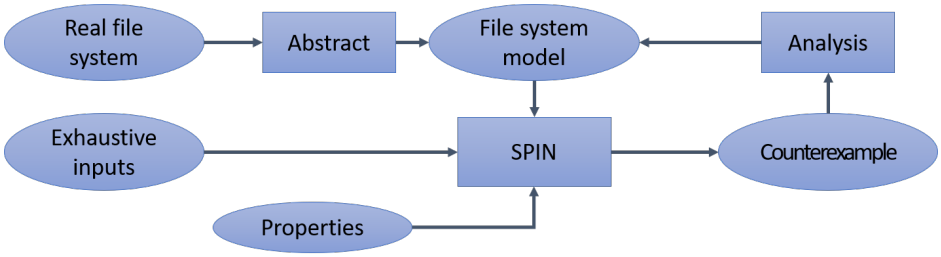


Figure 3.10: Block diagram of model checking

it heavily relies on abstraction techniques to construct a minimal sufficient model of the system, ignoring details irrelevant to the properties being verified. The core of the verification process is the systematic exploration of the state transition graph, for which Depth-First Search (DFS) is a fundamental and widely used algorithm. DFS is favored in many explicit-state model checkers, including SPIN, due to its memory efficiency, as it only requires storing the states along the current search path. This efficiency is crucial for scaling to larger models. Furthermore, explicit-state model checking is often performed on-the-fly, so the entire state graph need not be constructed beforehand. The model checker generates successor states as it progresses, immediately checking properties and halting as soon as a violation is found. This on-the-fly approach, combined with DFS, is highly effective for early error detection and for generating concise counterexample paths that lead to a property violation, which is invaluable for debugging complex systems such as file systems.

3.3.3 SPIN Tool

The SPIN (Simple Promela Interpreter) [48] model checker is a highly influential, widely-used open-source tool for the formal verification of distributed software systems, with a particular emphasis on validating the logic of concurrent processes and communication protocols. Developed primarily at Bell Labs, SPIN is recognized for its efficiency in analyzing asynchronous, event-driven systems and for its ability to detect concurrency-related flaws, such as race conditions, deadlocks, and livelocks. SPIN's verification process is characterized by its on-the-fly verification capability. Rather than directly performing model checking, SPIN generates C source code for a problem-specific model checker, compiles these sources, and invokes the checker. This approach saves memory, improves performance[1].

The systems to be verified are modeled in Promela (Process Meta Language), the input specification language for SPIN. Promela is a non-deterministic,

C-like modeling language designed explicitly for describing the interacting processes of a concurrent system[47]. Its core components include: Processes, Defined as `proctype`, these are the concurrent entities that execute asynchronously; Channels, declared as `chan`, support both synchronous (rendezvous) and asynchronous (buffered) message passing, as well as shared variable communication; Control Structures: It provides familiar constructs like if for non-deterministic selection and do for looping; Data Types: It offers basic data types (bit, bool, int, byte) and arrays; Assertions: The assert statement is used to embed correctness claims directly into the model.

SPIN facilitates the integration of embedded C code into Promela models. In Promela models, we can define C data structures, reference C variables, and call C code. This C embedding simplifies the design of data-intensive models: C data structures can be used to define complex file-system components such as dentries and inodes. SPIN treats embedded C code as a set of deterministic atomic transitions in Promela. Additionally, C code can be used to describe specific atomic operations, for instance, verifying the correctness of the file system, which requires scanning the entire file system.

In this study, we employ the explicit state model checking method to comprehensively verify the robustness of file systems, including the correctness of file system operations, the integrity of file data, and crash consistency. Compared to black-box testing, Model Checking offers greater advantages for file systems that run within the operating system kernel. Model checking examines the file system model rather than executing a concrete system, exhaustively examines all reachable system states, covering all corner cases, including system crashes at any point in time. Model Checking achieves higher testing efficiency, particularly for test items like crash recovery, which are very time-consuming to run on physical machines. When failures are detected, model checking provides the complete execution path. This feature makes it easier to reproduce failures and quickly identify root causes. However, using model checking to examine file systems also presents several challenges.

1. How to generate a workload that provides comprehensive coverage. File systems depend on external operations, which we refer to as workloads. Different workloads produce different states and outcomes. File systems can perform numerous operations, including directory and file operations, file reads and writes, and file system operations. Each operation can be combined with various parameters, and these operations and parameters are subject to different dependency constraints; for example, reading or writing a file requires that the file exists and is open.

2. The problem of state explosion. File systems have vast amounts of

data, and the total number of possible states is astronomical. Different workloads in a file system may ultimately produce identical states. Detecting identical states improves the efficiency of model checking and avoids redundant checks. However, file systems are tree-structured, and identifying identical states is challenging. In our research, in addition to reducing the file system's scale through model abstraction, we encode the file system's directory tree to eliminate identical states and avoid redundant testing.

3. How to define the correctness of a file system. Checking a file system requires a definition of its correctness. We designed a set of properties to define file system correctness, including the correctness of file and directory operations, the integrity of file data, and the definition of crash consistency.

In this study, we defined properties to verify the correctness of file systems, including operational correctness, file data integrity, and crash consistency. We developed an object-based workload generation method. Compared with the skeleton-based workload generator in ACE, the object-based workload generator generates workloads more comprehensively and more efficiently, without creating dependency traces. We also proposed an encoding algorithm to detect isomorphic directory tree structures, thereby avoiding redundant state checks.

Chapter 4

Checking File Mapping Using SPIN

Comprehensively inspecting the correctness of a file system is a non-trivial task, and we need to address the following four key issues: (1) How to extract and establish a file system model; (2) How to traverse the whole state space of the file system exhaustively; (3) How to simulate the occurrence of crashes; (4) How to verify the correctness of the file system after a crash occurs. In this study, we achieve our objectives in two phases. In the first phase, we use the SPIN model checker to verify two small-scale file system models. By exploring the entire state space of file systems, we identify crash consistency issues in both link- and index-type file systems, understand their root cause, and propose a solution. However, we find that manually constructing file system models in Promela is limited, making it challenging to handle more complex file systems. In the second phase, we develop file system models in C to improve their alignment with concrete file systems. Subsequently, we propose a comprehensive file system verification approach and implement it in the MC^3 tool to verify the F2FS file system. This chapter focuses on the first phase, in which file system models are verified using the SPIN tool. The second phase, dedicated to checking full-stack file system models, will be elaborated in the subsequent chapter.

4.1 Proposal

To address the challenges encountered in file system model checking, we propose the following solutions: (1) We proposed an approach to abstract file system models and convert them into Promela languages; (2) We employ an on-the-fly workload creation approach to traverse the file system model

states exhaustively; (3) We add a reset feature to the file system model to simulate the occurrence of crashes; (4) We define file system properties to verify the correctness of the file system after a crash. In the subsequent sections, we first elaborate on our proposed solutions in detail, then describe their implementation in Promela, and finally validate their effectiveness.

4.1.1 File Mapping Model

Most modern file systems are implemented in the C programming language. When conducting model checking of file systems using SPIN, it is imperative first to construct file system models in Promela. Meanwhile, file systems are data-intensive systems with massive volumes of data. A typical file system manages several GB or TB of data (1 GB = 1024 MB; 1 TB = 1024 GB), resulting in an enormous number of states. To avoid state explosion during model checking, we must minimize the size of the file system model.

In this section, we describe how to construct a minimal file system model in Promela, and in the next section, we present concrete examples of link-type and index-type file system models. To establish a minimal file system model, we decompose the file system into two layers: the path-mapping layer and the file-mapping layer. This study focuses on the file mapping layer. We then exclude file system components irrelevant to crash consistency testing, reduce the number of files and data blocks, and aim to construct a minimal file system model. Finally, we implement the model in Promela.

We aim to construct a minimal on-disk data structure for a file system. The essence of a file system is to manage data in units of files. To read data from a file, we typically open the file using its path and name to obtain a file handle. Then, by providing the file handle and a logical address (data offset within the file), the file system maps the logical address to a physical address on the storage device and performs read or write operations at that address. Therefore, we can decompose the file system into two layers: the upper layer maps file paths to files, which we refer to as path mapping. The second layer, for each file, maps logical addresses to physical addresses on the storage device; this process is called file mapping. Figure 4.1 demonstrates the file system structure and two layers of a file system.

In this phase, we focus on the file mapping layer, as it serves as the foundation of the file system, with the path mapping layer built upon it. For instance, ext file systems use `dentry` structures to describe directory hierarchies. We will verify the path-mapping layer in the second phase.

Next, we simplify the on-disk structure of the file system. Typically, in addition to file data, a file system includes a `superblock` to store basic system information, `dentries` to record directory structures, `inodes` and index

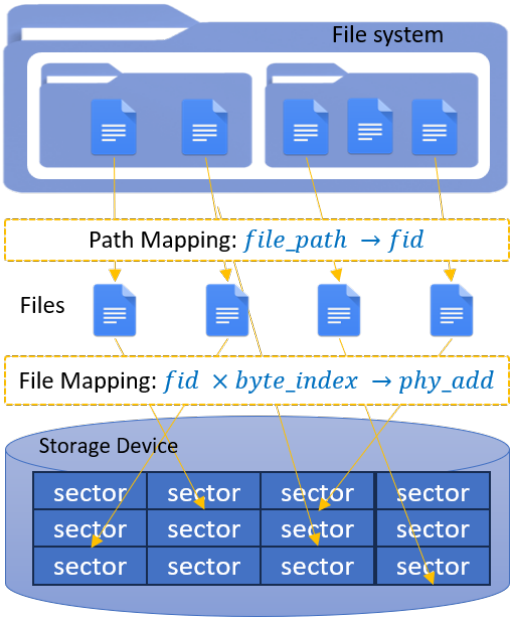


Figure 4.1: Path Mapping and File Mapping

blocks to describe file structures, and structures to manage the allocation of physical storage space. In our model, we hard-code core file system parameters (e.g., storage capacity and root-directory location), thereby eliminating the need for the superblock. Since the path-mapping layer is outside the scope of our current work, the `dentry` structure is redundant. Our minimized model retains only two essential components: `inodes` and the block bitmap (for managing block allocation).

In a standard file system, an `inode` contains not only the index table of file data blocks (or the starting block for link-type systems) but also metadata such as file length, creation/modification timestamps, permissions, and space quotas. Given that timestamps, permissions, and similar metadata have no bearing on crash consistency test results, our file mapping model retains only two critical `inode` attributes: file length and the index table of data blocks (or the starting block position for link-type systems).

Furthermore, we reduce the file system's scale by limiting the number of supported files and data blocks. We limit the number of files to four and the number of data blocks to 16. We consider that (1) computer systems often use powers of two to utilize the expression range of counting variables fully; (2) the number of files should include maximum and minimum boundary conditions, as well as non-boundary conditions. Therefore, 4 is the smallest number that satisfies these conditions.

Finally, we use the following approaches to implement file system operations (e.g., `create`, `write`) in Promela. First, we remove unnecessary code by abstracting the on-disk data. For example, because we store all files in the root directory, parsing the parent directory and determining it before creating a file are unnecessary. Subsequently, we convert operations to a state machine. Each disk write serves as a trigger for a state transition. The state machine is then described in Promela, while embedded C code handles complex operations within individual states.

By following these steps, we can construct a minimal file-mapping model in Promela. Specific examples of index-type and link-type models will be provided in the section 4.2.1.

4.1.2 On-the-fly Workload Creation

A file system is a passive module within an operating system that responds to external commands or API calls. To comprehensively explore the entire state space of a file system, a set of operation sequences (or workloads) that can traverse all states without repetition is required. In traditional black-box testing (e.g., xfstester), a predefined set of workloads is used to test file systems. However, given the enormous scale of file systems, manually designed workloads struggle to cover all possible states and identify corner-case errors. On the other hand, each file system operation has specific preconditions; for example, to create a file b under directory A, the prerequisite is that directory A must exist.

Therefore, we propose an on-the-fly trace-generation approach that exhaustively enumerates all possible operation sequences and traverses the entire file system state space. Figure 4.2 illustrates our concept of the approach. Starting from the file system's initial state, we apply all feasible operations to it, taking into account its current content, including files and directories. Executing an operation transitions the file system to a new state; we then repeat this process for all new states, enabling us to explore the entire set of reachable states starting from the initial state and detect all corner-case errors.

In this phase of the research, we focus on the file mapping layer. Accordingly, we design only four operations per file: `create`, `delete`, `write`, and `read`. The first two operations take only the file as a parameter, whereas write and read additionally require parameters specifying the starting position and the operation length. For the write operation, there are two scenarios: overwrite and append. The former updates existing data blocks in the file (which modifies only already allocated blocks and does not affect metadata for in-place update block file systems, thus having no impact on

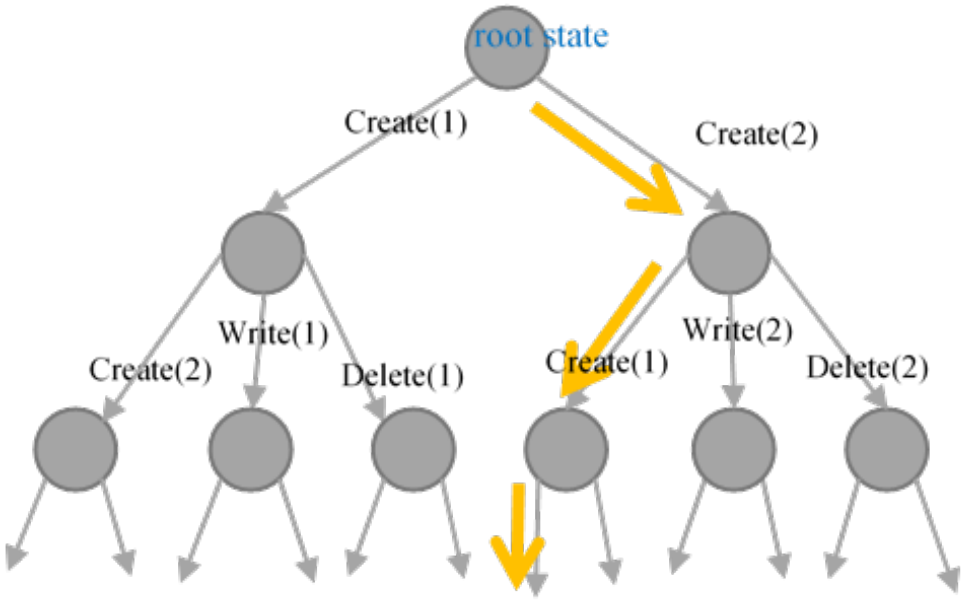


Figure 4.2: Exhaustively searching file system[1]

crash consistency results), while the latter appends new data blocks to the end of the file. Given this, we focus on the append scenario when designing the parameters for the write operation: each write operation appends exactly one block to the file. Scenarios involving multiple blocks can be simulated by executing multiple single-block write operations sequentially.

4.1.3 Crash Simulation

A file system crash is an unexpected cessation of file system operation caused by incidents such as computer system breakdowns or sudden power outages. In such scenarios, data structures in memory are partially or fully not persisted to disk. When the system recovers next time, storage inconsistencies may cause file system errors. Considering that mainstream storage devices (e.g., hard disk drives (HDDs) or solid-state drives (SSDs)) currently feature end-to-end data protection and error correction capabilities, we can assume that disk read/write operations are atomic, i.e., for a read/write command sent by the file system to the disk, either the entire dataset is read/written successfully, or the command is not executed at all. Even in the event of a power outage, no scenario exists in which only part of the data in a single command is modified.

To verify crash consistency issues, we need to simulate file system crashes.

We add a reset signal to the file system model: when the file system receives the reset signal, it abandons incomplete operations, reverts to the initial state, and then reloads the file system from disk. Given that our file system model is implemented as a state machine, the reset signal is designed to be synchronous: it is checked only when the file system transitions between states. If a reset signal is received, the system will not transition to the next state but instead revert to the initial state.

4.1.4 Properties of File Systems

The purpose of this study is to verify the correctness of file systems in the occurrence of a crash. In this section, we elaborate on how to define the correctness of the file mapping model. In addition to clarifying the correctness of the file system, we provide a formal definition of this property in terms of specific attributes. Explicitly defining file system properties facilitates programmatic verification of their correctness. We will detail how to implement the verification of these file system properties in Promela in Section 4.2.5. Before introducing these properties, we define several notations. We use two distinct arrow symbols, $\rightarrow$ and $\Rightarrow$, which belong to different realms of our formalization. Their meanings are strictly separated as follows:

- The single arrow $\rightarrow$ is used in type signatures to denote the domain and co-domain of a function or operation. It specifies what types of inputs a function accepts and what type of output it produces.
- The double arrow $\Rightarrow$ is used in logical formulas to represent implication.

We then define functions that help characterize properties.

- A block allocated state is denoted as $State(b)$, where b is a block. It returns the allocation state of block b; the result may be *allocated* or $free$. In the file system model, we can achieve the block state by reading the block allocation table.
- A block set of a file is denoted as $Block(f)$, where f is a file. It returns a block set in which blocks are assigned to the file f. The block set can be achieved from the file index table.
- A file size is denoted as $SizeOf(f)$, where f is a file. It returns the block number of the file f. The file size can be obtained from the file's `inode`.

Table 4.1: Notations on file system

Symbol	Description
$\rightarrow$	"maps to". Declare the types of functions and operations
$\Rightarrow$	"implies". Define axioms or properties
$State(b) \rightarrow allocated \vert free$	The allocation state of block b. The possible values are $allocated$ and $free$
$Block(f) \rightarrow \{b\}$	A set of blocks which are assigned to the file f
$SizeOf(f) \rightarrow Int$	Denotes the block number of file f
$Read(fs, f, i) \rightarrow b$	Read from file f in the file system fs, return the block content of logical address i
$Write(fs, f, i, b) \rightarrow fs'$	Write block b to logical address i of file f in the file system fs. After writing, the file system updates to fs'

- Read data from a file is denoted as $Read(fs, f, i)$, where f is the file want to read, fs is the file system that contains the file f, i is the logical address of the file. The read function returns the contents of block i from file f.
- Writing data to a file is denoted as $Write(fs, f, i, b)$, where f is the file want to write, fs is the file system before writing, i is the logical address, and b is a content of block needs to be written. The write function returns a new state of the file system fs' that contains the updated file f'.

Table 4.1 summarizes the above notations.

Consistency of File Data

The correctness of a file system primarily encompasses two core aspects, file contents and metadata. Table 4.2 summarizes all properties of file mapping models. Consistency of file data is a fundamental responsibility of file systems, ensuring that user data remains non-corrupted and unaltered. The consistency of file data can be naively defined as follows: for all logical addresses of all files, the data read must be identical to the data written in the most recent [24]. We thus formalize this property below.

$$\forall f, i < SizeOf(f) \Rightarrow Read(Write(fs, f, i, b), f, i) = b \tag{4.1}$$

Violating this property leads to data corruption. We consider this property critical. File data can be read using `read()` operations; we verify this property while the file system is running.

Correctness of Metadata

Metadata refers to essential data structures in the file system that manage files and sustain system operation, distinct from user data. Checking metadata besides file contents serves three key purposes:

- Certain metadata errors do not cause file system data loss but impair system functionality, for instance, leading to the loss of available storage space.
- Detecting metadata errors enables the identification of issues before user data is corrupted, allowing for the discovery of more problems within a limited search depth.
- Diagnosing the root cause of metadata errors is more straightforward than troubleshooting file data errors, as data errors can arise from a multitude of factors.

In our file mapping model, metadata falls into two primary categories: One responsible for physical block allocation, such as the block bitmap in index-type file systems. The other is to manage the mapping from logical blocks to physical blocks in a file, such as the index table in index-type file systems. In link-type file systems, the File Allocation Table (FAT) serves both roles: it handles physical block allocation and indexes all data blocks of a file beyond the first. The index of the first data block is stored in the `start_clust` field of the `inode` (see Figure 4.5).

Metadata consistency requires that allocated blocks be referenced by exactly one file, and that blocks referenced by any file be allocated. Based on this principle, we define the following metadata-related properties:

- **No Dead Block** A dead block is a block that has been marked as allocated in the block allocation table but is not referenced by any file. The "No Dead Block" property requires that any block with $state = allocated$ be referenced by at least one file. We formally define this property as follows:

$$\forall b, State(b) = allocated \Rightarrow \exists f, b \in Block(f) \tag{4.2}$$

  The state of a block can be directly obtained from the block allocation table, whereas verifying whether a block belongs to a file requires traversing every file index table. Violating this property does not directly cause data corruption but reduces the file system's available capacity; thus, we classify it as non-critical.

Table 4.2: Summary of the file system properties[1]

Error mode	Description	Result	Error Level	Recover	Checker
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	Online/off-line
Double pointer	More than one pointer points to a block	Causes data corruption, causes data conflict	Critical	No	Off-line
Contents error	The data read from the file does not equal the data written	Data corruption	Critical	no	Online

- **No Lost Block** A lost block is a block that is referenced by a file but not marked as allocated in the block allocation table. Contrary to a dead block, any block referenced by a file must be allocated. We formally define this property as follows:

$$\forall b, (\exists f, b \in Block(f)\ ) \Rightarrow State(b) = allocated \tag{4.3}$$

  Violating this property may cause the referenced block to be erased or reallocated to another file, resulting in irreversible data loss. Thus, this property is classified as critical.

- **No Double-Pointed Block** A double-pointed block is a block that is referenced by more than one file. The "No Double-Pointed Block" property requires that no block, whether allocated or not, can be referenced by more than one file. We formally define this property as follows:

$$\forall f_i, f_j, file_i \neq file_j \rightarrow block(f_i) \cap block(f_j) = \emptyset \tag{4.4}$$

  Verifying this property requires traversing the index tables of all files and utilizing auxiliary data structures to track block references. For efficiency, this property is checked only after a crash.

4.2 Implementation

We developed file mapping models for both link-type and index-type file systems in Promela and validated them using SPIN. In this section, we decompose the model into its component modules to elaborate on the implementation and verification process of the file system model in Promela. Figure 4.3

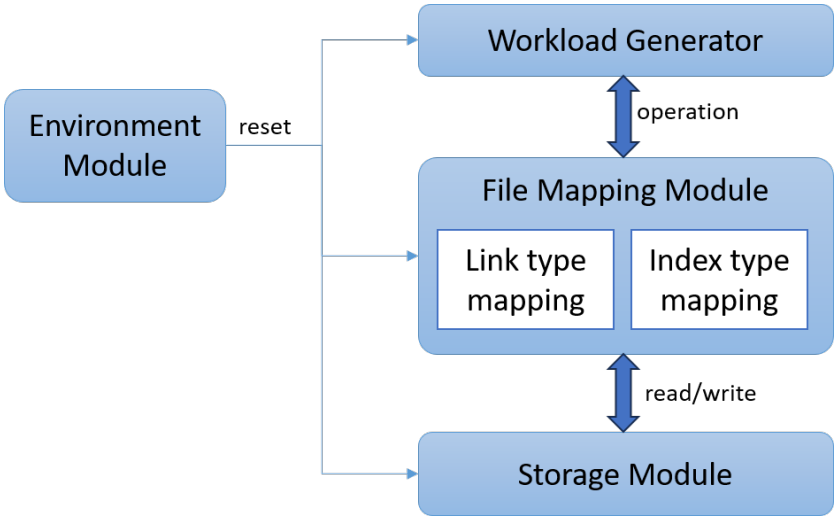


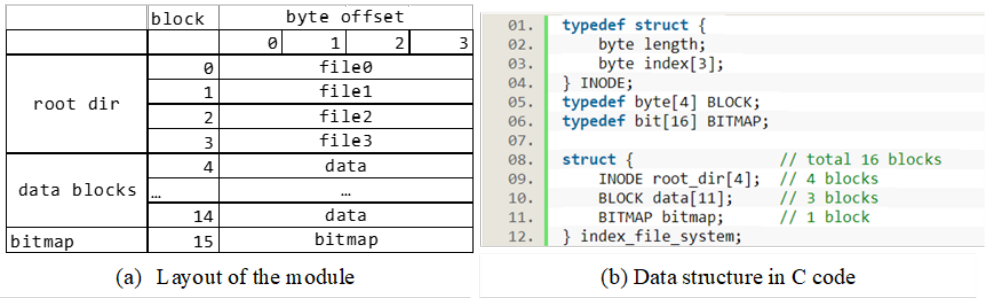
Figure 4.3: Overview of the file system model[1]

demonstrates the modules of the file mapping model. The model comprises four core modules: a file mapping module, a storage module, a workload generator, and an environment module. The link-type and index-type models share a nearly identical architectural framework, in which all modules are universal except for the file-mapping module.

The file mapping module serves as the core component of the model, derived from an abstraction of real-world file systems. By replacing this file-mapping module, we can verify a variety of file systems. The model operates the workload generator, which, in turn, invokes the operational functions within the file-mapping module. The file mapping module then calls the storage module's read or write functions to load data from or store data to the storage device. The storage module simulates a block-based storage device that houses the file system's data. The environment module runs in parallel with the other three modules and asynchronously sends a reset signal to simulate system crashes. Upon receiving the reset signal, the other three modules clear all variables, revert to their initial states, and suspend operation until the environment module releases the reset signal, after which they resume. Detailed introductions to each module will be provided in the subsequent sections.

4.2.1 File Mapping Module

This section presents a method for building a Promela model of file systems. We created two file mapping modules for link-type and index-type file sys-



(a) Layout of the module

(b) Data structure in C code

Figure 4.4: Data structure of index-type file mapping module [1]

tems. The link-type file mapping module was abstracted from the "FAT 16/32 File IO Library v2.6" [49], while the index-type one was abstracted from ext2 source code [40]. First, we describe how to construct on-disk data structures for minimal file-mapping models. Next, we explain how to use Promela to describe file system operations. We then introduce optimizations to mitigate state exploration during checking. Finally, we discuss the gap between the abstracted model and a concrete file system.

Abstraction of On-disk Data Structure

Figure 4.4 depicts the on-disk data structure of the index-type file mapping module, which is derived from the abstraction of the ext2 file system [40], encompassing (a) the layout and (b) the C-code data structure. This structure can be partitioned into three components: a four-block root directory, a single-block bitmap, and 11 data blocks. The root directory also contains four `inodes`, each with a one-byte length field and a three-byte index table, with one byte allocated to each entry. Consistent with the ext2 file system, our index-type file system model supports indirect index blocks. Specifically, the first two entries point to data blocks, while the third entry points to an indirect block that contains four entries targeting data blocks. Consequently, each file can include up to six data blocks. The indirect block is dynamically allocated from the data blocks. Given that the `inodes` are statically assigned, optimization of the bitmap for these `inodes` is feasible. We use a single data block bitmap to manage data block allocation, with 16 bits per bitmap block, where each bit indicates the allocation status of a data block.

The link-type file mapping model was constructed using the same approach, derived from the abstraction of a real FAT file system (FatIO [49]). Similar to the index-type model, the link-type model supports four fixed `inodes` (note: the actual FAT file system lacks the `inode` concept; this concept is introduced herein for unified description and simplified comparison).

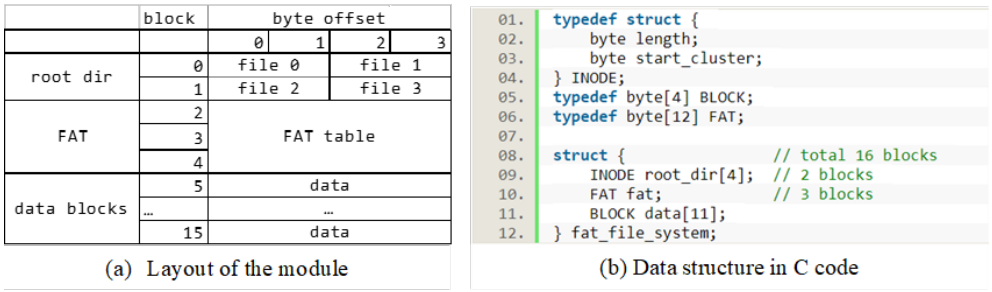


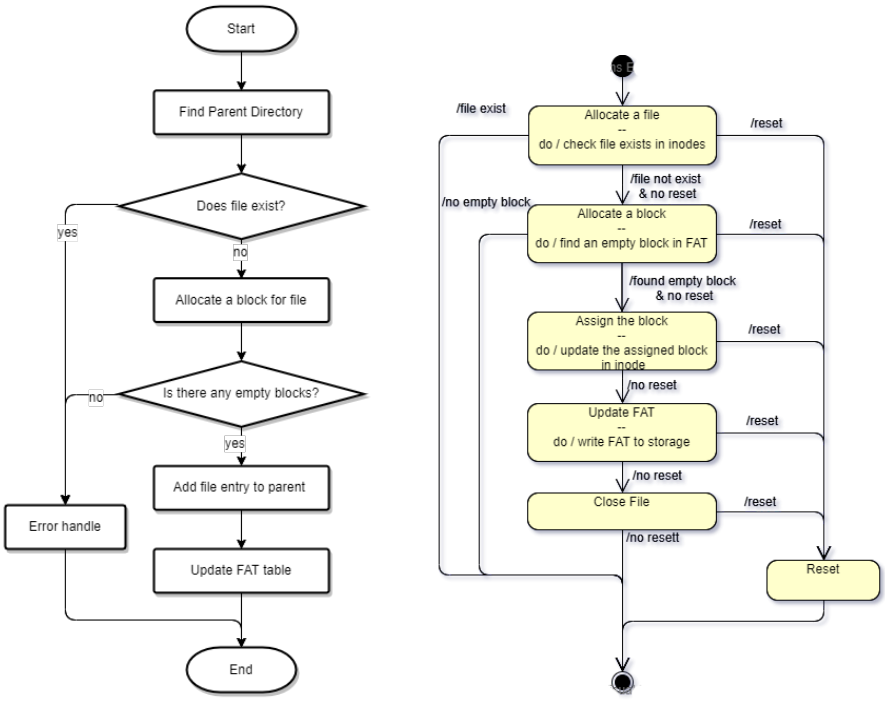
Figure 4.5: Data structure of link-type file mapping module [1]

In the link-type model, the file index table is managed by the FAT, so each `inode` only requires two bytes: one records the file length, and the other stores the first block of the file. As a result, each block contains two `inodes`, and four `inodes` occupy two blocks. The entire storage device contains 16 blocks; thus, three blocks are allocated to the File Allocation Table (FAT), with each byte representing one entry. In total, 11 data blocks are available, which is consistent with the index-type model.

Figure 4.5 illustrates the on-disk data structure of the link-type file mapping module, including (a) the module layout and (b) the C-code data structure. The data is divided into three parts: a two-block root directory, a three-block FAT table, and 11 data blocks. The root directory contains four `inodes`, indicating a maximum of four supported files. Since the FAT file system does not support hard links, only one file entry can be associated with a file's `inode`. We integrate the file entry (`dentry`) into the `inode` structure; each `inode` has two fields: a one-byte file length field (indicating the file size in bytes) and a one-byte start block field (pointing to the head of the file's block list). A value of FFh in the start block field denotes an unavailable file. Each FAT entry corresponds to one data block; the FAT table contains 12 entries, which is sufficient to store 11 data blocks.

Abstraction of File System Operations

This section presents an example of modeling the create operation for the link-type file system model. Figure 4.6(a) illustrates the flowchart of the create operation in a real FAT file system. First, the FAT file system locates the target file's parent directory by parsing the file path. Next, it checks whether the specified file name already exists. Then, the file system loads the FAT table and searches for an available block. If a free block is found, the file system creates a new file entry and adds it to the parent directory. Finally, the FAT table is updated to mark the block as in use.



(a) Flowchart of the create operation in the FAT file system

(b) Abstracted create operation in the link type file system model

Figure 4.6: Flowchart and state machine of CreateFile operation [1]

Figure 4.6(b) shows the state machine of the abstracted create operation. Because our model includes only a root directory, the step of locating the parent directory is omitted. Additionally, because loading the FAT table and searching for a free block do not involve writing data to storage, these two procedures are integrated into a single "block allocation" step. As the workload generator drives the file system model, there is no need to provide error feedback. In our model, the create operation terminates immediately if the target file already exists or no free blocks are available.

Figure 4.7 presents the Promela code implementing the create-file operation, which describes the aforementioned state machine. Each branch within the `do-od` block corresponds to a specific state, identified by the `sub_state` variable. The state machine starts executing from the initial state, where `sub_state=0`. Each state specifies the ID of the next state to transition to. The first state in the code is the reset state, which is triggered by the `sys_reset` variable.

```
do
:: sys_reset == true ->
    stFS[id] = FS_RESETING;
    break;

:: atomic { (!sys_reset && sub_state == 0 &&
    !file_lock[fid] && root[fid*2] == DENTRY_EMPTY) ->
    sub_state = 1;
    length = root[fid*2+1];
    file_lock[fid] = true;
}

:: (!sys_reset && sub_state == 0 &&
    !(!file_lock[fid] && root[fid*2] == DENTRY_EMPTY) ->
    break;

:: atomic {(!sys_reset && sub_state == 1 && !fs_locked) ->
    fs_locked = true;
    fat_entry = DENTRY_EMPTY;
    compare = true;        // use for C code return
    c_code {               // find empty entry in FAT
        int ii = 0;
        for (ii = 0; ii < DATA_CLUSTER; ++ii) {
            if (now.fat[ii] == FAT_UNUSED) break;
        }
        if (ii>=DATA_CLUSTER) {
            now.disk_full = 1;
            PFS->fat_entry = DENTRY_EMPTY;
        } else {
            PFS->fat_entry == ii;
            now.fat[ii] = FAT_ENDOFCHAIN;
            now.root[PFS->fid*2] == ii;
            now.root[PFS->fid*2+1] = 0;
        }
    }
    fs_locked = false;
    sub_state = 3;
}

:: (!sys_reset && sub_state == 3 && disk_full) ->
    file_lock[fid] = false;
    break;

:: (!sys_reset && sub_state == 3 && !disk_full) ->
    write_root;        // write back durty root
    sub_state = 4;

:: (!sys_reset && sub_state == 4) ->
    fat[fat_entry] = FAT_ENDOFCHAIN;
    write_fat;
    sub_state = 5;

:: atomic { (!sys_reset && sub_state == 5) -> // close file
    ref_file_len[fid] = 0;
    flush_cache;
    file_locke[fid] = false;
    break;
}

od
```

Figure 4.7: Promela code of the CreateFile in the link-type file mapping module [1]

Optimization of File Mapping Modules

In our Promela model, the on-disk data of the file system is part of the state. Any change to any part of the on-disk data results in a new file system state. In the previous section, we abstracted the file system into a file-mapping model and reduced its on-disk data to 256 bits (16 blocks, each block containing 4 bytes, each byte 4 bits). Even so, the maximum number of file system states is 2^{256}, which remains astronomical. Next, we leverage the file system's data characteristics to further reduce the number of file system states. These characteristics do not affect normal file system usage.

A file system's data consists of two components: metadata and file data. Metadata is used for file system management, such as `inodes` and block allocation. File data refers to data stored in files. When verifying file system correctness, we ensure the integrity of file data without considering the data content itself. As long as the file structure remains the same, different file contents can be considered equivalent to the same file system state. We leverage this characteristic to optimize the data in the file-mapping model.

Filling File Data by Hash Value When we invoke the write operation, we write a Hash value instead of a random value. The hash value is calculated from a triad,

$$value = Hash(fid, blockid, offset) \tag{4.5}$$

Where fid is an integer number that uniquely identifies a file, it also indicates the `inode` number of a file. The *blockid* is the physical block address where the values are stored, and the offset is the logical location inside the file. Using the Hash value makes the model simpler, so it does not need to remember the value we wrote to the file. To verify what we wrote to the file, we can compute its hash. Furthermore, if we create and write a file from a different execution path, the random value in that path should differ, causing the SPIN tool to generate two distinct states even when we check the same file. A hash value ensures that the same file has the same value, regardless of how it is created or written. Thus, the search tool treats two states with the same file as identical, thereby reducing the number of states during file system search.

Filling Invalid Block with a Fixed Value We consistently set unallocated blocks to a fixed value (e.g., 0xF) instead of retaining their original data. In practical file systems, when a block is recovered, the pointer referencing the block is removed while the block's internal data is preserved for performance optimization purposes. In our file mapping module, unallocated blocks are irrelevant to core functionality. However, retaining the

original data can lead to inconsistencies in the observed state of identical file systems, solely due to variations in the data stored in their unallocated blocks. Filling unallocated blocks with 0xF ensures uniform data storage across all such blocks, thereby mitigating the state space during file system verification. By leveraging these optimization measures, we can conduct a comprehensive search and verification of our file system models within an acceptable time frame.

Gaps in The Abstracted Models

Nevertheless, discrepancies exist between our abstract models and practical file systems. These gaps are primarily attributed to constraints inherent to the scope of this research. One key gap is that our module does not incorporate the file system's directory structure, including tree-structured directories, symbolic links, and hard links. Consequently, our model cannot detect directory-related errors in real-world file systems. Such aspects fall outside the scope of this study, which focuses on the fundamental-level correctness of file system metadata and user data in the event of unexpected crashes. The directory structure represents a higher layer in the file system hierarchy, and its correctness depends on the integrity of the fundamental layer. We intend to integrate the directory structure into our model in future research endeavors.

Another gap is the absence of system cache representation in our model. System cache is managed by storage device drivers, with implementations varying across different operating systems, an aspect beyond the scope of this study. System cache may reorder write commands sent to the storage device, potentially causing errors during unexpected crashes that our model cannot detect. However, this study establishes minimum requirements for maintaining file system correctness under crash conditions. The results can serve as a reference for formulating appropriate preventive strategies when designing the cache layer, such as implementing barrier mechanisms at critical junctures. We believe that meeting these minimum requirements will provide valuable insights for future cache design in systems.

On the other hand, counterexamples identified in the abstracted models may be spurious. Validating these counterexamples against real file systems is necessary to determine whether they correspond to actual errors. We will elaborate on our validation approach and present a specific example in Section 4.3.4.

4.2.2 Workload Generator

The workload generator module employs a dynamic workload-generation method. It leverages Promela's non-deterministic constructs, namely `do-od` and `if-fi`, to exhaustively enumerate all possible operations (Listing 4.1). The `Tester` process type (lines 8-25) constitutes the core code of this module. The `id` parameter specifies the task ID when this process type is executed multiple times to simulate concurrent tasks in the file system. Within the `do-od` statement, the first branch (line 11) verifies whether the file system has crashed and is awaiting mounting; if so, the file system must be mounted first. The remaining branches (lines 12-23) of the `do-od` statement are used for non-deterministic operation selection. In this model, each operation supports four parameter options, corresponding to the four files that the model can handle. We invoke the `SelectFile()` function (lines 1-8) to non-deterministically select a file (identified by `fid`). Should an operation be incompatible with the target file, for example, attempting to append data to a non-existent file (lines 15-17), the operation will return without producing any effects.

4.2.3 Storage Module

The storage module is responsible for accommodating file system data and emulating a block-based storage device. It consists of 16 sectors (denoted by `CLUSTER_NUM`), with each sector containing 4 bytes. This module facilitates block read and write operations, which are treated as atomic operations, because most practical storage devices are equipped with data protection mechanisms such as ECC. For this reason, we implement these functions in integrated C code, thereby mitigating the model's complexity. Listing 4.2 presents the code for the `ReadSector()` function (lines 6-19) within the storage module: this function copies data from the specified sector of the storage device (i.e., `now.storage_data`) to the buffer provided by the caller (lines 13-5). Lines 1-5 illustrate how to invoke the storage functions in Promela.

4.2.4 Environment Module

The environment module is designed to simulate external failures, such as unexpected system-wide crashes. It operates in parallel across multiple file system threads to asynchronously send reset signals. Figure 4.8 depicts the state machine of the environment module, which comprises two states: resetting and running. In the running state, the environment module non-deterministically transitions to the resetting state while simultaneously trans-

```
1   inline SelectFile(fid) {
2       if
3       :: fid=0;
4       :: fid=1;
5       :: fid=2;
6       :: fid=3;
7   }
8   proctype Tester(byte id) {
9       byte fid;
10      do
11      :: stFS[id] == FS_RESETING -> // Files system
            crashed and pending for mount
12      :: stFS[id] == FS_READY && freesize>0 ->
13          SelectFile(fid);
14          CreateFile(fid);
15      :: stFS[id] == FS_READY && freesize>0 ->
16          SelectFile(fid);
17          Write(fid)
18      :: stFS[id] == FS_READY ->
19          SelectFile(fid)
20          DeleteFile(fid)
21      :: stFS[id] == FS_READY ->
22          SelectFile(fid);
23          Verify(fid);
24      od
25  }
```

Listing 4.1: Workload generator in Promela[1]

```
#define read_sector(sec,sec num, buf){
    assert((sec+sec num)<=CLUSTER_NUM);
    sector =sec;
    c code{ Readsector(PFs->sector,sec_num,PFs->buf);}
}
c_code {
    void ReadSector(int sec,int sec_num, void * buf) {
        unsigned char*buf=(unsigned char *)bif;
        int ss =0;
        for(ss=0; ss<sec num; ++ss) {
            int i;
            int ptr=(sec+ss)*CLUSTER SIZE;
            for(ii=0;ii<CLUSTER SIZE;++ii) {
                buf[ii]=(now.storage_data[ptr++ii] & 0
                   x0F);
                offset+=CLUSTER_SIZE;
            }
        }
    }
}
```

Listing 4.2: Sample code of the ReadSector function in the storage module[1]

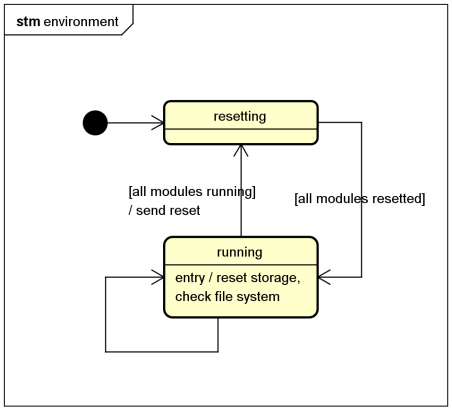


Figure 4.8: State machine of environment module

mitting a reset signal, thereby mimicking an unexpected crash. Once all other modules have completed their reset procedures, the environment module reverts from the resetting state to the running state. Upon entering the running state, it performs an offline check of the file system, as detailed in Section 4.2.5. Subsequently, it mounts the file system and activates the workload generator to initiate subsequent testing. Using the SPIN model checker, we can conduct exhaustive exploration of the state space between the two asynchronous submodels, enabling comprehensive evaluation of the system across all potential crash scenarios.

4.2.5 Property Checkers

To verify the files, we designed an array, `ref_fs`, to record which files are valid and the amount of data written to each. Each element in the array corresponds to a file and records its length. If the file is invalid or missing, we set the element to the special value `INVALID`. When a file is created, we record the file's actual length in the corresponding element. We also designed an online checker that verifies file data consistency while the file system is running, and an offline checker that verifies metadata consistency after crashes.

Online Checker

The online checker is implemented as an operational component within the file mapping module, running in parallel with core operations, including file creation, deletion, and writing. It is continuously schedulable by SPIN and invoked during the file system mounting process. Upon execution, it reads

```
1  int Verify(fid) {
2      if (ref_fs[fid].length == INVALID) return 1;
3      inode = Read_inode();
4      if (ref_fs[fid].length != inode.length) return 0;
5      offset = 0;
6      for (block=0; block<file_block_num; ++block) {
7          physical_block=GetPhysicalBlock(fid,block);
8          if (!IsAllocated(physical_block) ) {
9              Assert(0);  //lost block error
10         }
11         buf = ReadSector(physical_block);
12         for (ii=0; ii<BLOCK_SIZE; ++ii, ++offset) {
13             if (offset >= ref_fs[fidl.length) break;
14             if (ref_fs[fid].content[offset]!= buf[ii]{
15                 Assert(0);// data corruption error
16             }
17         }
18     }
19     return 1;
20 }
```

Figure 4.9: The code of the online checker[1]

every block of all valid files, verifies whether the corresponding block has been properly allocated (in compliance with Property 4.3), and compares the block data against a precomputed hash value (to validate Property 4.1). Because it executes during file system runtime, the verification operation may run concurrently with other operations. Since read operations do not alter the file system state, the online verification operation is implemented as an atomic operation. However, it supports concurrent execution without interfering with other operational processes. Figure 4.9 presents the pseudo-code of the online checker's verification function. This function first targets a specific file via its `fid` (line 1). It then checks if the actual file length matches that of the corresponding reference file (line 4). Subsequently, it scans all blocks utilized by the file and verifies their allocation status (lines 8-10); if a block is found unallocated, an assertion is triggered to flag the error. Additionally, it compares the content of each block with the reference file (lines 12-17); any mismatch triggers an assertion in Promela. The verification function is consistent across both file system models, with the only distinction being the implementation details of the `GetPhysicalBlock()` and `IsAllocated()` functions (lines 7-8).

Offline Checker

To ensure metadata consistency, a full scan of the entire file system is required to detect anomalies such as dead blocks and double-referenced block errors. Accordingly, an offline checker is integrated into the environment module (rather than the file mapping module) and runs when the file system is unmounted. Following a system crash and a subsequent filesystem reset, the environment module invokes the offline checker before remounting the filesystem. Implemented in embedded C, the offline checker executes atomically. Figure 4.10 illustrates the pseudo-code of the offline checker. After the file system is unmounted, the checker first scans all files in the system (line 2) and all blocks assigned to them (line 3), then cross-references the allocation table to verify the status of block allocations (lines 4-6). If a block is not recorded in the allocation table, a lost block error is detected, and an assertion is triggered (line 5). To verify that each block is assigned to only one file, the checker marks each accessed block (line 10); if a block is already marked, a double-referenced error is identified, and an assertion is issued (lines 7-9). Finally, the checker scans all blocks marked as allocated in the allocation table (lines 13-17); any unmarked block indicates a dead block error (line 15). Since dead block errors are non-critical, our model reclaims such blocks to restore system integrity.

Ultimately, both file system models are implemented in Promela across three files, totaling 1770 lines of code. The storage model, shared by both file system variants, is defined in "storage.pml," which contains 105 lines. The link-type file mapping module, along with integrated copies of the environment module and test model, is documented in "link.pml" (744 lines). The index-type file mapping module is implemented in "index.pml" (921 lines) and is similar to the link-type model; it includes embedded copies of the environment module and the workload generator. A summary of the Promela models is provided in Table 4.3.

4.3 Checking and Error Analysis

This section presents the verification results for both link-type and index-type file mapping models. We first elaborate on the validation findings under single-thread and multi-thread scenarios in Sections 4.3.1 and 4.3.2, respectively. The verification process adopts SPIN's depth-first search algorithm. For optimized memory utilization, we employ the BITSTATE hashing technique for state compression. This approach uses a bit array to store the state hash table, with the hash value of each state serving as the index for array

```
int OffLineChecker() {
    for (blk=0; blk<BLOCK_NUM; ++blk) mask[blk]=0;
    for (fid=0; fid<4; ++fid) {
        for(blk=0; blk<file_block_num[fid]; ++blk) {
            int phy blk =block of fine[fid] [blk];
            if (Allocated(phy blk)){
                return 0;    // lost block error
            }
            if (mask[phy blk]) {
                return 0:    // double pointed error
            }
        }
    }
    for (blk=0; blk<BLOCK_NUM; ++blk) {
        if(mask[blk]==0){
            return 0:        // dead block error
        }
    }
    return 1;
}
```

Figure 4.10: The code of the offline checker[1]

Table 4.3: Summary of the file system models[1]

File	Dependency	Contents	Scale (lines)
storage.pml	non	Storage module	105
link.pml	storage.pml	Link-type file mapping module; Workload generator; Environment module;	744
index.pml	storage.pml	Index-type file mapping module; Workload generator; Environment module;	921

access. While BITSTATE hashing is a lossy compression method, it offers significant memory efficiency advantages. It achieves performance comparable to standard hashing algorithms with only 2% of the memory overhead, albeit at the cost of reduced state coverage to 97%[48]. Counterexamples were identified under both single-thread and multi-thread operating modes. Subsequently, Section 4.3.3 details the design of countermeasures to address these issues and enhance the robustness of file system models. Finally, Section 4.3.4 applies one representative counterexample and its corresponding solution to a real-world file system. Experimental validation confirms that the counterexample reproduces in the actual file system and that the proposed fix effectively resolves the issue, demonstrating the effectiveness of our models in detecting crash-induced errors.

4.3.1 Model Checking for Single-thread Mode

Initial verification focused on the file system models under single-thread conditions. Normal operation was confirmed to be error-free and crash-resistant. However, the introduction of simulated unexpected crashes yielded several counterexamples. Figure 4.11 illustrates a double-reference error instance in the link-type file mapping module.

In standard execution, when the test component invokes the file creation operation `1:CreateFile(0)`, the file system first searches for unallocated blocks in the FAT via `1.2:FindCluster()`. Upon locating a free block, it assigns the block to the new file, updates the corresponding `inode`, and marks the block as allocated by setting its FAT entry to an end-of-chain indicator. Finally, the file system writes the updated `inode` (`1.3:Write(inode)`) and FAT table (`1.4:Write(fat)`) to the storage medium. Data consistency between the `inode` and FAT must be maintained throughout this process.

In the error scenario, an unexpected crash occurs between the writing of the `inode` and the FAT table. Post-recovery, the `inode` retains the updated block assignment, but the FAT table remains unmodified. This creates a discrepancy: the file reports the block as allocated, whereas the file system still considers it free. Subsequent file-creation requests may cause the same block to be reassigned to a new file, thereby triggering a double-reference error. Concurrent writes to this shared block by both files can lead to data corruption or loss.

Root cause analysis attributes this issue to violations of data consistency. File operations typically involve updating multiple interdependent data structures stored separately, requiring sequential write commands to modify them. Such write sequences are vulnerable to interruptions caused by crashes or power outages, thereby disrupting data consistency.

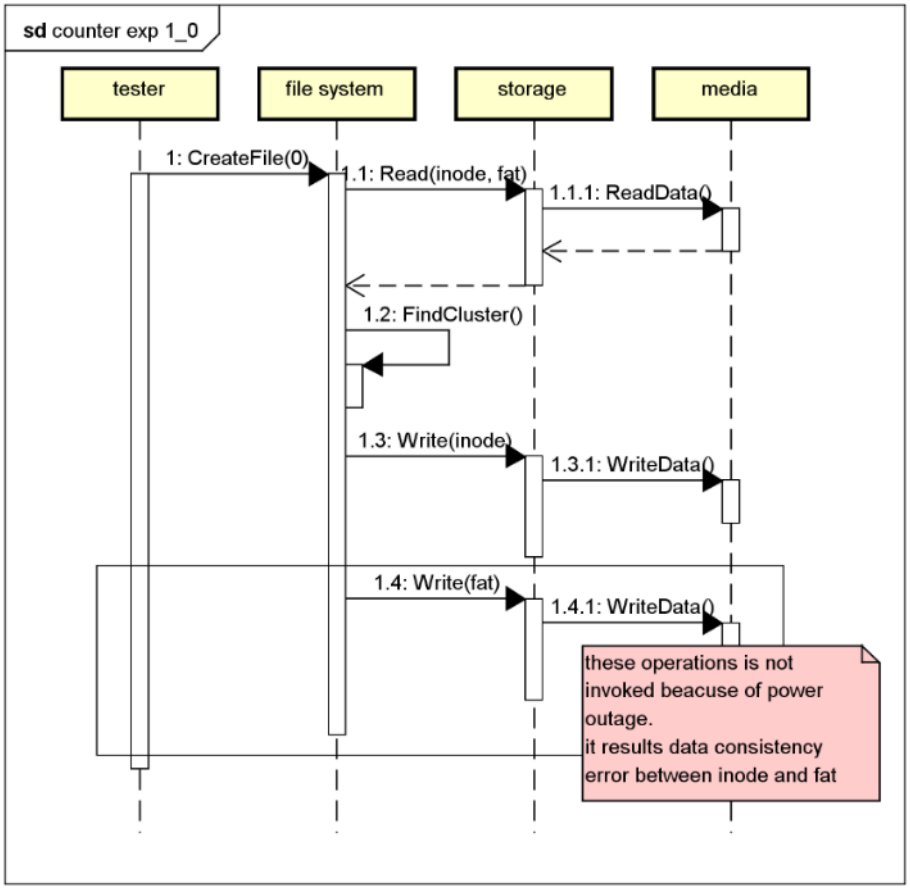


Figure 4.11: Counterexample of double-pointed error[1]

Table 4.4: Verification result[1]

Error mode	Error level	Link type		Index type	
		Single-thread	Multi-thread	Single-thread	Multi-thread
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
Content error	Critical	No error	Error	No error	Error

* To expose the critical error, we skip checking the dead block error

In the identified counterexample, file creation necessitates updates to both the file's `inode` and the FAT table, two distinct data structures that are written via separate commands. A crash between these two write operations disrupts data consistency, leading to conflicting block allocation states between the file and the file system and ultimately causing the double-reference error.

To address this consistency issue, we explored leveraging the volatile cache inherent in most storage devices. Under this approach, all data destined for storage is first cached in the volatile memory. Once all interdependent data updates are complete, the file system issues a FLUSH command to force the storage device to transfer all cached data to non-volatile media. Since the FLUSH command is executed atomically within the storage device, any crash during a file operation will discard all unflushed cached data, thereby preserving a consistent state on the non-volatile medium.

We implemented this volatile cache functionality in the storage model, designing a cache with a capacity matching the storage device. Upon model reset (or simulated power-on), all non-volatile media data is loaded into the cache. Runtime read/write operations exclusively interact with the cache; the FLUSH command triggers a full cache-to-non-volatile media data transfer. Crash-induced cache data loss results in restoration from non-volatile media, discarding all post-last-FLUSH updates. Re-verification of both file mapping modules with the enhanced storage model in SPIN confirmed the elimination of single-thread-mode errors, even under crash conditions. Table 4.4 summarizes the verification results using this mechanism.

4.3.2 Model Checking for Multi-thread Mode

Verification of the file system model under multithreaded conditions still uncovered critical errors because the aforementioned volatile cache mechanism fails to address multithread-specific consistency issues. Figure 4.12

presents a lost block error counterexample involving two concurrent threads: thread one executes `1:CreateFile(0)` (file creation), while thread 2 performs `2:DeleteFile(1)` (file deletion).

In the create operation, the file system reads the `inode` and FAT (`1.1: Read(inode, fat)`), locates a free block (`1.2: FindCluster()`), updates the `inode` and FAT, writes these updates to storage (`1.3: Write(inode)`, `1.4: Write(fat)`), and issues a FLUSH command (`1.5: Flush()`) to complete the operation. In the delete operation, the file system reclaims all blocks associated with the target file (`2.2: RecycleCluster()`), updates the FAT and `inode`, writes these changes to storage (`2.3: Write(fat)`, `2.4: Write(inode)`), and issues its own FLUSH command (`2.5: Flush()`).

While these operations appear concurrent to the user, storage devices do not support parallel access, requiring the file system to serialize storage commands. A critical issue arises when thread 2's FLUSH command forces all cached data (including thread 1's unflushed `inode` updates for file 0) to be written to non-volatile media. The storage cache does not distinguish data by thread, so a crash occurring immediately after thread 2's FLUSH command (`2.5: Flush()`) leaves thread 1's data in an inconsistent state: the `inode` is persisted, but the corresponding FAT update is not. This indicates the need for a thread-aware consistency mechanism independent of storage cache functionality to enhance the robustness of the file system under multi-threaded access.

Analysis of multi-thread counterexamples reveals that errors consistently occur during new block allocation. Block allocation involves updating multiple entries in the mapping and block allocation tables, and inconsistencies can arise from writes to upper- and lower-layer mapping tables, as well as from writes to the mapping and allocation tables. In the link-type file mapping module, the upper layer corresponds to the start block entry in the `inode`, while the lower layer comprises FAT block-chain entries (the FAT also serves as the block allocation table). An interruption of the write sequence, either by a crash or by a FLUSH command from another thread, between `inode` and FAT updates disrupts data consistency and triggers critical errors.

4.3.3 Improvement of Robustness

While complete elimination of all errors is challenging, critical errors can be avoided by optimizing the data write order for each file operation. File block mapping in file systems adopts a hierarchical structure organized by pointer dependencies. In link-type modules, the `inode`'s start block entry points to the FAT block chain, which in turn references data blocks, forming a three-tier hierarchy (inode entry $\rightarrow$ block chain $\rightarrow$ data blocks). In index-type

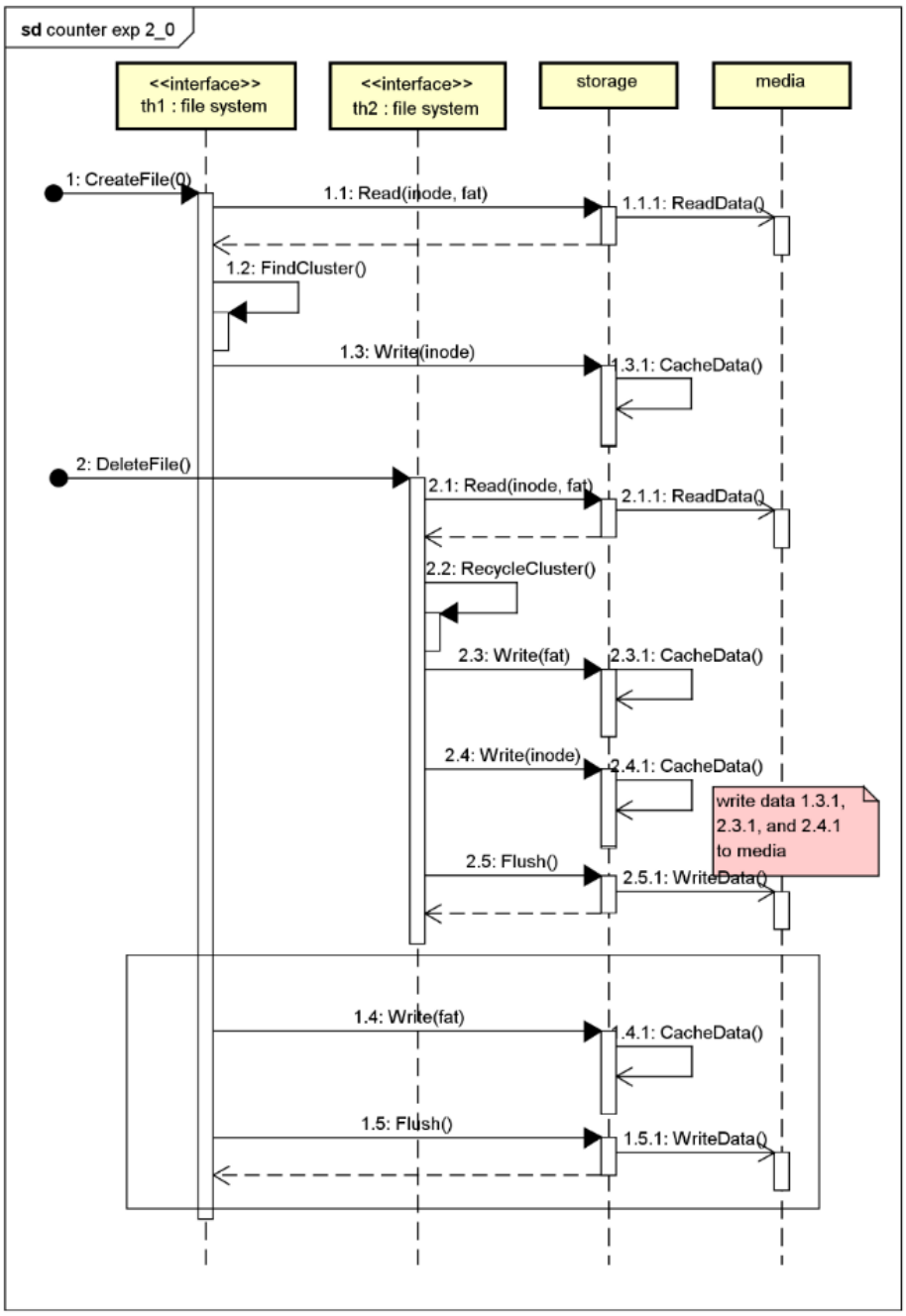


Figure 4.12: Counterexample for the multi-thread mode[1]

modules, the `inode`'s index table entries point to indirect index tables (with potential multi-level nesting) that ultimately reference data blocks, creating a hierarchy of inode index table → indirect index tables → data blocks.

Based on these block-pointer dependencies, we propose a consistency-enhancing update mechanism: when allocating a new block, write the target (pointed-to) block to disk before updating the pointer entry that references it. When reclaiming a block for reuse, first remove the pointer entry referencing the block, then reinitialize or reuse the block on disk. This rule ensures that pointer entries never reference invalid or conflicting blocks, enabling the file system to avoid critical errors even in the presence of unexpected crashes.

We modified both file mapping modules to implement this mechanism: For link-type modules (where the FAT serves dual roles as mapping table and block allocation table):

- File creation: First, mark the free block as allocated in the FAT and write the updated FAT to storage. Then set the `inode`'s start block field to the allocated block and write the updated `inode` to storage.
- File deletion: First clear the `inode`'s start block field and write the updated `inode` to storage. Then, clear entries in the FAT block chain from head to tail and write the updated FAT to storage.

For index-type modules (with block allocation tracked via a bitmap):

- File creation: First, allocate a data block by marking it as used in the bitmap. If an indirect index block is required, allocate and mark it in the bitmap, update its entries to reference the data block, and write the indirect index block to storage. Finally, update the `inode`'s index table entry to reference the indirect index block and write the `inode` to storage.
- File deletion: First clear the `inode`'s index table and write the updated `inode` to storage. Then clear any existing indirect index tables. Finally, mark the associated blocks as free in the bitmap and write the updated bitmap to storage (temporarily storing upper-layer index table information in memory during this process).

Re-verification of both models with the enhanced mechanism first confirmed compliance with all four defined properties. Non-critical dead block errors were detected in both models; disabling abort on dead block errors in the offline checker and re-running verification yielded no errors.

Table 4.5: Verification result with reordering write data[1]

Error mode	Error level	Link type		Index type	
		Single-thread	Multi-thread	Single-thread	Multi-thread
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
Content error	Critical	No error	No error	No error	No error

Final verification covered all four combinations of file mapping type (link/index) and execution mode (single-threaded/multi-threaded). Table 4.5 summarizes these results, confirming the elimination of critical errors. This validates that the proposed mechanism effectively enhances the file system's robustness to crash-induced critical errors.

4.3.4 Reflect on The Real File System

Counterexamples identified in abstract models may be spurious. Given that our models abstract concrete file system elements (operations, workflows, data structures), we can attempt to reproduce these counterexamples in concrete file systems. Reproducible counterexamples indicate real errors; non-reproducible ones are spurious. For the link-type model under multi-threaded conditions, we identified six counterexamples that share the same root cause, all reproducible on a real file system. Correcting the write order in all operations eliminated these counterexamples. Below, we detail the reproduction of a `CreateFile()` counterexample in a real file system and its resolution.

Analysis of the model checker's counterexample trace identified incorrect write order in the `CreateFile()` operation as the root cause. A crash occurring between `inode` update and FAT table update results in inconsistent data, leading to double-reference errors in subsequent `CreateFile()` operations. Since the `CreateFile()` operation in our model was abstracted from the `_create_file()` function (Figure 4.13) of the real-world "Ultra-Embedded FAT 16/32 File IO Library v2.6" [49], we targeted the `fatfs_add_file_entry()` (line 14, Figure 4.13) and `fatfs_fat_purge()` (line 22, Figure 4.13) functions, responsible for `inode` and FAT updates, respectively, injecting a crash between these two functions to reproduce the error.

Error reproduction involved a file creation test: invoking `create_file()` to create a test file in the root directory, writing approximately two sectors of data, and closing the file. Storage simulator logs recorded the write command

```
static FL_FILE* _create_file(const char * filename)
{
//...Sanity check and initialize ...
//...Open the parent directory ...
    // Create the file space for the file (at least one
       cluster worth!)
    file->startcluster=0;
    if (!fatfs_allocate_free_space(&_fs,1,&file->startcluster
       ,1)
    {
        _free_file(file);
        return NULL;
    }
//... file name processing ...
    // Add file to disk
    if (!fatfs_add_file_entry(&_fs, file->parentcluster,(char
       *)file->filename,
        (char*)file->shortfilename,file->startcluster,0,0))
    {   // Delete allocated space
        fatfs_free_cluster_chain(&_fs,file->startcluster);
        _free_file(file);
        return NULL;
    }
//... set general attributes ...
    fatfs_fat_purge(&_fs);
    return file;
}
```

Figure 4.13: Source code of _create_file() in the Ultra-Embedded FAT IO[1]

sequence (Figure 4.14), identifying four write commands `W1` to `W4`:

- `W1` (LBA 32h, root directory location): Saves the file entry to the parent directory (root).
- `W2` (LBA 08h, FAT location): Updates the FAT table.
- `W3` (LBA 52h, file data location): Writes user data.
- `W4` (LBA 32h, root directory location): Rewrites the file entry to update file length.

To simulate crashes after each write command, we constructed a series of disk images: starting with the original `img0`, applying `W1` to generate `img1`, `W2` to `img1` to generate `img2`, and so on to `img4`. Each image represents the disk state immediately after a crash following the corresponding write command.

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

Figure 4.14: Write command log of the wrong order[1]

Running `fsck` on these images revealed a lost block error in `img1`, confirming that a crash between `W1` and `W2` (assuming atomic write commands) triggers a critical consistency error. This error arises because the file entry is persisted in the root directory, but the FAT table remains unupdated, matching the model-identified counterexample.

Resolution required: ensure that link-type file systems first persist block allocation information (FAT table) before updating file entries in the root directory. We added line 12 (Figure 4.15) to force FAT table write-back before file entry storage. This fix minimizes source code modifications relative to reordering operations, with the acceptable side effect of increased data writes during file creation (which is outweighed by the critical error prevention).

Repeating the crash test on the fixed file system yielded the write command log in Figure 4.16, confirming FAT table writing (`W1`, LBA 08h) precedes file entry writing (`W2`, LBA 32h). Since the FAT table remains unchanged after line 11, the `fatfs_fat_purge()` function in line 22 does not perform additional FAT writes. Post-fix testing detected only a non-critical dead-block error in `img1`, with no data loss or other critical errors, consistent with the model verification conclusions.

This case demonstrates the advantage of model checking in detecting corner-case errors that are difficult to identify via traditional software testing. The reproduced error occurs only when a crash occurs between `W1` and `W2` (Figure 4.14), necessitating extensive test-case repetition to detect it using conventional methods.

4.4 Evaluation

We evaluate the verification of the file mapping model from the following aspects: (1) Can it detect crash consistency-related issues, and what is the overhead of identifying these issues? (2) What are the root causes of crash consistency errors, and how significant are the differences between different file systems? (3) What is the coverage rate of the file mapping model verification? (4) How effective are the optimization methods?

```
static FL_FILE* _create_file(const char * filename)
{
//...Sanity check and initialize ...
//...Open the parent directory ...
    // Create the file space for the file (at least one
       cluster worth!)
    file->startcluster=0;
    if (!fatfs_allocate_free_space(&_fs,1,&file->startcluster
       ,1)
    {
         _free_file(file);
         return NULL;
    }
//<FIX> save fat to fix the crash consistency issue
    fatfs_fat_purge(&_fs);
//... file name processing ...
    // Add file to disk
    if (!fatfs_add_file_entry(&_fs, file->parentcluster,(char
       *)file->filename,
         (char*)file->shortfilename,file->startcluster,0,0))
    {   // Delete allocated space
         fatfs_free_cluster_chain(&_fs,file->startcluster);
         _free_file(file);
         return NULL;
    }
//... set general attributes ...
    fatfs_fat_purge(&_fs);
    return file;
}
```

Figure 4.15: The fixed _create_file() in the Ultra-Embedded FAT IO[1]

```
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
```

Figure 4.16: Write command log of the correct order[1]

Table 4.6: Summary of the verification result[1]

Model		Depth	State number	Transition number	Mem (GB)	Time (hrs)
Link Type	single-thread	763 408	0.75×10^{10}	1.71×10^{10}	4.67	0.82
	multi-thread	2 652 768	1.20×10^{10}	3.83×10^{10}	4.73	1.81
Index Type	single-thread	3449	0.56×10^{10}	1.15×10^{10}	4.64	0.60
	multi-thread	37 686	1.10×10^{10}	2.86×10^{10}	4.65	1.66

4.4.1 Experiment Execution

We verified the link-type and index-type file system models on our desktop PC, with an Intel Core i7-8700K CPU, 16 GB of memory, and 64-bit Windows 10. Each model was tested in two configurations: single-thread and multi-thread modes. Each execution consumed approximately 4.7 GB of memory and took 4.9 hours. Table 4.6 presents the verification time for each mode.

Crash consistency issues were detected in both file system models. Notably, the occurrence of such issues is sensitive to the timing of crashes; problems are triggered only if a crash occurs between two specific disk operation commands. We argue that conventional black-box testing is unlikely to identify such low-probability events.

4.4.2 Root Causes of Crash Consistency Errors

The fundamental cause of crash consistency errors lies in the inconsistency between file index metadata and data block allocation metadata. For link-type file systems, the file index metadata refers to the first block of the file stored in the `inode`, while the data allocation metadata corresponds to the FAT table. For index-type file systems, the file index metadata is the file index table within the `inode`, and the data allocation metadata is the block bitmap. Since these metadata are non-contiguously stored on physical storage, they cannot be updated atomically. A power outage or system crash during metadata updates will inevitably result in metadata inconsistencies. We confirmed this root cause by simulating atomic metadata writes in single-thread mode using the storage cache. Our experiments further demonstrate that, without the support of additional data structures (e.g., journals) or algorithms, it is impossible to guarantee metadata consistency in the presence of crashes. However, critical errors or data loss can be avoided by adjusting the order of metadata writes.

From the above analysis, there is no significant difference in crash consis-

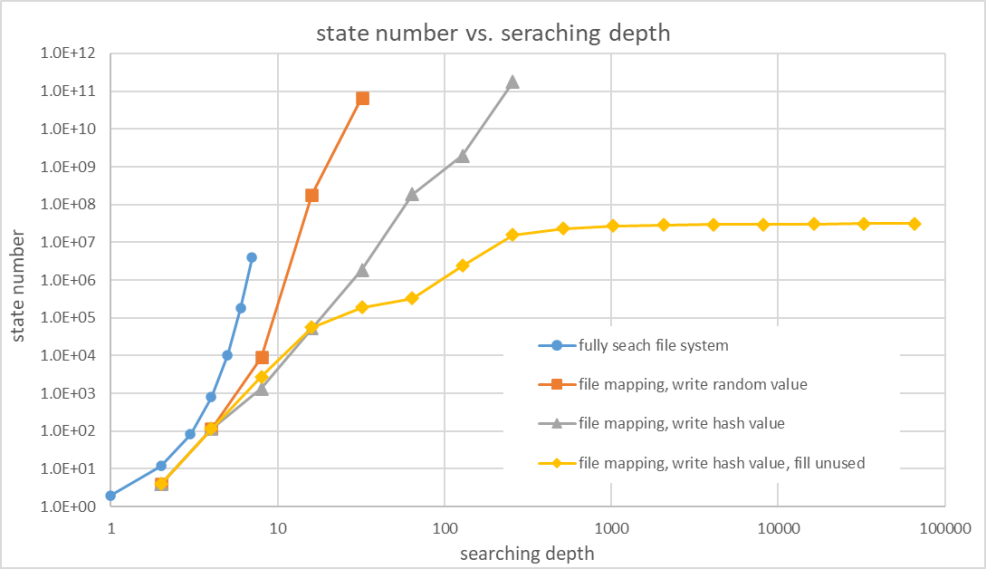


Figure 4.17: Number of states in searching the file system by different models[1]

tency robustness between link-type and index-type file systems. Nevertheless, recent index-type file systems, such as ext3 and NTFS, have improved crash consistency fault tolerance by introducing journaling mechanisms; however, this aspect falls outside the scope of our research.

4.4.3 Effect of Optimization

We implemented two optimization strategies for the file mapping models: (1) utilizing hash values to populate file data instead of random values, and (2) filling invalid blocks with a fixed value. These strategies remarkably reduce the number of model states. Figure 4.17 illustrates the relationship between the number of states and search depth during the verification of file mapping models, with both coordinates presented on a logarithmic scale. The blue line denotes the number of states at each search depth for the full FAT model, serving as a reference benchmark.

The orange line shows the state count during the file-mapping model search without optimization. The gray line shows the state count when the same file-mapping module is searched using the first optimization (i.e., using hash values to fill the file data). This optimization mitigates the growth rate of state counts, resulting in near-linear (index-based) growth. However, comprehensive state-space exploration of the model remains infeasible. The yellow line shows the state count when both optimizations are enabled (i.e.,

the second optimization, which fills invalid blocks with a fixed value). When the search depth exceeds 1000, the number of states reaches saturation, indicating that the model's entire state space has been fully explored. The abstraction of the file mapping model from real-world file systems, combined with these two optimization strategies, enables comprehensive file system verification within an acceptable time frame.

4.4.4 Model Checking Coverage Rate

As shown in the figure, after adopting two optimization methods, hash file padding and invalid block padding, the number of states searched no longer increases with increasing search depth, indicating that state-space saturation has been achieved. Our search thus covers the entire valid state space of the file mapping model.

4.5 Discussion

In this study, we proposed an approach that converts the file system operation flowchart into state machines to create abstract file system models in Promela. We designed two minimal file system models: one link-type, abstracted from the FAT file system, and one index-type, from the ext2 file system. We optimized the two file-mapping models, enabling a comprehensive evaluation of their performance. We also designed four properties to indicate the file system's correctness. Subsequently, we used SPIN model checker to comprehensively evaluate the robustness of both models, including under crash conditions.

By applying model checking to these models, we identified critical errors that led to an unexpected crash. We also provided a mechanism to improve the robustness of file systems. By comparing results with and without caching in single-file mode, we confirmed that the inability to perform atomic updates to multiple metadata entries is the root cause of file system crash consistency violations. However, when multiple files are opened simultaneously, caching cannot update each file's metadata atomically. Since it is impossible to ensure the consistency of all metadata in the block file system without resorting to other methods, such as checkpoints or journals, we have reclassified the repairable property, dead block, as non-critical. Ultimately, by adjusting the file system's metadata write order and repairing non-critical errors, we have ensured crash consistency. Finally, we verified that the improved file systems contained no errors.

In this study, we summarized the advantages and limitations of using the SPIN model checker for file systems.

- Comprehensively evaluated the file system's crash consistency. The term "comprehensively" indicates that our verification encompasses both link and index file systems, both single-thread and multi-thread modes, and multiple properties. According to this study, we also compare the robustness of different types of file systems, and different modes (Table 4.4 and Table 4.5)

- Exhaustively evaluated the file system's robustness. Our evaluation covered all possible workloads, including crashes that occurred under any conditions. By optimizing the file system model, we searched the file system state space using the SPIN model checker, including all crash conditions. The exhaustive checking helped us identify corner-case errors that are difficult to detect with black-box testing.

- The model checking approach helps us analyze counterexamples. We can recall corner-case errors in the real file system and correct them. By analyzing the corner-case errors, we understood the mechanism of crash-consistency and improved the file system's robustness. The improvement mechanism applies to most block file systems and does not require additional approaches such as checkpointing or journaling.

However, our study was subjected to the following threats to validity.

- This model presents only a minimal file-mapping model. It does not include full file system features, such as path mapping or directory structure. Checking the full file system model requires massive resources.

- The process of abstracting a real file system into a Promela model is inefficient. We manually abstract the file system into a Promela model, which is time-consuming and error-prone. Such an approach is difficult to apply to a complex file system, including the file-mapping component, such as a log-structured file system.

In conclusion, while SPIN is a powerful tool for formally verifying core file system algorithms and detecting deep corner-case bugs in file system consistency, its practical application is constrained by the state-explosion problem and the significant manual effort required to create an accurate, simplified model. The chapter demonstrates that it is best suited to verifying critical core components, such as file mapping. In the next chapter, we present our research on modeling and verification of a full log-structured file system.

Chapter 5

Model Checking of Full-Stack File System

In Chapter 4, we proposed a Promela-based file mapping model and leveraged SPIN for crash consistency verification. While this approach validated the robustness of link-type and index-type file systems across diverse configurations, it was limited to the minimal file mapping layer, excluding complex functionalities such as directory tree management, journaling, and garbage collection (GC) in Log-Structured File Systems (LFS). These limitations restricted its applicability to real-world full-stack file systems, which integrate multi-layered architectures and intricate algorithms.

To address this research gap, we extend model checking to full-stack file systems, thereby enabling verification of end-to-end functionality (e.g., dynamic file/directory management in the path-mapping layer) and advanced mechanisms (e.g., LFS GC). However, scaling to full-stack models introduces three critical challenges: (1) Promela's expressiveness is insufficient to model complex file system logic, leading to manual modeling errors; (2) generic state handling in SPIN fails to mitigate state explosion for large-scale models; (3) lack of compatibility with real-world file system implementations (typically in C/C++) hinders practical adoption.

This chapter presents MC^3 (Model Checking for Crash Consistency), a dedicated tool to overcome these challenges. We first analyze the core obstacles in full-stack file system modeling and verification, and then detail MC^3's design, including a C/C++-compatible model interface, an efficient state-space search, and crash-simulation mechanisms. Finally, we validate MC^3 on the F2FS file system and uncover a fundamental crash consistency flaw inherent to LFS.

5.1 Full-Stack File System Checking

5.1.1 Overview

Full-stack file system verification demands solutions to three interconnected challenges, which directly inform MC^3's design philosophy:

- **Model Compatibility and Simplicity**: How to align the model with real-world file system implementations (C/C++) while simplifying state replication and reducing redundant components?
- **Efficient State Space Exploration**: How to traverse the exponentially growing state space of full-stack models without sacrificing coverage or performance?
- **Faithful Crash Simulation**: How to accurately simulate crash scenarios (e.g., power outages mid-write) across multi-layered file system operations?

To address these, MC^3 is designed as a modular, C++ implemented tool with three core innovations, as outlined below:

1. C/C++ Native Model Interface. MC^3 adopts a POSIX-compliant unified interface for file system models, leveraging C/C++ to maximize compatibility with real-world file system code (enabling partial code reuse). Key optimizations include:
    - Simplifying non-essential features (e.g., symbolic links, timestamps, and implicit read/write offsets) to focus on crash consistency-critical logic;
    - Introducing verification-specific helper functions (e.g., `clone()`, `reset()`) to support state replication and crash simulation;
    - Using pre-allocated object arrays (instead of dynamic memory) for core structures (inodes, dentries, pages) enables efficient state duplication during search.
2. Scalable State Space Search. To balance coverage and efficiency, MC^3 integrates:
    - **A multi-threaded Depth-First Search (DFS)** algorithm, with thread-local `OpenList` to minimize synchronization overhead;
    - An **on-the-fly workload generator** extended for full-stack models, which dynamically generates feasible operations (e.g., `mkdir`, `move`) based on the current file system state;

- A **hash-based state encoding scheme** (derived from the optimized AHU algorithm) to eliminate redundant states from isomorphic directory structures and identical operation outcomes.

3. Differential Crash Simulation. MC^3 implements a command rollback mechanism via a differential storage model:

   - Stable states are defined as the file system state post-operation execution; intermediate states are generated by replaying IOs between consecutive stable states.
   - For general file systems, all 2^n combinations of IO reordering are simulated (accounting for OS-level IO scheduling); for LFS, sequential write guarantees reduce this to n ordered replays.
   - Crash scenarios are triggered via the `reset()` interface, which aborts in-progress operations and reloads the file system from the simulated post-crash disk image for consistency checks.

In the following sections, we detail the design of MC^3's core modules (full-stack model interface, state search, state encoding, and crash simulation), followed by experimental validation on F2FS.

5.1.2 Full-Stack File System Model

Our goal is to simplify the design of the file system model under verification. Since most file systems are implemented primarily in C or C++, our model is also developed in C or C++. This design choice brings the model closer to real-world file systems, even allowing partial reuse of existing code, and facilitates the implementation of complex functionalities such as garbage collection. Additionally, we modularize the file system model under verification and provide a unified interface, enabling verification of any model that conforms to it.

The interface of the file system model comprises three parts: The first part consists of POSIX-based file system APIs, through which the file system checker drives the model's operations. To streamline the model, we introduce three key simplifications: (1) File system objects only support files and directories, excluding symbolic links, hard links, and other similar types; (2) Auxiliary file system features are omitted, including timestamps, permission management, disk quotas, and content compression; (3) Certain functionalities of the Linux Virtual File System (VFS) are simplified. For instance, the Linux VFS maintains a read/write offset for each opened file; read/write operations begin at this offset, requiring users to specify only the size of the

data block to be read/written; the offset is then updated automatically. To read/write from other positions, users must call the `fseek()` function to adjust the offset. In our model, we eliminate the need to maintain this implicit offset; instead, users are required to provide an explicit offset parameter when invoking the `read()` / `write()` functions, thereby obviating the need for the `fseek()` function.

In this study, we assume that the `read()` operation does not modify any storage data in the file system. Specifically, `read()` is invoked only to verify the correctness of file data and not participate in the exploration of file system states. For file systems that perform disk-writing operations during `read()` calls, such as appending access logs or modifying file timestamps, we can emulate these write behaviors via dedicated `write()` operations in our model.

The second part of the interface provides support for file system verification, including core functions such as `initialize()`, `clone()`, and `reset()`: The `initialize()` function initializes the model state (e.g., creating the initial root directory), analogous to the `mkfs()` utility for file systems; The `clone()` function enables self-replication of the file system during state space search; The `reset()` function is used to simulate crash occurrences; The `get_fs_info()` and `get_file_info()` functions retrieve file system and file-related metadata (e.g., the number of directories, file size) without opening files, supporting property checking and debugging. Table 5.1 illustrates the interfaces supported by the file system model.

In addition to following the interface, the file system model must be duplicable. During file system checking, we must search all file system states and roll back to previous states, using either a depth-first or a breadth-first search. This requires copying and backing up the file system state during the search process. However, actual file systems extensively use dynamic memory allocation and pointer references. For example, Linux file systems frequently use dynamic allocation to obtain inode objects or allocate cache pages. These dynamically allocated objects are non-contiguous in memory. When attempting to copy the file system state, we cannot simply copy the memory that contains it. We need to retrieve all dynamically allocated objects individually, copy them, and relink them. This makes state copying very inefficient.

In our model, we manage dynamic objects using pre-allocated arrays, called object allocators, which include inodes, index nodes, pages, and dentries. The memory for these data structures, as part of the file system state, is allocated as fixed-size arrays during initialization. When the file system needs to allocate objects of these data structures, the object-allocator allocates one unit from the array and returns its array index. When referencing

Table 5.1: Interface of file system model

Category	Method	Description
POSXI	mount	mount the file system
	unmount	safely unmount the file system
	fsck	check the file system consistency and recover
	create	create a file
	mkdir	create a directory
	open	open a file
	write	write data to an open file
	read	read data from an open file
	truncate	remove data from file, and shrink file size
	remove	remove a file
	rmdir	remove an empty directory
	move	move or rename a file or directory
	fsync	persist a file and its data
	sync	persist the file system
checking help	initialize	create a file system and initialize it
	reset	crash simulation, reset a file system's in-memory data without persist
	clone	copy a file system in memory data to another instance
	get_fs_info	retrieve file system information, for debug
	get_file_info	retrieve a file's information, like length, etc

these data structures in the file system, the indexes are used. The object allocator converts indexes to object pointers only in local functions when the file system needs to read or write fields in the object. Object pointers are not stored in the file system state. When the checking program copies the file system state, the object arrays are included in the copy. References to specific objects in the file system refer to the object array indices and therefore don't need to be changed. After this optimization, the file system model becomes state-duplicable, enabling search over the file system's state space.

5.1.3 Search Full-Stack File System State

To verify full-stack file systems, we continue to use state-space search. The on-the-fly trace generator generates states after each file system operation. We then replay the write I/O for each operation, create an intermediate state to simulate crash scenarios, and check properties during the search to verify file system correctness, including crash consistency.

We define the file system's state as stable after each operation, and the state search engine is responsible only for exploring stable states. Between two consecutive stable states, S_1 and S_3, the file system sends a set of write commands $W_1, W_2, ..., W_n$ to persist data. Each write operation transitions the file system into intermediate states, such as $S_1 \rightharpoonup S'_{31} \rightharpoonup ...S'_{3n-1} \rightharpoonup S_3$; only after all writes are completed does the file system enter the next stable state S_3.

Figure 5.1 demonstrates the workflow of full-stack file system checking. We designed two state sets for the search: `OpenList` to store states currently under exploration, and `VisitedTable` to store states that have already been visited to avoid redundant visits. The search engine retrieves a file system state S_1 from the `OpenList`. The on-the-fly workload generator analyzes the existing objects in the file system S_1, such as the root directory and a subdirectory `"/A"`, to generate all possible operations, such as creating a subdirectory `"/B"` in the root or a file `"/A/e"` in directory `"/A"`. The search engine applies these operations to the file system, producing new states (S_2, S_3, ...) and then checks whether these states satisfy the file system properties. We then check whether the state is already in the `VisitedTable`. If it is a new state, the search engine enqueues it to the `OpenList` and adds it to `VisitedTable`.

For each operation, we record all write I/O commands and then create intermediate states (S_{31}, S_{32}, ...) by replaying them. We use the file system model to recover from these states and to check for any crash consistency errors. After recovering, these intermediate states are enqueued into the `OpenList` and added to the `VisitedTable` for further investigation. If an er-

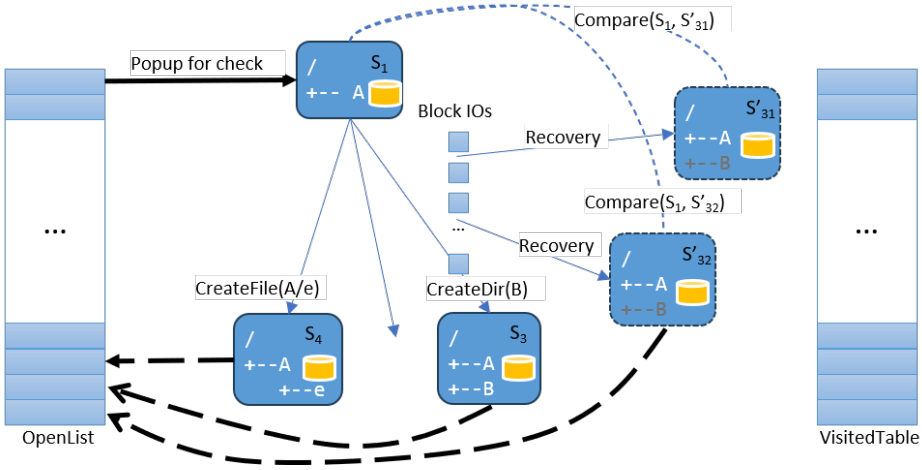


Figure 5.1: Workflow of On-The-Fly File System Checking

ror is detected, the trace from the initial state to the current state is recorded as a counterexample.

Crash Simulation

A critical step in crash consistency verification is the faithful simulation of crash scenarios that may occur between stable states. To this end, we construct a post-crash disk image of all intermediate states by replaying the sequence of IOs $(W_1, W_2, ..., W_n)$ between S_1 and S_3. Each replay step stops after a special number of IOs (e.g., stopping after W_1 to simulate a crash right after the first write) and generates a corresponding post-crash disk image $(S_{31}, S_{32}, ...S_{3n})$. We then reload the file system model from each of these images, trigger the file system's built-in recovery logic (`fsck`), and check for consistency violations.

The number of crash states to simulate varies significantly between conventional block-based file systems and log-structured file systems (LFS) such as F2FS, depending on the file system's write semantics. In conventional file systems, the operating system's I/O scheduler may reorder the execution sequence of write commands $W_1, W_2, \ldots, W_n$ to optimize disk access efficiency. Since a crash can occur at any point during these reordered writes, we must simulate all 2^n possible subsets of executed IOs to cover every potential crash scenario.

For LFS, the core design principle is sequential, out-of-place writes; all new data is written to contiguous, unused disk segments in the order of the write commands, rather than overwriting existing blocks. This design inherently eliminates IO reordering for data persistence; write commands

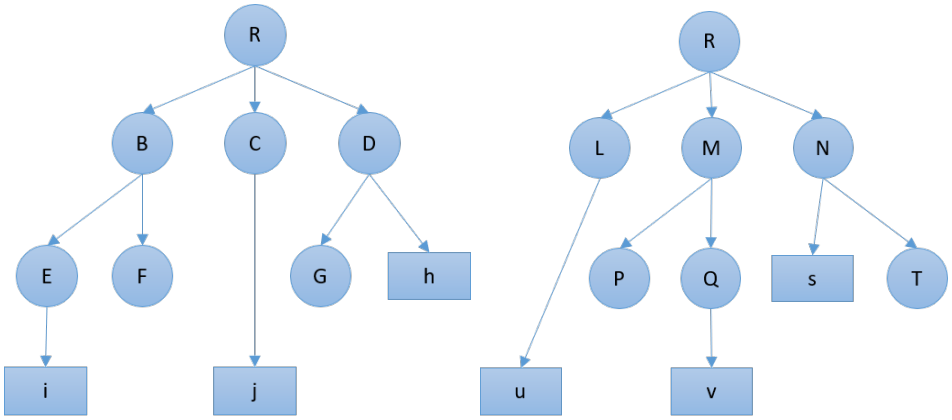
$W_1, W_2, ..., W_n$ are guaranteed to be executed and persisted to disk in the exact order they are issued. Thus, we only need to replay the write sequence in order and simulate n crash states.

Property Checking

During the state space search and crash recovery checks, our verification covers three core dimensions to ensure comprehensive crash consistency:

- Correctness of file data (e.g., whether written data is intact after recovery).
- Validity of file operations (e.g., whether a `create` operation after recovery returns expected data).
- Consistency of metadata (e.g., consistency between inodes, block bitmaps, and directory entries; validity of pointer references in index tables).

Metadata consistency checking is an indispensable component of file system crash consistency verification, for three key reasons: (1) Certain file system errors do not cause immediate data loss but induce latent side effects that degrade system usability over time. A typical example is the dead block issue discussed in the file mapping layer. Although this anomaly does not directly cause data corruption, it gradually consumes the file system's free space, ultimately rendering the file system unavailable. (2) Detecting metadata errors enables the early identification of potential data loss risks before irreversible damage occurs. Consider a scenario where File A references an unallocated data block; this inconsistency does not immediately result in data loss. However, once the same block is subsequently allocated to File B, any write operation on File B will overwrite the block, thereby corrupting File A's data. Metadata checking can detect the initial invalid block reference, preventing subsequent data loss. (3) Furthermore, diagnosing the root cause of metadata errors is significantly easier than troubleshooting post-facto data loss issues. Metadata inconsistencies leave explicit traces (e.g., mismatched block pointers, invalid bitmap entries) that directly point to the source of the problem. In contrast, data loss errors often present as vague symptoms with obscure origins. Accordingly, in our crash consistency verification framework, we not only validate the integrity of file data and the functionality of file system operations but also rigorously verify metadata consistency immediately after crash recovery.



(a) One of the two isomorphic directory structures

(b) Another directory structure, which is isomorphic with (a)

Figure 5.2: Isomorphic directory structure.

5.1.4 Encode File System State

In the `VisitedTable`, we use a Hash Table to record already visited states. To implement hashing, we first encode the file system state. Traditional model-checking tools, such as SPIN, consider different models' binary values as different states. In the file mapping model, all variables, including storage data, are treated as a state. We used two optimizations to reduce the state number of the file mapping model. However, the full-stack file system model is more complex. It consists of directories and files. Some directory tree structures appear different but are identical; we call them isomorphic. Such states include, but are not limited to, having the same number of files and directories under a directory with different filenames, or having two subdirectories with different sorting orders within their parent directory due to different directory names. Figure 5.2 (a) and (b) show two isomorphic directory tree structures. Circular with uppercase letters represent directories, rectangular with lowercase letters represent files. Directory D in figure (a) and directory N in figure (b) are isomorphic subdirectories, both containing one subdirectory and one file, despite having different names. Subdirectories B, C, D in figure (a) are respectively isomorphic to subdirectories L, M, N in figure (b). Although their sorting order within the root directory differs, their root directory R remains isomorphic. In our model checking, we need to exclude such identical directory structures to avoid redundant checking. The table shows the impact on the search space when identical structures are excluded versus included.

Our goal for file system state encoding is to fully reflect isomorphic di-

rectory tree structures while minimizing redundant states during file system state search. To achieve this, we propose that the encoding of a file system state consists primarily of two components: the file system content and the operation trace.

The content of the file system primarily comprises the directories and files currently present in the system, as well as the contents of these files. According to Wolper's Data Independence Theorem [50], we do not care about the specific data of files, only their size (measured in block units). Similarly, for the directory tree, or the path-mapping layer, we focus only on its structure, for example, the number of subdirectories and files under a directory, rather than the specific names of these subdirectories or files. We designed an encoding scheme for the file system construct based on the AHU algorithm [51], optimizing for the characteristics of file system directory trees.

The AHU algorithm determines the isomorphism of rooted unordered trees, i.e., the order of the children is unimportant; as long as the parent-child relationships in the tree structure are the same, the trees are considered identical. The core idea of the AHU algorithm is to assign a unique code to each node of the tree bottom-up and to determine whether two trees are isomorphic by comparing the codes of their root nodes. The AHU algorithm has been proven to guarantee completeness and reliability: isomorphic trees necessarily produce the same code, and trees with the same code are necessarily isomorphic [52].

Our improvements to the AHU algorithm mainly include three aspects: (1) Distinguishing between directory nodes and file nodes. In the original AHU algorithm, all nodes are treated as equivalent; in particular, all leaf nodes are considered identical. However, in file systems, leaf nodes can be either directories or files, and these two structures must be differentiated in file system encoding. (2) Adding file size encoding for files to distinguish the results of file write operations. (3) Handling file system operations that do not affect the directory structure but impact subsequent operations. For example, the `Open()` operation: a file must be opened before it can be read or written, but opening a file does not alter the file system structure. We therefore add a flag to track open files. This scheme still ensures that identical directory tree structures yield the same encoding, while different structures produce distinct encodings.

However, considering only the file system's directory structure is insufficient. In our practical tests, we found that different operation sequences can yield the same directory structure yet affect the results of crash consistency checks. Two different sequences of write and create operations may generate identical directory trees but result in different metadata states, which

in turn lead to distinct crash behaviors. For instance, in Figure 5.4, we can reach state S_4 through the sequence `CreateDir(A), CreateFile(b)`, or reach state S_8 through the sequence `CreateFile(a), CreateDir(D)`, where S_4 and S_8 contain identical structures. However, in crash consistency testing, different operation sequences yield different results, particularly for the final few operations before the crash. Therefore, in addition to the file system structure encoding, we append an operation-path encoding. Thus, as part of the file system state encoding, we append the encoding of the operation sequence to the directory structure encoding.

5.2 MC^3: Model Checking on File System Crash Consistency

In the previous section, we proposed the Full-Stack File System Checking approach. In this section, we design the MC^3 (Model Checking on Crash Consistency) [1] tool based on this approach to verify the F2FS file system.

MC^3 comprises five core components: the State Search Engine, Workload Generator, File System Model, Storage Model, and Checker. Figure 5.3 illustrates the architecture of our file system verification tool.

The State Search Engine serves as the core verification module, employing a depth-first search (DFS) to exhaustively explore the file system model and verify whether each state satisfies predefined requirements. The Workload Generator produces a suite of file system operations to fully exercise the file system, enabling it to traverse all potential states. It incorporates crash-condition generation logic to evaluate the file system's crash consistency.

Serving as the test subject, the File System Model implements a standardized interface; this design enables diverse file systems to be modeled and tested by adhering to the interface. The Storage Model manages the persistence of file system data and supports command rollback, which facilitates the enumeration of power outages between any two consecutive stable states.

Given that different file systems exhibit distinct properties, we designed a dedicated verification module (Checker) to validate file system properties, with particular emphasis on the correctness of post-crash recovery. The Checker is typically designed with reference to the file system's native `fsck` utility.

[1] https://github.com/jcyuan78/model_checking_for_crash_consistency

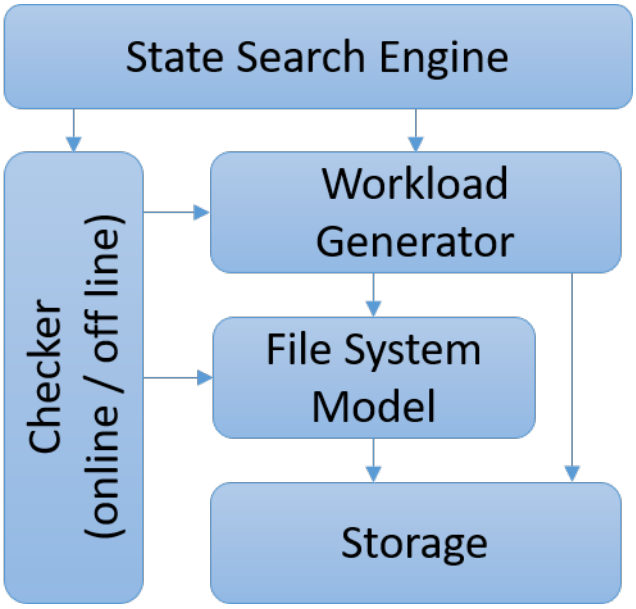


Figure 5.3: Block diagram of MC^3 tool

5.2.1 State Search Engine

The State Search Engine implements the full-stack file system search approach proposed in Section 5.1.3 using a depth-first search (DFS) algorithm. To avoid stack overflow and reduce function-call overhead, we adopt an iterative implementation rather than recursion, treating `OpenList` as a stack and using a First-In-Last-Out (FILO) order for pushing and popping states to implement DFS traversal.

The search algorithm is outlined in Algorithm 1. We use `OpenList` to store pending exploration states and `VisitedTable` to track all previously verified states. The process initiates by pushing the initial file system state into `OpenList`. We then repeatedly retrieve and examine the state at the top of `OpenList`, popping it once exploration is complete; this cycle continues until `OpenList` is empty.

During state exploration, we invoke the Workload Generator to generate a complete set of feasible operations, derived from the current directories and files in the file system. The file system sequentially executes each generated operation, yielding a new stable state with each execution. Concurrently, crash states are generated during the transition from the prior stable state to the new stable state. We then validate whether each new state (and its intermediate crash states) complies with the predefined file system properties. If a property violation is detected, a counterexample is logged, tracing the

path from the initial state S_0 to the problematic state S'. For states with no property violations, we check if the new state has been previously visited. If the state is newly identified, or if a shorter path to an already visited state is discovered, it is pushed to the top of `OpenList` for subsequent exploration. The mechanism for determining whether a file system state has been visited is described in the "File System Encoding" section.

Algorithm 1 Depth-first searching file system state space

```
1: OpenList.Initialize()
2: OpenList.Push(S_0)
3: VisitedTable.Initialize()
4: while OpenList is not empty do
5:     S ← OpenList.PopUp()
6:     for all op in WorkloadGenerator(S) do
7:         S' ← S.Invoke(op)
8:         if S' not in VisitedTable or S'.depth is smaller then
9:             if error in check(S') then
10:                report error;
11:                record trace from S_0 to S'
12:            end if
13:            OpenList.push(S')
14:            VisitedTable.Insert(S')
15:        end if
16:    end for
17: end while
```

To improve search speed, we implemented multithreaded parallel processing for the search expansion component, the `for all` block. When the workload generator creates the operation list for the current state, the state search engine duplicates the current state. It assigns each operation to a separate thread for invoking it. Then the state search engine waits for all operations to be complete and inserts the result status into the open list.

To reduce synchronization overhead among multithreaded tasks, we introduce several modifications to the DFS algorithm. Three shared variables are used within the `for all` block: the state S to be expanded, `OpenList` and `VisitedTable`Ṡtate S is read-only during the expansion phase and thus requires no special handling.

For `OpenList` we allocate a thread-local `OpenList`[i] for each thread. After a thread verifies state S', it pushes S' into its local `OpenList`[i] since only thread-local variables are involved, no inter-thread conflicts occur. At Line 5, when popping a state for expansion, we pop the state from the longest

`OpenList`[i] among all thread-local lists. This pop operation is executed in a single-threaded context, and as multi-threaded processing has already completed at this stage, reading `OpenList`[i] will not cause conflicts.

For `VisitedTable` shared access among multiple threads is unavoidable. We therefore apply the Slim Reader/Writer Lock (SRWLOCK) to implement separate read/write locking for `VisitedTable` which reduces locking overhead. The modified parallel DFS algorithm is presented in Algorithm 2. We will report the performance of the parallel DFS algorithm in the Evaluation section.

Algorithm 2 Parallel Depth-first searching file system state space

```
1: OpenList[].Initialize()
2: OpenList[0].Push(S_0)
3: VisitedTable.Initialize()
4: while OpenList is not empty do
5:     i ← Max(OpenList[].Length)
6:     S ← OpenList[i].PopUp()
7:     for all op in WorkloadGenerator(S) do Parallel
8:         S' ← S.Invoke(op)
9:         if S' not in VisitedTable or S'.depth is smaller then
10:            if error in check(S') then
11:                report error;
12:                record trace from S_0 to S'
13:            end if
14:            OpenList[thead_id].push(S')
15:            VisitedTable.Insert(S')
16:        end if
17:    end for
18:    Wait for all threads to complete
19: end while
```

Parallel processing can appropriately enhance search speed. Figure 5.10 shows the time required to scan the entire file system at a search depth of 11 layers under different degrees of parallelism. When the degree of parallelism increases from 1 to 8, the search time decreases noticeably. Further increasing the degree of parallelism yields only a small additional reduction in search time. The main reason is that parallel search is only used when expanding a particular state. If the number of branches (possible operations) for expanding a state is less than the number of parallel threads, the full potential of parallel computing cannot be realized. Even when the number of branches exceeds the number of threads, some waiting time is introduced.

Another issue with the parallel algorithm is that it can lead to instability in search results. Since each branch completes at a different time, the order in which results are returned has a degree of randomness. This randomness causes variations in the order in which states are placed into the open list. The change in order may result in slightly different search paths each time. However, no impact on the inspection results has been observed so far.

5.2.2 Workload Generator

MC^3's Object-Based Workload Generator extends the On-the-Fly Workload Generator in the File Mapping Checker. During state-space exploration of the file system, it incrementally generates new operations, enabling exhaustive traversal of the entire file system state space. The key extensions of this Object-Based Workload Generator target the newly added path mapping layer and corresponding operations for Full-Stack file systems, primarily involving the following aspects:

1. File System Object Expansion.

   Full-Stack file system objects include not only files but also directories and the file system itself. For files, different operations are allowed depending on their state (open or closed). For directories, new operations such as `mkdir`, `rmdir`, and `move` are introduced. For the file system itself, operations like `sync`, `unmount/mount`, and `crash/recovery` are supported. The `crash/recovery` operation is designed to simulate file system crashes: it resets the file system under test, then instructs the Storage Model to replay a specified number of IOs from the previous stable state to simulate a post-crash disk image. The file system then attempts to recover from this image, followed by error-checking for inconsistencies. Table 5.2 demonstrates all possible operations for each object. We do not list the read operation for a file because it does not change the file system state. Additionally, we read all files during the verification process.

2. Operation Parameter Design.

   The core parameters of the operations include: the name of the target directory/file, the offset and length for `write` operations, and the number of IOs for `crash/recovery` operations. Details of key parameters are as follows: For `create` or `mkdir` operations: A selected directory is used as the target, and a new file or subdirectory (i.e., the target object) is created under it. Target names are randomly generated, specifically as 1 8-character strings in the F2FS model. For `write` operations: Two

additional parameters (offset and data length) are introduced. The operation uses a selected file as the target, writing data of the specified length starting at the specified offset. If the offset is less than the file's current size, partial overwriting occurs; otherwise, data is appended to the end of the file.

3. Parameter Optimization for State Space Control

   Covering all possible combinations of offsets and write lengths would yield an excessively large number of substates, thereby degrading the efficiency of other operations. For each file, the number of potential `write` parameter combinations is approximately $O(m \times n)$, where m denotes the file's current length (in blocks) and n represents either the number of free disk blocks or the file's maximum allowed length. To mitigate state explosion, we randomly sample the offset and write length for each `write` operation. Specifically, only one `write` operation is generated per file in a given state.

4. Parameter Handling for `crash/recovery`

   The `crash/recovery` parameter specifies the number of IOs to roll back. MC^3 iterates over all valid IO counts, ranging from 1 to the total number of IOs written by the previous operation.

Table 5.2: Object-based File System Operations

Objects	**Operations**
File System	sync, unmount/mount, crash/recovery
Directory	mkdir, create, move, rmdir
Closed File	open, move, rm
Opened File	write, close, fsync, truncate

Unlike ACE, object-based load generation does not require establishing explicit dependencies because targets are generated only when they already exist. For example, if we need to create a file named *Foo* in the folder *Bar*, ACE must analyze the dependency conditions for this operation, namely, that the directory *Bar* must exist beforehand. ACE will then create the dependency by creating the directory *Bar*. For MC^3, because we use object-based creation, we create the file under *Bar* only if the directory *Bar* already exists. Additionally, if the directory *Bar* exists, we iterate through all possible operations, including file creation. MC^3 does not suffer from omissions.

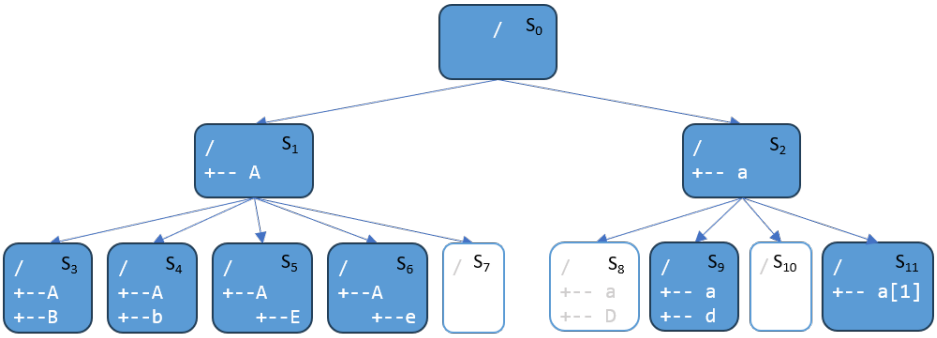


Figure 5.4: Object-based workload generation

Because of the object-based approach, parameter errors are largely avoided. Some exceptions include file name conflicts. Because the target names for creating files or directories are generated randomly, it is possible to encounter a situation in which a file already exists. In such cases, we simply regenerate a random file name.

Testing begins with an initial file system state, a freshly created or formatted system containing only a root directory. The workload generator identifies all possible operations for each object in the current state. It then executes these operations sequentially, with each operation leading to a new state. These new states are duplicated and saved as successors to the original state. This process continues until all file system states have been examined.

Figure 5.4 illustrates the workload-generating process. The initial state S_0 contains only the root directory. From this state, we can either create a subdirectory or a file, yielding states S_1 and S_2, respectively. S_1 contains directory A in the root directory, while S_2 contains file a in the root directory. From S_1, we can perform five operations: creating a subdirectory or file under the root directory, creating a subdirectory or file under directory A, or deleting directory A. These operations generate five successor states. However, when we delete directory A (resulting in S_7), we're left with only the root directory, identical to S_0. We consider S_7 a duplicate of S_0 and remove it from further checking. From state S_2, we can create a subdirectory or file in the root directory, delete file a, or write data to file a. These operations generate four successor states, two of which are duplicates.

Limitation The object-based workload generation method has certain limitations; it cannot generate non-compliant workloads. For example, the generator cannot generate a workload that deletes a nonexistent file or creates a file in a nonexistent directory. It cannot generate workloads similar to those used to test whether the file system returns the correct error codes under such

illegal operations. However, this does not affect crash consistency checking of the file system. Typically, these illegal operations do not cause side effects in the file system and do not write data to storage; they do not alter the file system state and do not affect the file system's crash consistency. On the other hand, this limitation simplifies the verification of file system correctness, which we elaborate on in Section 5.2.5.

5.2.3 File System Encoding

The encoding of a file system state consists of two components: directory tree encoding and operation trace encoding. The directory tree encoding is extended from the AHU algorithm. In contrast, the operation trace encoding is designed to address the issue that different operation traces can affect the results of crash consistency checks. We will elaborate on the implementation and optimization strategies for these two encodings separately in the following.

Directory Tree Encoding Since the directory tree structure of a file system is essentially a rooted, unordered tree, we adopt the AHU algorithm as the core for determining the isomorphism of its static structure. However, compared with a general rooted unordered tree, the directory tree structure of a file system has characteristics that necessitate improvements to the AHU algorithm.

Moreover, encoding only the file system's static structure is insufficient to distinguish among different file system states. For example, opening a file does not change the static structure of the file system. If only the static structure is considered, whether a file is open or not would affect the code. Consequently, the state of having a file open would be considered equivalent and thus excluded. Therefore, we augment the file system's encoding with operation codes to distinguish the effect of Operations that do not affect the static structure. We added operation encoding as follows.

- **Distinguish directory node and file node** In the AHU algorithm, all nodes in a rooted unordered tree are defined as indistinguishable; in particular, all leaf nodes are considered identical. However, in a file system tree, there are two types of nodes: directories and files, both of which can serve as leaf nodes. Therefore, in our encoding, directories and files are assigned different codes: directories use brackets [ ], and files use parentheses ( ).
- **Encode file length** For files, files of different lengths occupy different numbers of physical blocks. We also need to distinguish between files of

different lengths. For file encoding, we use the file's length (in blocks) enclosed in parentheses ( ).

- **Opening and closing file**. Opening a file changes the file system's state, and closing it restores it to its previous state. Reading from and writing to a file must occur after the file is opened; therefore, this state must be preserved. We add a sharp # inside the parentheses () to indicate that the file is open.

- **File system level operations** File system level operations like sync, umount/mount, or crash/recovery, do not change the static structure but require further testing after these operations. Hence, the file system state after these operations should be preserved. To distinguish the states, we assigned a version number that is incremented after each file system operation. We placed the version number outside the brackets of the root directory to preserve the distinct states. We also use the version number to limit the number of times we invoke file system operations within a test, thereby reducing the number of states explored. Repeatedly invoking file system operations generally does not help in discovering new file system errors.

- **No layer compression** Unlike the original AHU algorithm, which simply compares whether two trees are isomorphic. The layer-by-layer comparison allows the AHU algorithm to introduce code compression. However, it is not suitable for us. We store the file system state encoding into the `VisitedTable`. We recorded the original code for each layer in the tree without compression.

Operation Trace Encoding We append an operation-trace encoding to the directory-tree encoding. We assign a unique letter to each file system operation. During the exploration and execution of file system operations, we record all operations that transform the initial state into the current state. The sequence of operations from the initial state to the current state forms a string, which we refer to as the operation path encoding. We append this path encoding after the structure encoding. To improve checking efficiency, we need not retain the entire path encoding; only the last few characters suffice. The results will be discussed in detail in the subsequent evaluation section.

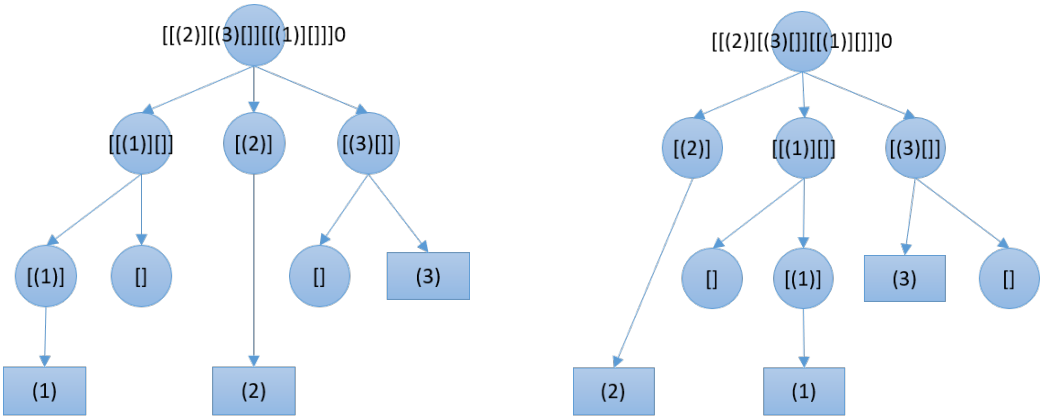
Algorithm 3 illustrates the full algorithm of encoding a file system by starting from a bottom-up perspective. First, for files (as detailed in `EncodingFile`), we represent the file by its size, append an asterisk on the right if the file is open, and then enclose the resulting string in brackets. We then recursively

Algorithm 3 Encode a file system state

```
1: function EncodingDirectory(dir)
2:     Input: Directory dir
3:     Output: String encode
4:     for all item in dir do
5:         if item is Directory then
6:             SubCode[i] ← EncodingDirectory(item)
7:         else
8:             SubCode[i] ← EncodingFile(item)
9:         end if
10:    end for
11:    SubCode[].Sort()
12:    encode ← "[" + SubCode.join() "]"
13:    return encode
14: end function

15: function EncodingFile(file)
16:    Input: File file
17:    Output: String encode
18:    encode ← "(" + file.size
19:    if file is opened then
20:        encode += "#"
21:    end if
22:    encode += ")"
23:    return encode
24: end function

25: function EncodingFileSystem
26:    Output: encode
27:    encode ← EncodingDirectory(root)
28:    encode += (FsVersion)
29:    encode += OperationTrace.SubString()
30:    return encode
31: end function
```



(a) Two isomorphic directory structures have the same encoding

(b) Two isomorphic directory structures have the same encoding

Figure 5.5: Isomorphic directory structure.

encode directories (as detailed in `EncodingDirectory`). When encoding a directory, we first obtain the encodings of all its children (subdirectories and files). These encodings are then sorted lexicographically and concatenated into a single string. A version symbol is appended to the right of this string, which is then enclosed in parentheses. Finally, to encode the entire file system (as in `EncodingFileSystem`), we first encode the root directory, then append an index to the far right corresponding to the number of crash recoveries. Figure 5.5 illustrates the bottom-up encoding process, where rectangular blocks represent files and circular blocks represent directories. Figure 5.5a and 5.5b show two isomorphic directory trees. Although they appear different, they can be made identical by swapping the order of sibling nodes. These two isomorphic trees ultimately yield the same encoding.

Optimization Considering the characteristics of file system encoding, we have implemented the following three optimizations in the MC^3 tool:

- **Encoding Cache**. Given the nature of the MC^3 tool, which operates on a single file system object at a time and makes minimal local changes to the file system with each operation, caching the encoding for each file system node is effective. Whenever an operation changes the file system, we only need to update from the changed node upward, layer by layer, until we reach the root directory. For sibling nodes that have not changed, their cached encodings can be directly retrieved. This caching reduces the average number of nodes that need to be updated from $O(N)$ to $O(\log N)$, where N is the total number of nodes, including

the directory and files.

- **Hash Function**. Our file system encoding is relatively long, proportional to the total number of directories and files, and uses a limited character set. If we restrict the version number to 10 or fewer digits, the entire encoding requires only 15 characters. Murmur Hash [53] is a non-cryptographic hash function invented by Austin Appleby in 2008. It is known for its high speed, good distribution, and low collision rate. We use the 128-bit version of MurmurHash3 from SMHasher [54], and the actual hash value is truncated to its lower 64 bits.

  We compared MurmurHash3 with Boost's native hash function under the same search conditions. The search time with MurmurHash3 was reduced by 32%, whereas memory usage increased by 47%. The increase in memory overhead indicates MurmurHash's low collision rate, and the reduction in time demonstrates its speed. This indirectly confirms the MurmurHash function's high speed and low collision rate. Table 5.3 shows the improvement using Murmur Hash, compared to the Boost built-in Hash.

- **Shorten Operation Trace Code** Typically, more recent operations have a greater impact on the crash consistency result. We retain the most recent operation trace code to reduce redundant state searches. This approach can also improve the searching efficiency by avoiding dynamic memory management. According to our experiment, retaining the latest three operation codes can significantly reduce the search space while preserving the same ability to detect crash consistency errors.

Table 5.3: Time cost and memory cost of searching a 12-level file system, Boost Hash vs Murmur Hash

Hash Type	**Search Time (s)**	**Memory Cost (MB)**
Boost Build in Hash	5100	3123
Murmur Hash	3461	6593
Improvement	32%	47%

5.2.4 File System Model

In this section, we describe how to construct a model from the F2FS file system. The primary goal of constructing a file system model is to minimize

the scale of various data structures within the file system, enabling the exploration of all its boundary conditions at minimal cost. While reducing storage capacity can shrink the overall file system size, many file system data structures have fixed sizes; for instance, each segment contains a fixed number of 512 data blocks. Similar fixed-size structures include index tables, dentry blocks, and others. Furthermore, file systems impose a minimum capacity requirement for normal operation; for example, F2FS requires a minimum capacity of approximately 100 MB, which is excessive for model checking. Thus, we need to develop a file system model to reduce the scale of these fixed-size data structures, enabling model checking to cover a wider range of boundary conditions, including triggering garbage collection (GC).

On the other hand, to support file system state-space search, the model must be self-contained and reproducible. This means all pointers within the file system model must reference internal objects exclusively. Moreover, these pointers should use relative addresses to ensure they remain correctly directed to their target objects when the file system model is copied.

Our tool tests the file system on a file system model rather than a concrete file system, based on the following rationale

1. Different from the black box testing tool, our tool exhaustively checks the file system's state space. This approach to checking requires storing and rolling back the system state, as well as checking other paths. Hence, we request the system support state duplication, which is not supported by most concrete file systems.

2. Reducing the size of the file system to decrease the number of states and storage space. It is not enough to shrink the storage device's capacity. Our model also reduces the metadata footprint in the file system. The following sub-sections will introduce how to model our file system.

3. To evaluate variant file system models, we defined an interface for the file system model. Any file system model that adheres to the defined interface can be checked in the MC^3 tool.

Modeling Data Structures

Abstracting data structures from a real-world file system serves three key purposes:(1) Excluding data structures irrelevant to crash consistency checks to improve verification efficiency and highlight key inspection focus areas;(2) Reducing the scale of the file system model to facilitate the detection of boundary conditions during crash consistency verification;(3) Enabling state

space search, the model must be duplicatable and free from background operations that may interfere with verification.

The core data structures of F2FS include: `page`, `superblock`, `inode`, `dentry`, SIT (Segment Information Table), NAT (Node Address Table), SSA (Segment Summary Area), and Checkpoint. The `page` is not a file system data structure; rather, it is used in Linux for storage I/O caching. F2FS stores file system information, such as the total number of segments in the superblock. Since our F2FS model has a fixed scale, we hard-coded this information and omitted the superblock. The SIT, NAT, SSA, and Checkpoint are the F2FS special data structures. We abstracted these data structures from the concrete F2FS file system, removed unnecessary fields, and shrunk their scale. Table 5.4 demonstrates the scale shrink from the concrete F2FS file system. In the following section, we will introduce how to abstract and construct the core data structures in our F2FS model.

Page and Page management In Linux, both memory management and disk caching operate on a page-by-page basis. A page is typically 4KB in size, which matches the block size of F2FS and most common file systems. Linux's disk caching mechanism is implemented via page cache, with the `struct page` data structure used to manage and track pages. Each page records the inode of the corresponding file and its logical block number. When a file system reads data from disk, it first consults the page cache: if a cache hit occurs, the data is returned directly; otherwise, a new page is allocated, and the data is read into it. Read pages are not immediately released; instead, they are placed in an LRU (Least Recently Used) cache, from which they are evicted only when the cache replacement mechanism is triggered. When data is written to a page, the page is marked as dirty rather than flushed to disk immediately; flushing occurs only when predefined conditions are met.

Our F2FS model captures page management and implements a write cache, but omits read caching and page eviction. Listing 5.1 shows the `CPageInfo` data structure for pages in the F2FS model, which retains the inode number (`nid`), logical block number (offset), and dirty state flag (dirty) from Linux's page management. However, the page size in our model is not limited to 4KB, and the page structure directly stores the data block (rather than pointing to external data storage, as in Linux pages).

For state-space search compatibility, all data in the file system must be reproducible. To this end, we introduce the `CPageAllocator` structure to manage all pages. Listing 5.2 shows the sample code of the page allocator. `CPageAllocator` pre-allocates a fixed number of pages (`m_pages[]`) during initialization. When the file system requires a page, it retrieves one via `CPageAllocator::allocate()` and releases it via `free()` after use. All references to pages in the file system model use indices into `CPageAllocator::m_pages`.

```
class CPageInfo {
public:
    PHY_BLK phy_blk = INVALID_BLK;
    _NID    nid;
    LBLK_T offset = INVALID_BLK;
    bool dirty = false;
    bool in_use = false;
protected:
    union {
        PAGE_INDEX free_link;   // used for free page
           link
        BLOCK_DATA data;
    };
}
```

Listing 5.1: Data structure of page

To read data, the page address is obtained via the `CPageAllocator::page()` method.

In the F2FS model, we cache written data blocks at the page level. Dirty pages are only flushed to disk when a file is closed or the `fsync()` operation is invoked.

Inode An inode is used to store a file, including a directory entry's information, such as the file size, file type (symbolic link, directory, or file), and block index table. We abstract the inode structure from F2FS's inode and omit unused fields such as timestamps and permission data. As shown in Listing 5.3, our `inode` retains only the following critical fields: file size, number of valid data blocks, file type (directory or regular file), hard link count, and an index table. Additionally, we reduce the size of the index table from 923 entries (in native F2FS) to 4 entries.

Dentry The dentry block data structure in our model is largely consistent with F2FS's native `dentry` structure, including a bitmap, a set of `dentry` structures, and a character array for storing filenames. The `dentry` structure is identical to that of F2FS. We reduce the number of `dentry` and filename slots per `dentry` block from 214 (native F2FS) to 4, and the maximum filename length per slot from 8 characters to 2 characters.

SIT (Segment Information Table) SIT is the structure F2FS uses to manage segments. In native F2FS, each `struct f2fs_sit_entry` manages one segment, recording the number of valid blocks in the segment, a bitmap of valid blocks, and the segment's modification time. SIT entries are stored in SIT blocks, with 55 entries per block. In our model, each SIT entry retains

```
1
2  class CPageAllocator {
3  protected:
4      CPageAllocator(void);
5  protected:
6      CPage m_pages[MAX_PAGE_NUM];
7      int m_free, m_used;
8  public:
9      int allocate();
10     void free(int page_id);
11     CPageInfo * page(int page_id);
12 };
13
14 class CF2fs {
15 public:
16     void duplicate_to(CF2fs *dst) {
17         memcpy_s(dst, sizeof(CF2fs), this, sizeof(CF2fs)
               );
18     }
19 protected:
20     ...
21     CPageAllocator m_pages;
22 };
```

Listing 5.2: Dummy code of page allocator

```
1 struct INODE
2 {
3     UINT blk_num;
4     UINT file_size;
5     F2FS_FILE_TYPE file_type;
6     UINT ref_count;
7     UINT nlink;
8     NID index[INDEX_TABLE_SIZE];
9 };
```

Listing 5.3: Data structure of inode

only the number of valid blocks and the valid block bitmap. Since each segment in our model contains only 4 blocks, the bitmap does not require an array and can be stored as a 32-bit integer. We also reduce the number of SIT entries per SIT block to 4.

NAT (Node Address Table) NAT is the structure F2FS uses to manage node blocks, providing a mapping from node IDs (nids) to node block physical addresses to avoid avalanche effects during out-of-place node updates. In native F2FS, the NAT mapping table is an array of `nat_entry` structures (indexed by nid), stored in NAT blocks with 455 entries per block. The file system size determines the number of NAT blocks. Each `nat_entry` records the node's version number, corresponding inode number, and physical address. In our model, each NAT block stores 4 entries; across 4 node blocks, this yields a maximum of 16 nodes.

SSA (Segment Summary Area) SSA primarily provides two functions: (1) a mapping from physical addresses to file logical addresses (used for garbage collection (GC), where data blocks must be moved from physical address A to B, and the logical address is required to update the corresponding index table entries or NAT entries); (2) a journal cache to reduce the frequency of SIT and NAT updates. In native F2FS, each SSA block records 512 entries (corresponding to one segment's summary). Our model records SSA entries in the same manner and on the same proportional scale, but omits the journal cache component for SIT and NAT updates.

Checkpoint Checkpoint is the core data structure F2FS uses to maintain crash consistency. There are two redundant checkpoint copies; only one is updated at a time to ensure that at least one complete copy is always available. Each checkpoint copy contains two version numbers, which are used to verify checkpoint integrity. Checkpoint updates follow a strict order: first write the first version number, then update the checkpoint data blocks, and finally write the second version number. A checkpoint is considered complete only if the two version numbers match. When the file system loads, it checks both checkpoint copies. It prioritizes the complete copy; if both are complete, it uses the newer version; if neither is complete, an error is raised.

Native checkpoints store a wealth of data; our model omits recoverable variables and retains only critical content: current segment information, the active (in-progress) segment in each log, SIT and NAT journals, and SIT/-NAT version bitmaps. To maintain synchronization between SIT and NAT, F2FS stores two copies of each in the metadata. Between checkpoint updates, only one copy is written to, and modified SIT/NAT entries are marked. These modified entries are recorded in the version bitmap during the next checkpoint update. Each bit in the version bitmap corresponds to one SIT or NAT entry, with 0 or 1 indicating whether the entry is valid in the corresponding

Table 5.4: Size of data structure in actual F2FS vs model, unit: Byte

Data structure	Actual F2FS	F2FS model
Entries in inode	928	4
Entries in direct/indirect block	1018	4
Structure in dentry block	214	4
Level of dentry	64	4
Block per segment	512	4
SIT entry per block	55	4
Node number	455	16
NAT entry per block	455	4
Block size	4KB	148 Byte

SIT or NAT copy. This is the key mechanism by which F2FS maintains SIT and NAT consistency during crashes.

We experimented with omitting the SIT/NAT version bitmaps and using only the SIT/NAT journals. This modification does not affect normal file system operation when no crash occurs, but can cause errors due to SIT-NAT inconsistencies during crash recovery. This is because the journal length in checkpoints is limited and cannot store all operations. For example: (1) A node's new physical address may be updated in NAT, but the valid bitmap for that physical block is not updated in SIT, leading to the block being reallocated to other data after crash recovery. (2) Conversely, data may be written, and SIT updated. However, the corresponding node's physical address in NAT is not updated (it still points to the old data block), which may cause the old block to be deleted during segment recycling.

We will elaborate on the differences in crash consistency between F2FS and block-based file systems in the Evaluation section.

Modeling File System Operations

In abstracting file system operations from the concrete F2FS code to our model, we focus on aligning functional implementation with simplified data structures, while preserving the core crash consistency and I/O semantics of F2FS.

All file system operations are redesigned to adapt to the scaled-down data structures, ensuring functional completeness while avoiding redundant state expansion: File I/O Operations: `read()` / `write()`. We added an offset parameter to these two operations and moved the offset management to the workload generator. In Linux, the VFS (virtual file system) maintains all opened files for each process, including the read-write cursor or offset of the file. The MC^3 tool manages the read-write offset by the workload

generator, hence it calls read-write operations using offset and data length. Additionally, we cache only the inode and index tables and omit the data cache and read-ahead features. Data is directly read from or written to the storage as demanded. All node blocks are cached in the `CPageInfo` structure and flushed to storage when the file is closed or `sync()` is called.

Directory Operations: `create()`, `mkdir()`, `rmdir()`, and `move()` operations interact with the scaled-down dentry blocks (4 slots per block, 2-character filename limit). Directory creation allocates dentry slots from the preallocated dentry array, while deletion marks slots as invalid via the dentry bitmap, avoiding dynamic memory allocation and ensuring state reproducibility. The `move()` operation reuses the simplified dentry structure to update filename-to-inode mappings, with consistency guaranteed by foreground-only execution.

This modeling approach ensures that all core F2FS functionalities are preserved while making the model deterministic, state-reproducible, and compatible with exhaustive state space search, laying the foundation for subsequent crash consistency verification.

Modeling `fsck`

`fsck` is a file system utility designed to check for inconsistencies and repair file system errors. Different file systems typically have dedicated `fsck` tools tailored to their specific data structures and operational semantics. In our F2FS file system model, we also constructed an `fsck` model that serves as a component of the checker, responsible for inspecting and repairing the F2FS model after a simulated crash.

The modeling approach for fsck aligns with that of other file system operations, focusing on simplifications adapted to our shrunk data structures. For instance, it excludes the functionality to check and repair symbolic and hard links, features deemed non-essential for our core goal of verifying crash consistency. We will elaborate on the detailed algorithm of the `fsck` model in subsection 5.2.6.

5.2.5 Property of File System Correctness

To verify the correctness of the file system, we introduce four properties in section 4.1.4. These properties describe the link and index types of file mapping models. To verify a full-stack file system model, we extend it to define more properties, including (1) *Operation properties*, which define the correctness of each operation. We check the corresponding properties after each operation. (2) *File integrity properties*, which define that each file is in

its correct location and its data has no corruption. We verify these properties after operations and crash recovery. (3) *Crash consistency properties*, which define the metadata consistency of the file system, especially after crash recovery. We check these properties after crash recovery. However, different file systems have different crash consistency properties. We will provide the example of crash-consistent properties for the F2FS file system. In the remainder of this section, we introduce notation and then present the three categories of properties. In the next section, we will introduce how to check these properties in the MC^3 tool.

We introduce notation before presenting the properties. The notations are extended from Table 4.1. A file in the file system is denoted as a triplet as

$$file \triangleq (Path, Size, Content)$$

where the $Path = String^*$ is a string that identifies the path of the file, $Size = N$ is the size of the file, and $Content = Byte^*$ is the content data of the file. In this study, we ignore other file properties, such as timestamps and permissions. We define the directories as a special type of file.

$$dir \triangleq (Path)$$

. We can use the $isdir()$ property to check if a file is a directory or a file.

$$isdir(d) = T$$

if file d is a directory. A file system is denoted as a mapping from a path to files

$$fs : Path \rightarrow (Size, Content)$$

Operation Properties

We define operation properties for the file and directory operations: *create*, *mkdir*, *remove*, *rmdir*, *move*. A file system operation is denoted as a function from the file system state and parameters to a new file system state.

$$operation :: FS \times PARAM_1 \times ... \times PARAM_n \rightarrow FS$$

Where FS is a file system state, $PARAM_i$ are parameters of the operation, and the number of parameters depends on each operation. For example, we denote the create file operations as

$$create(fs, path) \rightarrow fs'$$

where fs is the file system state before creating the file, *path* is the path of the new file to be created, fs'

The object-based workload generator in MC^3 ensures that all preconditions of operations are satisfied. For each path of searching the file system states, the workload generator creates a sequence of operations, called a trace. In each trace, the post-condition generated by each operation is the precondition for the next operation. Here, we simply define the precondition and focus on the post-condition of each operation.

create The `create` file operation adds an empty file to the file system at the specified path if the path does not exist. Obviously, we can define the correctness of the `create` operation as

$$path \notin fs \wedge create(fs, path) \rightarrow fs' \Rightarrow fs' = fs \cup \{(path, 0, null)\} \quad (5.1)$$

where the *path* is the path of the new file. $(path, 0, null)$ denotes the new created file.

mkdir Similar to the `create` operation, the `mkdir` operation adds a new subdirectory to its parent directory. The path to the new subdirectory is specified as a parameter to `mkdir`. We define the correctness of the `mkdir` operation as

$$path \notin fs \wedge mkdir(fs, path) \rightarrow fs' \Rightarrow fs' = fs \cup \{(path)\} \quad (5.2)$$

Where the *path* specifies the path of the new subdirectory.

remove The `remove` operation deletes a specified file from the file system. The path of the deleted file is specified in the parameter of `remove`. Here, we assume the specified file exists in the file system; the object-based workload generator can ensure this by generating operations based on existing directories. We define the correctness of the `remove` operation as

$$path \in fs \wedge remove(fs, path) \rightarrow fs' \Rightarrow fs = fs' \cup \{(path, 0, null)\} \quad (5.3)$$

rmdir The `rmdir` operation deletes an empty directory from the file system. As with `remove`, the object-based workload generator ensures that the directory to be deleted exists and is empty. For any non-empty directories, the workload generator removes all its sub-items before deleting the directory. We define the `rmdir` as

$$rmdir(fs, path) \rightarrow fs' \Rightarrow fs = fs' \cup \{(path)\} \quad (5.4)$$

move The `move` operation moves a file from the source path to the destination path without changing the file content. The source and destination can be the same directory with different file names, or different directories with the same file name. We define `move` operation's property as

$$\begin{aligned}&(p_s, l, d) \in fs \wedge (p_d, l, d) \notin fs \wedge move(fs, p_s, p_d) \rightarrow fs' \\ \Rightarrow\ & fs = (fs' \cap fs) \cup \{(p_s, l, d)\} \wedge fs' = (fs' \cap fs) \cup \{(p_d, l, d)\}\end{aligned} \tag{5.5}$$

Where (p_s, l, d) is the file with length l, content d, and source path, p_d is the destination path.

File Data Integrity

The fundamental function of a file system is to store user data as files. Ensuring the integrity of file data is a basic requirement of a file system. The so-called integrity of file data means that the data read from a file must match the data written to it. The `read` and `write` operations are denoted in Table 4.1. The property of the file consistency is defined as the same as Equation 4.1

Crash Consistency

For crash consistency, it is essential to verify the integrity of the file system's metadata. File systems rely on interconnected data structures, such as block allocation bitmaps and index tables, to manage storage space and file mapping. If a block is allocated in the bitmap, it must be assigned to exactly one file. If these structures become inconsistent, data loss may occur. Conversely, when a file references a block in its index table, the block must be marked as allocated in the bitmap. However, variant file systems have different metadata, and the properties for metadata consistency differ.

We will demonstrate the metadata consistency of F2FS. The F2FS primarily includes the following metadata:

SIT (Segment Information Table) Used to allocate and manage segments and physical blocks. In an allocated segment, blocks must be written sequentially. Each block in segments has three states: Free, the block has not yet been written to; Valid, the block has been written to and contains the latest version of the corresponding logical block; Invalid, the block has been written to, but the corresponding logical block has been updated to another block. The SIT records which segments are allocated, and how many valid blocks each segment contains. The free blocks only appear in segments that

Table 5.5: Notations on file system

Symbol	Description
$node(file)$	A set of node blocks' nid which are used in the file
$NAT(nid)$	Denotes the physical block which is mapped to the nid. If the nid is not allocated then $NAT(nid) = \perp$
$status(block)$	Denotes the status of the physical block, which is defined in the SIT. The value may be one of $valid, invalid, free$
$block(file)$	Denotes a set of physical blocks that are used in the file.

are written, called current segments. The current segments also need to store the position of the first free block.

NAT(Node Address Table) Used for allocating and managing node blocks and node IDs. Node blocks include inode blocks, index blocks, and indirect index blocks. Each node block is assigned a unique ID, called an NID. The NAT records which NIDs are allocated and their corresponding physical block addresses. To mitigate the wandering tree snowball issue [4], all references to node blocks in F2FS use NIDs. When reading the contents of a node block, the NAT must first be consulted to obtain the physical address corresponding to the NID. When a node block's physical address changes due to content updates, only the NAT needs to be updated, avoiding the wandering-tree snowball effect caused by node updates.

SSA (Segment Summary Area) Records the mapping from physical addresses to logical addresses, primarily used for garbage collection. The SSA stores each physical block's NID (for node blocks) or the file ID and offset to which the block belongs (for data blocks) if the physical block is valid. When a physical block is moved during garbage collection, all pointers pointing to it must be updated. F2FS consults the SSA to find the pointers to the physical block, then modifies the NAT (for node blocks) or the file's index table (for data blocks).

File Index Table This includes the index table in the inode, as well as index blocks and indirect index blocks. The index table is a core part of the file system that maps each logical block of a file to a physical block.

We define properties of consistency among the metadata in Table 4.1.

We defined the properties of crash consistency for the F2FS file system from the following aspects.

- **NAT and File Index** Each allocated NID in the NAT is used by exactly one file.

$$\forall nid, NAT(nid) \neq null \Rightarrow \exists file, nid \in node(file) \tag{5.6}$$

$$\begin{aligned} \forall file_i, file_j \in fs \wedge file_i \neq file_j \Rightarrow \\ node(file_i) \cap node(file_j) = \varnothing \end{aligned} \tag{5.7}$$

- **NAT and SIT** Each node block that is allocated in NAT should be valid in SIT

$$\forall nid, NAT(nid) \neq null \Rightarrow status(NAT(nid)) = valid \tag{5.8}$$

- **File Index and SIT** Each data block allocated in SIT is used by exactly one file.

$$\begin{aligned} \forall b, status(b) = valid \Rightarrow \\ (\exists file, b \in block(file)) \vee (\exists nid, NAT(nid) \neq null) \end{aligned} \tag{5.9}$$

$$\begin{aligned} \forall file_i, file_j \in fs \wedge file_i \neq file_j \Rightarrow \\ block(file_i) \cap block(file_j) = \varnothing \end{aligned} \tag{5.10}$$

However, F2FS imposes a lower consistency requirement on the SSA. When inconsistencies are detected between the SSA and the file index table or the NAT after a crash, F2FS repairs the SSA content according to the index table or the NAT.

5.2.6 File System Checker

To verify whether the file system satisfies the above properties, we designed a file system checker in the MC^3 tool. The file system checker has two parts: the first, the operation checker, verifies operation and file consistency after each operation; the second, the crash checker, verifies crash consistency properties after each crash recovery.

Operation Checker

According to the property definitions, the operation property can be separated into two aspects: the effect of the operation and the remaining items. For example, the result of the `create` operation should be

$$fs' = fs \cup (path, 0, null)$$

. It means that the original set of file system fs should remain unchanged, and the newly added file $(path, 0, null)$ should exist. For the remaining set, we define a set called $RefFs$ that records all changes from operations. Algorithm 4 demonstrates an algorithm to check the property of creating a file. $RefFs$ contains the items in the file system fs' before creating file. FS is the file system fs' after creating file. $Path$ is the path of the newly created file. Because all paths are unique within the file system, if the following three conditions are held, we can say that $fs' = fs \cup (path, 0, null)$

$$(path, 0, null) \in fs'$$
$$|fs| + 1 = |fs'|$$
$$\forall f \in fs, f \in fs'$$

Where $|fs|$ denotes the number of items in the fs, hence, we first check the newly added file $Path$ exists in the file system Fs. Then we check the item number that $|RefFs| + 1 = |Fs|$. Finally, we verify that each item in $RefFs$ is also in Fs. After confirming the property holds, we add the new file into the $RefFs$ to track the file system state.

Similarly, for other operations, we verify the following conditions to ensure that the properties hold.

- `create(path)`

$$(path, 0, null) \in Fs$$
$$|RefFs| + 1 = |Fs|$$
$$\forall f \in RefFs, f \in Fs$$

- `mkdir(path)`

$$(path) \in Fs$$
$$|RefFs| + 1 = |Fs|$$
$$\forall f \in RefFs, f \in Fs$$

- `remove(path)`

$$(path, 0, null) \notin Fs$$
$$|RefFs| = |Fs| + 1$$
$$\forall f \in RefFs \wedge f.path \neq path \Rightarrow f \in Fs$$

- `rmdir(path)`

$$(path) \notin Fs$$
$$|RefFs| = |Fs| + 1$$
$$\forall f \in RefFs \wedge f.path \neq path \Rightarrow f \in Fs$$

Algorithm 4 Check Property of Create File

```
1: Input: RefFs, Fs, Path
2: file = Fs.GetFileInfo(Path)
3: if (file == null) or (isdir(file)) then
4:     return error
5: end if
6: if Fs.Count != RefFs.Count + 1 then
7:     return error
8: end if
9: for all item in RefFs do file = Fs.GetFileInfo(item.path)
10:    if isdir(item) then
11:        if !isdir(file) then
12:            return error
13:        end if
14:    else
15:        if isdir(file) Or (file.size != item.size)  then
16:            return error
17:        end if
18:    end if
19: end for
       RefFs.Add(Path)
20: return True
```

- `move`($path_s$, $path_d$)

$$|RefFs| = |Fs|$$
$$(path_s, l, d) \in RefFs \rightarrow (path_d, l, d) \in Fs$$
$$\forall f \in RefFs \wedge f.path \neq path_s \rightarrow f \in Fs$$

Crash Checker

The crash consistency checker checks the crash consistency properties after each crash. The crash checker is called after a crash recovery test. Since different file systems have varying consistency properties, the MC^3 tool provides an interface that allows the user to design their own crash checker. The Algorithm 5 demonstrates the crash checker for the F2FS file system. We iterate over all node blocks from the inode of the root directory (line 37) recursively, recording all nid and physical block addresses into two predesigned bitmaps. And then compare the nid bitmap with NAT data and the physical block bitmap with SIT. The function `CheckNodeBlock()` in (line 1 19 in Algorithm 3 checks all entries of the node block. If an entry points to another node block, it calls `CheckNodeBlock()` recursively (line 14). Suppose the entry points to a data block, the `CheckDataBlock()` (line 20 33) is called to check. If the data block is a dentry block, `CheckDataBlock()` analyzes the dentry and recursively checks its inode. While creating the nid bitmap and physical bitmap, the checker confirms that no nid or physical block is touched more than once. After creating the nid bitmap and physical bitmap, the checker compares whether the nid bitmap matches the valid nid in NAT. Similarly, the checker compares if the physical block bitmap matches the valid physical block in the SIT.

Algorithm 5: Checking F2FS Crash Consisntency

```
1:  function CheckNodeBlock(nid)
2:      input: NID nid
3:      if nid_bitmap.test(nid) == 1 then
4:          throw nid is double referenced
5:      end if
6:      nid_bitmap.set(nid)
7:      node ← ReadNode(nid)
8:      if block_bitmap.test(node.bid) ==1 then
9:          throw node.bid is double referenced
10:     end if
11:     block_bitmap.set(node.bid)
```

```
12:     for all entry in node.index_table do
13:         if entry is node then
14:             CheckNodeBlock(entry.nid)
15:         else
16:             CheckDataBlock(entry.bid)
17:         end if
18:     end for
19: end function

20: function CheckDataBlock(bid)
21:     Input: BlockId bid
22:     block ← ReadBlock(bid)
23:     if block_bitmap.test(bid) ==1 then
24:         throw bid is double referenced
25:     end if
26:     block_bitmap.set(bid)
27:     if block is dentry then
28:         for all entry in dentry do
29:             CheckNodeBlock(entry.nid)
30:         end for
31:     end if
32:     return encode
33: end function

34: function CheckConsistency
35:     nid_bitmap.clear()
36:     block_bitmap.clear()
37:     CheckNodeBlock(root.nid)
38:     for all block in SIT do
39:         if SIT(block) == valid then
40:             if block_bitmap.test(block) != 1 then
41:                 throw dead block
42:             end if
43:         else
44:             if block_bitmap.test(block) ==1 then
45:                 throw unallocated block
46:             end if
47:         end if
48:     end for
49:     for all nid in NAT do
```

```
50:         if NAT(nid) != null then
51:             if nid_bitmap.test(nid) != 1 then
52:                 throw dead nid
53:             end if
54:         else
55:             if nid_bitmap.test(nid) ==1 then
56:                 throw unallocated nid
57:             end if
58:         end if
59:     end for
60: end function
```

5.2.7 Storage Model

File systems typically run on storage devices. File systems manage data on storage devices, and various user data and metadata are also stored on these devices. The state of the storage device is also part of the file system state. To examine file system operations, we designed a storage device model to simulate block storage devices.

Our storage device model implements the same interface as block storage devices, providing Read, Write, and Sync operations. Additionally, to simulate power failure tests, we added a rollback feature to our storage device model. It can cancel any number of write instructions between two stable states. Through this operation, we can simulate situations where a power failure or crash occurs during file system operations.

To optimize memory usage during file system verification, we refined our storage model. Considering that each file system operation writes only a limited amount of data to the disk, the written data occupies a small proportion of the entire disk space. On the other hand, when checking the file system and searching its state space, the previous state needs to be retained for backtracking. As part of the file system state, data on the storage device must also be copied and saved. This would consume a significant amount of storage space.

Our idea is to record only the differences between two states, which is the data written by the file system since the previous state. The difference data are saved in physical block units and record the write order from the file system. This not only saves the differences between states but also enables rollback of data written to the storage device. We use a list to record the data written by the file system to storage. The data list is global, and all data updates are recorded in it. Each entry in the list corresponds to the

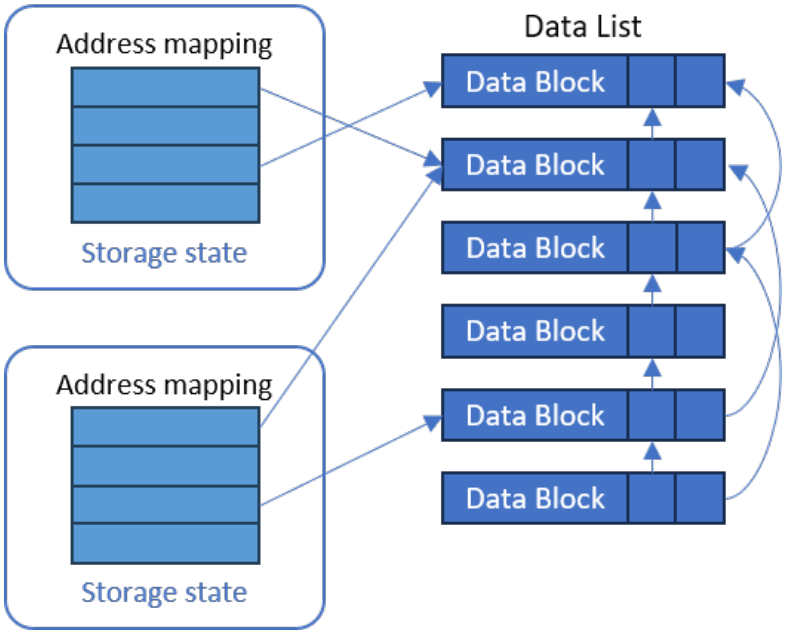


Figure 5.6: Storage Model, state and date list

data of a physical block. It has two pointers: one points to the previously written block in the order, and the other points to the previous version at the same physical address. In each copy of the storage model state, we designed a mapping from physical addresses to the corresponding entries in the data list to optimize read operations. Figure 5.6 indicates the structure of the storage model and data list. When the file system reads data from storage, it accesses the data entry from the address mapping according to the physical address. The search process simply needs to copy the address mapping when it duplicates the storage model's state.

When the file system writes data to storage, it appends the written data to the end of the data list, causing the list pointer to point to the original list tail. The version pointer points to the previous version of the same address. This pointer can be obtained from the address mapping table. Finally, update the address mapping in the current state. When we need to cancel some previously written data to simulate crash scenarios, we delete the entries at the end of the data list. When deleting data, redirect the link corresponding to the address in the address mapping table to the original data. The Figure 5.4 demonstrates the storage model.

```
class CDataEntry {
    DataBlock data;
    CDataEntry * m_list_pointor;
    CDataEntry * m_version_pointer;
};

class CStorage {
    void Write(DataBlock * data, int lba, int sectors);
    void Read(DataBlock * data, int lba, int sectors);
    void Sync();
    void RollBack(int steps);
protected:
    CDataEntry * m_mapping[MAX_LBA];
};
```

Listing 5.4: Dummy code of storage model

5.3 Evaluation

We will evaluate our MC^3 tool from the following perspectives.

- Can the MC^3 tool detect new counterexamples of the file system? What insights do these issues provide us?
- How does the coverage of MC^3 tests compare to CrashMonkey?
- How does the performance of MC^3 tests compare to CrashMonkey?
- The impact of optimization solutions on performance.

5.3.1 Experimental Preparation

We implemented the full-stack model checking approach using the MC^3 tool in C/C++. The core of MC^3 is constructed by 18 files with a total of 3765 lines, including C/C++ header files. The F2FS model comprises 10 files, totaling 3848 lines. The MC^3 can perform an exhaustive test on the specified file system model. If any counterexample is detected, the MC^3 tool outputs the trace of the counterexample into a JSON file. The trace file can be replayed by the MC^3 tool using `trace` mode for debugging.

We used the MC^3 tool to test the F2FS model and identified counterexamples. We run the MC^3 tool on a PC that was configured with an Intel(R) 12th GEN CPU Core(TM) i7-1260P (2.10GHz), 16GB memory, and 2TB SSD. MC^3 runs on Windows 11. To compare performance, we also

Table 5.6: Summary MC^3 tool and compare with file mapping model

	File Mapping Model	**F2FS full-stack model**	**MC^3 Tool**
Language	Promela	C/C++	C/C++
File Number	3	10	18
Code Size (line)	1770	3848	3765

```
1 mkdir("/A");
2 create("/foo");
3 write("/foo", 161233);
4 mkdir("/A/BAR");
5 write("/foo", 131826);
6 mv("/A", "/C");
7 sync();
8 crash()
```

Figure 5.7: The trace leads to a crash consistency error

ran CrashMonkey on the same PC running Ubuntu 16.04 LTS. We executed CrashMonkey on the physical machine rather than in a virtual machine.

5.3.2 Counterexample

First, we verify the correctness of the file system model. We disabled checking the crash condition and ran MC^3, up to a trace length of 13; no anomalies were found. We then re-ran MC^3 to enable the crash condition. A series of counterexamples was detected at trace lengths of 8 or more. Figure 5.7 demonstrates a typical error trace that leads to a crash consistency issue. After the crash, which follows a move operation after overwriting a file, all files and directories are lost.

Error Analysis

Upon analysis, the root cause of these errors is essentially the same reason, which is caused by out-of-place updates. In this trace (Figure 5.7, the first `write("/foo")` (line 3) writes 161233 bytes to the file, approximately 315 blocks or 20 segments of data. We assume the addresses of these segments are from S_1 to S_{20}. After the operation `mkdir("/A/BAR")` (line 4), a checkpoint is recorded. The `write("/foo")` (line 5) overwrites the original data with 131826 bytes, approximately 258 blocks or 16 segments. These updates

render the data blocks from S_1 to S_{16} invalid. Then, the `mv("/A","/C")` (line 6) modifies the dentry block (we assume the address is b_1) of the root directory, writing new data to another block b_1' and rendering the original block b_1 invalid. This coincidentally invalidates all data blocks in the segment S_1 that contain the root's dentry data block b_1 (in this case, block b_1 is in segment S_1), leading to their reclamation. At this point, a crash occurs; our simulated crash mechanism indicates that the crash occurs within `sync()` (line 7), thereby corrupting the checkpoint data saved in `sync()`. During the recovery process, F2FS restores the metadata from the previous checkpoint, which is after the `mkdir()` (line 4). Additionally, the pointer to the root directory's dentry block is restored to b_1. However, the original b_1 is reclaimed and reused by other data, and the contents of the root directory are lost.

This appears to be a general issue of log-structured file systems. We reproduced similar errors across different workloads. In the out-of-place update file system, when we update any data, the file system writes the new data to an empty block, which is different from the original location. Then the file system updates the pointer in the index block to point to the updated data. The original physical block is marked as invalid, but its data will be kept for a moment until the segment is reclaimed. Once a crash occurs and the index table is rolled back to an older version, the data block pointer is also rolled back to the older physical location. If the old physical block is reclaimed, the consistency error happens.

Error Recall

We attempted to reproduce this issue in the actual F2FS file system but were unsuccessful. The primary reason is that the actual file system maintains sufficient over-provisioned segments. Generally, it is difficult to fill the entire file system, especially for LFS, which prevents new data writes when physical space is nearly full. These free segments are allocated first, while invalid segments are not immediately reclaimed and reused. This allows the old data to be preserved. Even if segment reuse causes data loss, it typically affects file data and does not lead to entire file system corruption. In our model, changing the free list to FIFO can reduce the occurrence of this problem.

This is one reason why CrashMonkey or other black-box testing tools cannot detect this issue. In addition, there are two other reasons why CrashMonkey fails to identify this error: triggering this issue requires at least six key operations (lines 1, 6), whereas CrashMonkey's ACE generates skeleton traces with a maximum of three key operations. In our experiments, generating skeleton traces with more than three operations could not be completed within several days. Another reason is that this operation requires at least

Table 5.7: Comparison of Test Coverage Among MC^3, CrashMonkey/ACE and File Mapping Model

Features	**MC^3**	**CrashMonkey/ACE**	**File Mapping**
File Read Write	Yes	Yes	Yes
Directory/File Operation	Yes	Yes	No
Mount/Unmount	Yes	No	No
Crash Recovery	Yes, Multiple	Yes Single	Yes, Multiple
Garbage Collection	Yes	No	No

two file write operations, with each write data larger than the length of two segments, to ensure that at least one segment is invalidated and reused. However, ACE writes file sizes between 8KB and 20KB, which, for an actual F2FS file system with a segment size of 2MB, is insufficient to trigger segment recycling. In the next section, we will provide a detailed comparison of the test coverage between CrashMonkey and the MC^3 tool.

Test Coverage

Next, we analyze MC^3's test coverage from these aspects and compare it with the File Mapping Model and CrashMonkey. Test coverage is primarily evaluated based on the explicit functions of the file system, such as directory operations and file read/write, and its implicit background functions, such as block allocation and Garbage Collection in F2FS. Table 5.7 summarizes the test coverage of each test tool.

The object-based workload generator in MC^3 supports all file system operations, including directory and file operations, file reading and writing, file system mount/unmount, and crash consistency testing. CrashMonkey also covers basic directory and file operations, file reading and writing, and crash-consistency testing based on ACE configuration. However, CrashMonkey does not include normal mount/unmount operations, and its crash consistency testing involves only a single crash event, lacking consecutive crash testing. The File Mapping Model only includes file creation, deletion, and read/write operations, with no directory-related operations. While it supports multiple crash tests, it does not include mount/unmount testing.

Regarding other file system features, MC^3 tests can cover up to 12 subdirectories or files. For the entire F2FS model, comprising 60 segments and 960 blocks, our tests achieved a maximum data write of 2800 blocks and a minimum free space of 5 segments. This indicates that Garbage Collection was triggered, along with segment allocation and reclamation. In contrast,

CrashMonkey operates on a 100MB file system with 50 segments and a total of 25,600 blocks. However, CrashMonkey's operations involve a maximum of only 5 files and directories, with a maximum file write size of only 32KB, approximately 8 blocks. This level of data writing is insufficient to trigger the allocation of new segments or to activate the Garbage Collection mechanism in F2FS.

5.3.3 Performance and Running Cost

We evaluated the performance and memory usage of the MC^3 tool and compared them with those of the CrashMonkey/ACE tool. As a result, CrashMonkey exhibits lower efficiency than the MC^3 tool.

Test Performance

We ran the MC^3 tool and CrashMonkey/ACE to compare test performance across different trace-length configurations. We run both the MC^3 tool and CrashMonkey on the same PC described in subsection 5.3.1. To test a file system using CrashMonkey, we first need to generate a set of workloads using ACE scripts. ACE generates each workload as a C-language program that performs a series of file system operations. Then we compile the generated C-language workloads into dynamically linked .so files. Finally, we call CrashMonkey to execute the tests using workload .so files. We selected the parameters $-nFalse-dTrue$ for ACE to generate workloads with skeleton lengths of 1, 2, and 3, yielding average numbers of operations of 6, 10, and 13, respectively. As a result, we cannot find a file system bug using CrashMonkey. However, we were unable to generate heavier workloads, such as sequence length 4 or demo option set to false, on my PC. For MC^3, we performed the search with the same number of operations.

Figure 5.8 compares the test time and test efficiency between CrashMonkey and MC^3. We observe that CrashMonkeys' runtime is significantly longer than MC^3, with trace lengths of 6 and 10. For the trace length of 13, CrashMonkey is faster than MC^3. However, the total test traces or total operation count for CrashMonkey is much lower than that of MC^3. And CrashMonkey's execution efficiency (operations per second) is much lower than MC^3. 5.8 shows the time cost and memory cost for CrashMonkey and MC^3.

Memory Cost

MC^3's memory cost is higher than CrashMonkey's, but acceptable. MC^3 stores all search states and intermediate results in memory during runtime

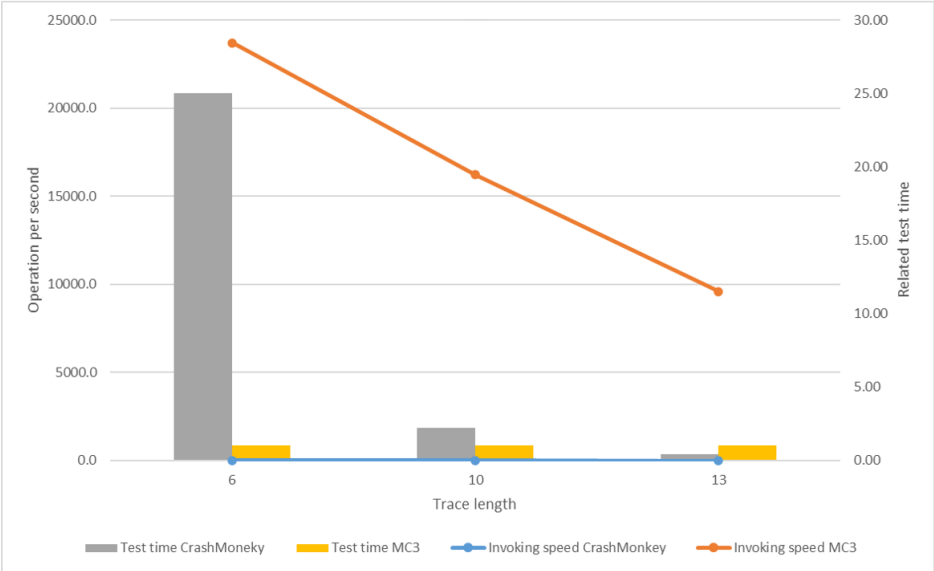


Figure 5.8: Test time and invoking speed, CrashMonkey vs MC^3

Table 5.8: Time cost and memory cost, CrashMonkey vs MC^3

	CrashMonkey			**MC^3**		
Trace Length	**Time (s)**	**Disk Size (MB)**	**Operations**	**Time (s)**	**Memory Size (MB)**	**Operations**
6	25	8.5	107	1	48	2.3×10^4
10	1062	339.0	7524	475	561	7.7×10^6
13	28,230	7873.2	16,423	68,117	11,198	6.3×10^8

and does not utilize external storage. Its primary memory consumption comes from the open list used for depth-first search and the hash table storing visited states. Consequently, its memory consumption is higher than that of CrashMonkey. CrashMonkey, which runs on a real operating system, primarily consumes memory to execute a test workload and simulate crashes, amounting to approximately 20 MB. Additionally, the intermediate states of workloads generated by ACE, including source code and compilation results, are stored on disk. The specific memory and disk consumption figures for MC^3 and CrashMonkey are recorded in Table 5.8. Overall, CrashMonkey's memory consumption is lower than that of MC^3. When testing to a depth of 13 operations, MC^3 consumes 11GB of memory, which is affordable for modern mainstream PCs configured with 8-16GB of RAM.

```
mkdir("/A", 0777)
open("/foo", O_RDWR|O_CREAT, 0777);
link("/foo", "/Abar");
open("/Abar", O_RDWR|O_CREAT, 0777);
fsync("/Abar");
checkpoint(0);
```

Figure 5.9: Duplicated prefix trace

Comparison Result

Overall, CrashMonkey's testing efficiency is lower than that of MC^3, and the cost of finding file system errors is higher. Running the seq-3 skeleton on our PC for about 8 hours failed to uncover file system errors; higher settings could not be completed on the PC. CrashMonkey's team required 780 virtual machines to run concurrently for 48 hours on a cloud computing platform with 65 nodes and 128GB of memory each to detect file system errors. In contrast, MC^3 can uncover deep-seated errors in F2FS within a few days on a personal PC.

The lower operational efficiency of CrashMonkey is primarily attributed to two reasons.

- CrashMonkey runs on a real file system, involving substantial actual disk I/O operations, which results in longer execution times for each operation. In contrast, MC^3 invokes operations on a file system model, which is small and fully runs in memory.

- ACE enumerates all possible workloads, generates independent traces, and then passes them to CrashMonkey for execution. Each execution must start from the file system's initial state. This results in a large number of repeated prefix sequences across all traces. For instance, a 6-step prefix trace (Figure 5.9) was used 852 times across various workloads. In contrast, MC^3, by employing a state-space search method, avoids re-executing traces with identical prefixes. This allows MC^3 to explore a larger search space for the same number of operations.

5.3.4 Optimization

Search with Multi-Thread

In the state search engine, we invoke each operation in parallel with multiple threads. Parallel processing increased the search speed and reduced the

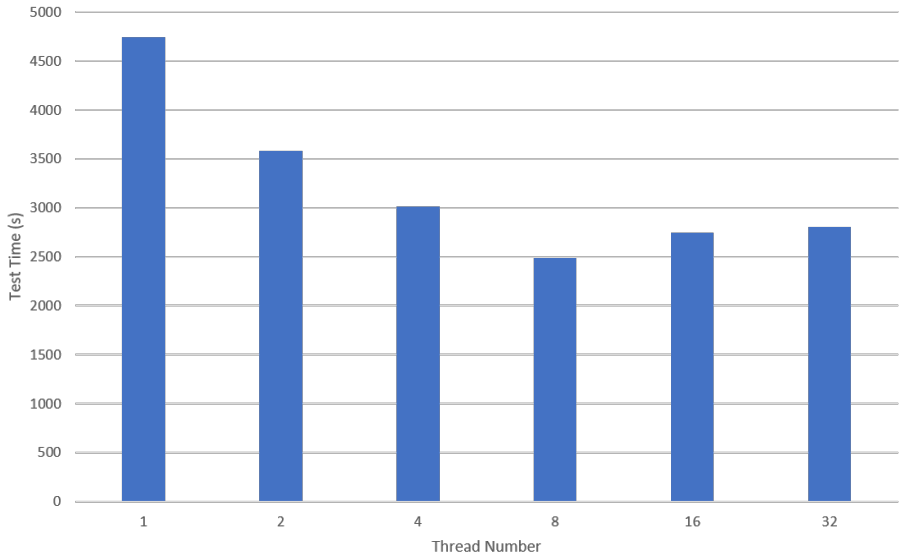


Figure 5.10: Comparison of checking duration for different threads

checking time. Figure 5.10 shows the time required to scan the entire file system at a search depth of 11 layers under different degrees of parallelism. When the thread number increases from 1 to 8, the search time decreases noticeably. Further increasing the thread time provides little additional reduction in search time. Since only the state expansion phase is parallelized, the states generated after operation execution must still be queued for insertion into the open list. Other search phases remain single-threaded; thus, search efficiency does not scale linearly with the number of threads. Additionally, in addition to the parallel worker threads, a main thread and a monitoring thread (for displaying inspection progress) are active, resulting in the actual thread count exceeding the configured parallel threads. Given that the test PC's CPU has only 16 cores, exceeding 16 threads incurs additional overhead, contributing to a decline in search efficiency beyond this threshold. This design was chosen for its relative simplicity and minimal additional memory overhead compared to a single-threaded approach.

However, a drawback of parallel search arises when it is combined with duplicate-state detection: it can introduce non-determinism in the search process. Because parallel-expanded states may complete in varying orders across executions, the sequence of state exploration can differ. For instance, when expanding successor states S_1 to S_n from state S_0, suppose the successor state S_{1i} of S_1 is identical to the successor state S_{2j} of S_2, and S_{2j} has a greater depth than S_{1i}. Although the parallel search consistently generates

the same set of states S_1 to S_n, the order in which they are completed is non-deterministic. If S_2 completes before S_1, the depth-first strategy will prioritize searching the successors of S_2, including S_{2j} and its subtree, before processing S_1 and S_{1i}. Consequently, S_{2j} and its descendants are fully examined. When S_{1i} is later encountered, its successors are still searched due to its lower depth. Conversely, if S_1 completes first, S_{1i} is searched early, and when S_{2j} is generated, it is discarded as a duplicate with greater depth. This variability reduces the total number of states searched in some cases but does not impact the final results.

Isomorphic Detection

We evaluated the effect of the isomorphic detection feature. We performed experiments to compare the number of operations and the number of states explored across the entire file system state space for different search depths. We disabled crash conditions to prevent the impact of counterexamples.

Figure 5.11 indicates the test result and shows the effect when we exclude isomorphic states. The x-axis represents different searching depths, while the y-axis represents the number of states or operations on a logarithmic scale. The blue and orange lines show the number of states and operations, respectively, including the isomorphic case. In this case, the number of operations is almost equal to the number of states, meaning that all states generated by operations are included, which matches the inclusion of isomorphic states. The gray and yellow lines show the number of states and the number of operations when isomorphism is excluded. Excluding isomorphic states makes the number of states less than the number of operations, meaning some isomorphic states are excluded. The graph shows that excluding isomorphic states reduces the number of states by about 99.97% compared to the case of including them. It shows that our isomorphic state detection effectively reduces the number of states to search.

Operation Trace Encoding

The previous subsection demonstrates that the isomorphic detection reduces the number of states explored and improves search efficiency. In this subsection, we evaluate whether excluding isomorphic states might lead to missed file system errors. The file system encoding comprises two main components: static structural encoding and dynamic operation-sequence encoding. The static structural encoding is based on an extended AHU algorithm, which has been proven correct and sound. That is, two distinct structures are guaranteed to yield different encodings.

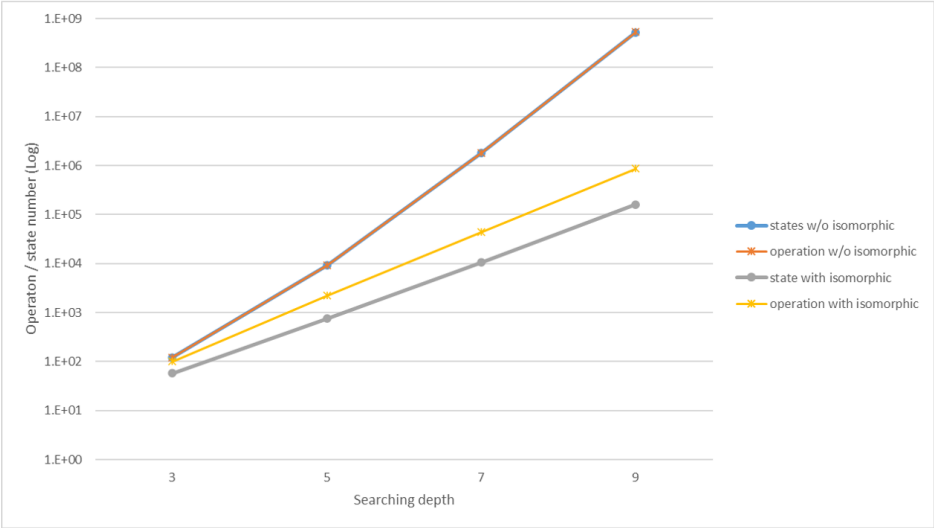


Figure 5.11: Operation number and state number comparison between without and with isomorphic detection

Since different file system operation sequences can produce the same static structure, Figure 5.12 illustrates two distinct traces that result in identical static structures, which contain one directory `"/C"` and one file `"/foo"` under the root. Directory `"/C"` contains one subdirectory `"/BAR"`. As well, the file `"/foo"` in the two traces has the same length. However, different operation sequences affect crash-test outcomes. The trace in Figure 5.12a triggers a crash consistency issue, as discussed in the section 5.3.2, while the trace in Figure 5.12b does not induce an error.

Through experiments, we compared the impact of different operation en-

```
1 mkdir("/A");
2 create("/foo");
3 write("/foo");
4 mkdir("/A/BAR");
5 write("/foo");
6 mv("/A", "/C");
7 sync();
8 crash()
```

(a) The trace leads to a crash consistency error

```
1 mkdir("/A");
2 create("/foo");
3 mv("/A", "/C");
4 mkdir("/C/BAR");
5 sync();
6 write("/foo");
7 write("/foo");
8 crash()
```

(b) The trace has the same structure code, but does not lead to an error

Figure 5.12: Different trace leads to the same structure encoding.

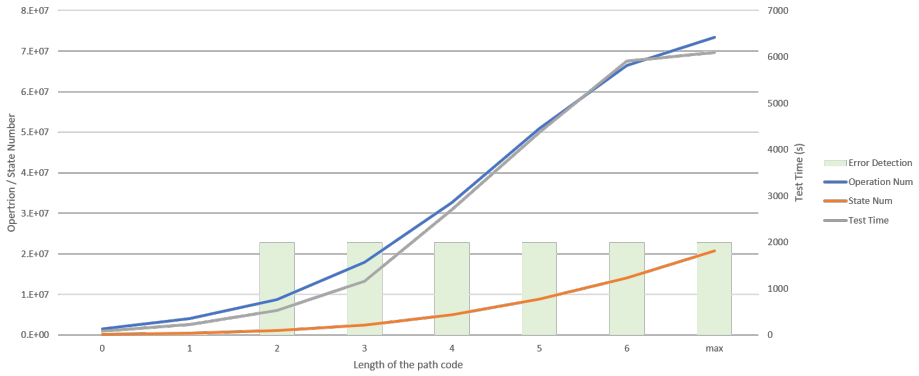


Figure 5.13: Path code length vs Operation number and state number

coding lengths on test results. Given that operations closest to the crash time typically have the greatest impact on crash consistency, we incorporate a partial operation encoding of the latest operations. Figure 5.13 presents the error-detection capability and the corresponding testing time for varying operation-encoding lengths. Using the inclusion of isomorphic states as a baseline, at a search depth of 8, two errors were detected. Without operation codes, we could not detect the errors in the same search depth. As the operation code length increases, the number of states searched and the search time increase. When the length of the operation code becomes greater than or equal to 2, we can detect the same error by including an isomorphic state.

5.4 Discussion

In this chapter, we present the design, implementation, and evaluation of MC^3, a comprehensive model-checking tool for verifying crash consistency in full-stack file-system models. Building upon the limitations identified in Chapter 4, where SPIN-based verification was constrained to simplified file mapping models, MC^3 enables the verification of full-stack file system models with complete directory tree structures, including log-structured file systems like F2FS.

The core of MC^3 consists of several key components: a state search engine that performs exhaustive depth-first exploration of the file system state space; an object-based workload generator that systematically produces all valid file system operations; a novel file system encoding algorithm that detects and eliminates redundant file system state to optimize state space exploration; and a flexible interface that supports C/C++ implementations of file system

models. These components work together to provide a practical and scalable solution for crash consistency verification.

Our experimental evaluation demonstrates MC^3's effectiveness in uncovering subtle crash consistency issues within an acceptable time frame and resource that elude traditional black-box testing tools. Through extensive checking of an F2FS model, we discovered a previously unknown vulnerability in which metadata rollback, combined with block reuse during garbage collection, can lead to silent data loss. This finding reveals a fundamental challenge in maintaining crash consistency in log-structured file systems and provides new insights into their failure modes.

A comparative analysis using black-box testing, such as CrashMonkey, highlights the distinct advantages and limitations of our model-checking approach. MC^3 has the following advantages over CrashMonkey:

- **Superior Testing Efficiency and Lower Cost**: MC^3 operates on an in-memory file system model, avoiding the costly disk I/O and system reboots required by CrashMonkey. Its state-space search algorithm avoids re-executing common prefixes across test sequences, allowing it to explore a vastly larger number of operations on a single desktop PC within a practical time frame.

- **Broader Test Coverage**: MC^3 can systematically test longer operation sequences, which is essential for triggering complex error conditions. By scaling down the model, it can easily reach boundary conditions, such as full directories, and triggering garbage collection, that are difficult to achieve in a real large-capacity file system with CrashMonkey.

- **Ability to Uncover Subtle, Design Level Bugs**: MC^3 successfully identified a latent bug in F2FS arising from the complex interaction between out-of-place updates, garbage collection, and metadata checkpoints, a bug that required a long, specific sequence of operations beyond CrashMonkey's practical generation and testing capabilities.

- **Higher Degree of Automation**: The object-based workload generator automatically produces all possible operations from any given state, requiring no manual test case design, whereas CrashMonkey relies on pre-defined or generated test skeletons.

However, our approach has certain limitations compared to black-box testing:

- **Requirement of Specialized Expertise**: Building an accurate file system model for MC^3 requires deep knowledge of the system's internal data structures and algorithms, posing a higher barrier to entry than using CrashMonkey, which only requires knowledge of standard file system APIs.
- **Potential to Miss Implementation Specific Bugs**: Since MC^3 verifies an abstract model, it may miss bugs that reside in the concrete implementation details abstracted away in the model like specific memory management errors). CrashMonkey, testing the real system, can, in principle, find any type of bug.

The development of MC^3 represents a significant advancement in file system verification methodology, bridging the gap between formal methods and practical file system testing. In conclusion, while black-box tools like CrashMonkey are highly valuable for testing and validating existing file system implementations, MC^3 is particularly suited for the design and prototyping phase of new file systems and algorithms, where it can efficiently detect profound, design-level crash consistency errors that are otherwise extremely difficult to find.

Chapter 6

Summary

This thesis presents a comprehensive and evolving methodology for applying model checking to the critical problem of verifying file system crash consistency. Our work demonstrates a clear trajectory from verifying core components with established tools to building a specialized, powerful framework for full-stack file system analysis.

In Chapter 4, we established the foundational feasibility of this approach. By abstracting the file-mapping components of both link-based (FAT) and index-based (ext2) file systems into Promela models and verifying them with SPIN, we proved that model checking can exhaustively uncover subtle crash-consistency bugs that are difficult to find through testing alone. This initial work yielded significant insights: it confirmed that the non-atomicity of metadata updates is the root cause of consistency violations, and it led to the development of a write-ordering mechanism that can prevent critical errors. However, this approach faced inherent limitations. The manual modeling process was labor-intensive, the state explosion problem severely constrained the model's complexity, and essential features like directory trees and advanced algorithms (e.g., journaling, garbage collection) were impractical to include.

Driven by these limitations, we developed MC^3 (Model Checking for Crash Consistency), detailed in Chapter 5. MC^3 represents a significant architectural and methodological advancement. By shifting the modeling language to C/C++, we enabled the creation of complex, full-stack file system models that faithfully capture directory structures and sophisticated mechanisms such as garbage collection and node address tables found in log-structured file systems. To make exhaustive checking of these larger models computationally feasible, we introduced key innovations: an object-based workload generator that systematically explores the operation space, and a novel file system encoding algorithm that identifies isomorphic directory

trees, reducing the state space by over 99% and making comprehensive exploration practical on a standard desktop PC.

The efficacy of our methodology is demonstrated by a significant finding: using MC^3, we identified a previously unknown crash-consistency bug in a model of the F2FS file system. This bug, which can lead to silent data loss, stems from a fundamental conflict in log-structured file systems between metadata rollback after a crash and the reuse of physical blocks by garbage collection. This finding, which evaded detection by black-box testing tools like CrashMonkey, underscores the unique value of model checking in exposing deep, design-level vulnerabilities that require long, specific operation sequences to trigger.

A comparative evaluation with CrashMonkey clearly delineates the strengths and trade-offs of our approach. MC^3 offers superior efficiency and coverage; its in-memory operation and state-space search strategy allow it to execute orders of magnitude more operations per second and explore significantly longer test sequences. This enables MC^3 to reach complex corner cases that are practically inaccessible to CrashMonkey, which is hampered by the overhead of real I/O and system reboots. However, the model checking approach has its own constraints. It requires deep expertise to build an accurate file system model, and it verifies the design rather than the implementation, so it may miss implementation-specific bugs that black-box testing would catch.

In conclusion, this research provides a multi-faceted contribution to the field of file system reliability. We have shown that model checking is not only viable but also highly effective for crash-consistency verification. Our journey from SPIN to MC^3 illustrates a scalable path for applying formal methods to systems software, moving from component-level to system-wide verification. The MC^3 tool serves as a powerful complement to existing testing techniques: while black-box tools like CrashMonkey are excellent for testing and validating concrete implementations, MC^3 is uniquely positioned for the design and prototyping phase, where it can efficiently and automatically uncover profound, design-level errors before they become entrenched in deployed systems. This work ultimately provides developers with a powerful framework to build more robust and reliable storage systems.

Bibliography

[1] J. Yuan, T. Aoki, and X. Guo, "Comprehensive evaluation of file systems robustness with SPIN model checking," *Software Testing Verification and Reliability*, vol. 32, 9 2022.

[2] L. Yu, C. Fu, Y. Du, A. Deng, M. Zhao, L. Shi, and C. J. Xue, "An Empirical Study of F2FS on Mobile Devices," in *IEEE 23rd International Conference on Embedded and Real-Time Computing System and Applications (RTSCA)*, 2017.

[3] C. Lee, D. Sim, J. Hwang, and S. Cho, "f2fs a new file system for flash storage," *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[4] G. Sachs, K. Munegowda, and G. T. RAJU Professor, "Avoidance techniques for Snowball effect of Wandering Tree in Flash Memory based File Systems Power fail safe FAT and ExFAT ile systems View project Query Context Model for Information Retrieval View project Keshava Munegowda Avoidance techniques for Snowball effect of Wandering Tree in Flash Memory based File Systems," tech. rep., 2015.

[5] C.-S. Park and T. H. Han, "Fast mounting method for NAND flash memory file system using offset information," *IEEE Xplore*, 2010.

[6] J. Yuan, K. Tanaka, and T. Aoki, "Performance Evaluation of Multi-Head Logging in Flash File Systems," in *2025 IEEE 49th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1764–1769, IEEE, 7 2025.

[7] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," *Proceedings of FAST 2012: 10th USENIX Conference on File and Storage Technologies*, no. September 2015, pp. 101–116, 2012.

[8] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang, "Specifying and checking file system crash-consistency models," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 02-06-Apri, no. 212, pp. 83–98, 2016.

[9] J. Corbet, "ext4 and data loss [LWN.net]," 2009.

[10] Jonathan Corbet, "Toward better testing," 2014.

[11] J. Yuan, T. Aoki, and X. Guo, "Comprehensive Robustness Evaluation of File Systems with Model Checking," *Proceedings - 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS 2020*, pp. 99–110, 2020.

[12] M. McKusick, Kirk and T. Kowalski, J., "FSCK : the UNIX file system check program," no. 4031, pp. 581–592, 1996.

[13] R. Card, T. Thedore, and S. Tweedie, "Design and implementation of the study," *Processdings - of the First Dutch International Symposium on Linux*, pp. 5–21, 1994.

[14] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," tech. rep., 1984.

[15] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," tech. rep., 2008.

[16] M. K. McKusick, "Running "fsck" in the Background," in *BSDCon 2002 (BSDCon 2002)*, (San Francisco, CA), {USENIX} Association, 2002.

[17] V. Henson, A. van de Ven, A. Gud, and Z. Brown, "Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair," in *Second Workshop on Hot Topics in System Dependability (HotDep 06)*, (Seattle, WA), {USENIX} Association, 2006.

[18] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. Vol.10, no. No. 1, pp. 26–52, 1992.

[19] Y. Son, H. Kang, H. Han, and H. Y. Yeom, "An Empirical Evaluation of NVM Express SSD," in *Proceedings - 2015 International Conference on Cloud and Autonomic Computing, ICCAC 2015*, pp. 275–282, Institute of Electrical and Electronics Engineers Inc., 10 2015.

[20] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. u. Lee, S. Kang, Y. Won, and J. Cha, "Deduplication in SSDs: Model and quantitative analysis," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, 2012.

[21] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*, pp. 33–50, 2018.

[22] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Crashmonkey and Ace: Systematically testing file-system crash consistency," *ACM Transactions on Storage*, vol. 15, no. 2, 2019.

[23] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," *OSDI 2004 - 6th Symposium on Operating Systems Design and Implementation*, pp. 273–287, 2004.

[24] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard, "Verifying a file system implementation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3308, pp. 373–390, 2004.

[25] R. Joshi and G. J. Holzmann, "A mini challenge: Build a verifiable filesystem," *Formal Aspects of Computing*, vol. 19, no. 2, pp. 269–272, 2007.

[26] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare logic for certifying the FSCQ file system," *SOSP 2015 - Proceedings of the 25th ACM Symposium on Operating Systems Principles*, pp. 18–37, 2015.

[27] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, pp. 576–580, 10 1969.

[28] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif, "Inside a verified flash file system: Transactions and garbage collection," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9593, pp. 73–93, 2016.

[29] G. J. Holzmann, "The Model Checker SPIN," Tech. Rep. 5, 1997.

[30] T. Ball and S. K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," in *SPIN 2001 Workshop on Model Checking of Software*, 5 2001.

[31] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," in *Automated Software Engineering*, vol. 10, pp. 203–232, 4 2003.

[32] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. I. Siminiceanu, "Model-checking the Linux Virtual File System," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5403 LNCS, pp. 74–88, 2009.

[33] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2006, pp. 243–257, 2006.

[34] M. Kokologiannakis, I. Kaysin, A. Raad, and V. Vafeiadis, "PerSeVerE: Persistency semantics for verification under ext4," *Proceedings of the ACM on Programming Languages*, vol. 5, 1 2021.

[35] Wikipedia, "Hard disk drive - Wikipedia."

[36] Wikipedia, "Solid-state drive - Wikipedia."

[37] C. E. Stevens and D. Colgrove, "Technical Comittee TI13 - ATA / ATAPI Command Set - 2," 2010.

[38] T. S. Chung, D. J. Park, S. Park, D. H. Lee, S. W. Lee, and H. J. Song, "A survey of Flash Translation Layer," *Journal of Systems Architecture*, vol. 55, pp. 332–343, 5 2009.

[39] Microsoft Corporation, "Microsoft FAT Specification," tech. rep., 8 2005.

[40] T. Ts'o, "Ext2fs Home Page."

[41] Kim T., Shin K., Lee T., and Jung K., "Design of a Reliable NAND Flash Software for Mobile Device," in *The Sixth IEEE International Conference on Computer and Information Technology*, (Seoul, Korea), pp. 173–173, IEEE, 2006.

[42] Lee Tae-Hoon, Park Song-Hwa, Kim Tae-Hoon, Lee Sang-Gi, Lee Ju Kyoung, and Chung Kidong, "RFFS : Design of a Reliable NAND Flash File System for Embedded system," *The KIPS Transactions*, vol. 12A, no. 7, pp. 571–582, 2005.

[43] D. Woodhouse, "JFFS : The Journalling Flash File System," tech. rep., Ottawa Linux Symposium. 2001, 2001.

[44] C. Manning, "How Yaffs Works," tech. rep., https://yaffs.net/, 2012.

[45] A. Kawaguchi, H. Motoda, and S. Nishioka, "A Flash-Memory Based File System," in *In Proceeding of the USENIX ATC*, (New Orleans, LA, USA), pp. 155–164, Proceedings of the 1995 USENIX Technical Conference, 1 1995.

[46] S. Liu, G. Xuetao, D. Tong, and C. Xu, "Analysis and Comparison of NAND Flash Specific File Systems *," *Chinese Journal of Electronics*, vol. 19, no. 3, 2010.

[47] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, *Handbook of model checking*. 2018.

[48] G. J. Holzmann, *Spin Model Checker, The: Primer and Reference Manual*. Addison Wesley, 2003.

[49] Ultraembedded, "Ultra-Embedded FAT16/32 File IO Library."

[50] Wolper Pierre, "Expressing Interesting Properties of Programs in Propositional Temporal Logic," in *Proceeding of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, 1 1986.

[51] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*. Addison-Wesley Publishing Company, 1 1974.

[52] A. Lindeberg, "Isomorphism Testing of Rooted Trees in Linear Time," 1 2024.

[53] Wikipedia, "MurmurHash," 9 2025.

[54] Austin Appleby, "SMHasher," 1 2016.