EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang

Joint work with Can Sar, Paul Twohey, Ben Pfaff, Dawson Engler and Madan Musuvathi

"Mom, Google Ate My GMail!"

Posted Thursday, December 28, 2006 at 12:19 PM P

Mom, Google Ate My GMail!

Update: If you are ready to give up your
and are considering switching to all-Goo
one of us has done, be warned that the
simple and fraught with risk. An increasi
complaining that their emails, accounts

For nearly 10 da
cleaning out th
wrote to us thi
Just when we
better. *(If you are you one of the victin*
try and get to Google people and see wh
on (what else) Google Threads, a user wr

> Not only we are surprised that these
> company like Google but we are piss
> our internet life.

TechCrunch

Forum About Contact Company Index Advertise Archives Cool Jobs

« Previous post Next post »

December 28 2006

Gmail Disaster: Reports Of Mass Email Deletions

Michael Arrington 133 comments »

Just a week after I wrote "**Uh Oh, Gmail Just Got Perfect**" a number of users started **complaining** that all of their Gmail emails and contacts were auto deleted.

The first message, posted on the Google Groups forum on December 19, stated "*Found my account clean..nothing in Inbox, contacts ,sent mail..How can all these information residing in different folders disappear? ..How to write to gmail help team to restore the account..is it possible?..Where to report this abuse?.Any help ..Welcome..Thanks in advance ps101*"

Flash: Software wings its way to Mars rovers

By Patricia Daukantas, GCN Staff

NASA's twin Mars rovers have been receiving medicinal shots of software over the agency's Deep Space Network.

The updates let Ear … job, said Roger Klemm, a flight software devel …

Last week, Spirit wa … Spirit had fallen into a mysterious commu … memory for the flash file system.

The rover's compute … trying to reboot itself, Klemm said.

Engineers comman … a checkdisk routine. They discovered the …

Each rover has 256 … . Certain blocks are reserved for dedicat … ormat Spirit's flash memory. The progra … om the flash memory of the second rover, O…

Most of the code is written in C, running under the VxWorks 5.3.1 operating system from Wind River Systems Inc. of Alameda, Calif. A few files are in assembly language, and one module is in C++.

Engineers at NASA Langley Research Center in Hampton, Va., adapted a flight-mechanics application, originally developed in the 1970s for planning shuttle missions, to model the complex interactions of the Spirit and Opportunity rovers' hardware and software.

Before the landings, NASA executed multiple simulations of parachute, rover capsule and back shell behavior during entry into the Martian atmosphere, said Eric Queen, a Langley research engineer.

Why check storage systems?

- Storage system errors: some of the most serious
  - **machine crash**
  - **data loss**
  - **data corruption**

- Code complicated, hard to get right
  - Conflicting goals: speed, reliability (recover from any failures and crashes)

- Typical ways to find these errors: ineffective
  - Manual inspection: strenuous, erratic
  - Randomized testing (e.g. unplug the power cord): blindly throwing darts
  - Error report from mad users

Goal: build tools to automatically find storage system errors

Sub-goal: comprehensive, lightweight, general

EXPLODE [OSDI06]

- Comprehensive: adapt ideas from model checking
- General, real: check live systems
  - Can run (on Linux, BSD), can check, even w/o source code
- Fast, easy
  - Check a new storage system: 200 lines of C++ code
  - Port to a new OS: 1 kernel module + optional modification
- Effective
  - 17 storage systems: 10 Linux FS, Linux NFS, Soft-RAID, 3 version control, Berkeley DB, VMware
  - Found serious data-loss in all
- Subsumes FiSC [OSDI04, best paper]

Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable
- Results

Long-lived bug fixed in 2 days in the IBM Journaling file system (JFS)

- Serious
  - Loss of an entire FS!
  - Fixed in 2 days with our complete trace
- Hard to find
  - 3 years old, ever since the first version

Dave Kleikamp (IBM JFS): “I really appreciate your work finding and recreating this bug. I'm sure this has bitten us before, but it's usually hard to go back and find out what causes the file system to get messed up so bad”

Events to trigger the JFS bug

creat("/f");

fl "/f"

5-char system call, not a typo

fsck.jfs

File system recovery utility, run after reboot

Buffer Cache (in mem)

Disk

/

Orphan file removed. Legal behavior for file systems

Events to trigger the JFS bug

creat("/f");

bug under low mem (design flaw)

flush "/"

crash!

fsck.jfs

File system recovery utility, run after reboot

Buffer Cache (in mem)

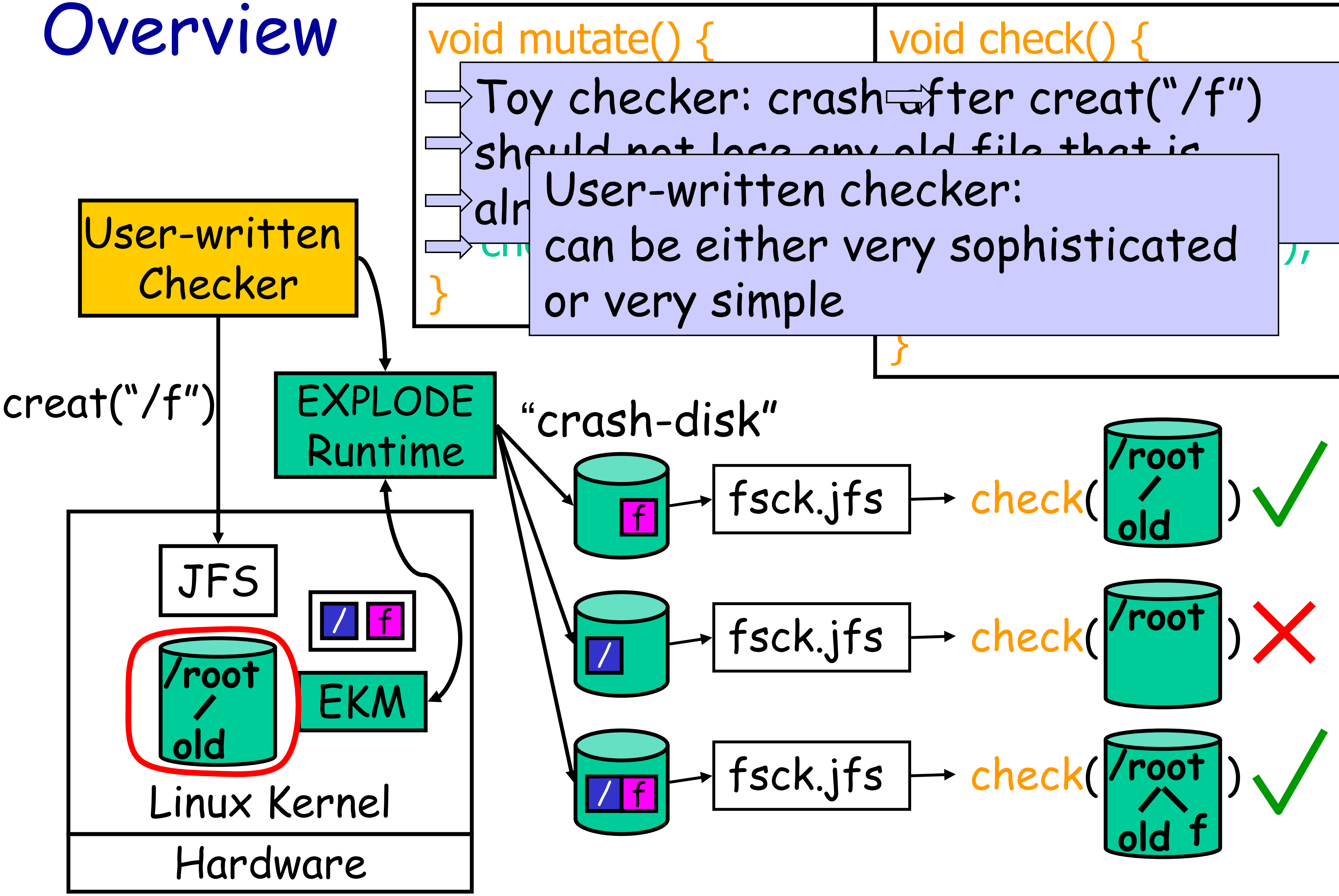
Disk

/

dangling pointer!

"fix" by zeroing, entire FS gone!

Overview



User code   Our code   EKM = EXPLODE kernel module

Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable
- Results

One core idea from model checking: explore all choices

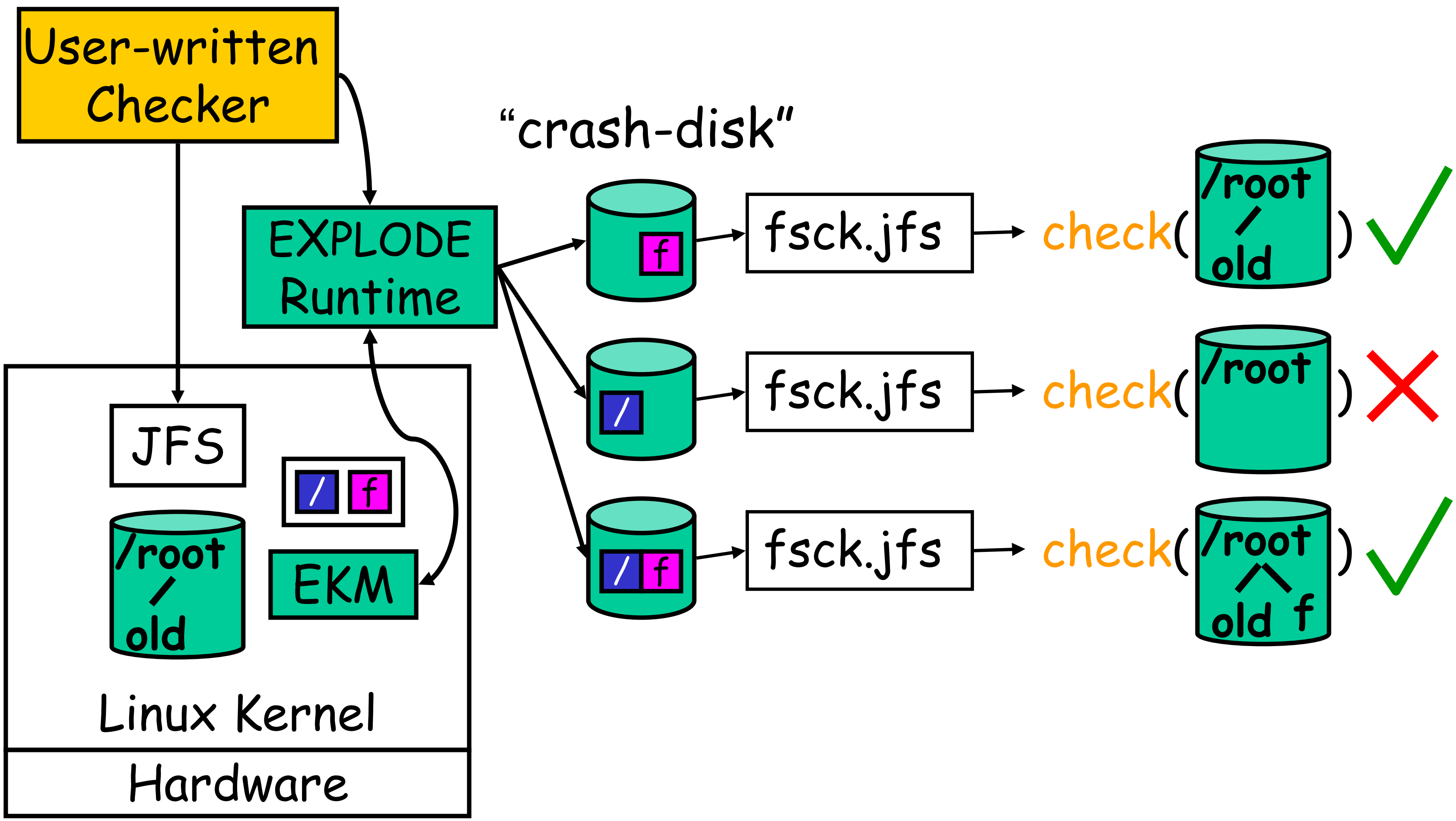
- Bugs are often triggered by corner cases
- How to find? Drive execution down to these tricky corner cases

Principle

When execution reaches a point in program that can do one of N different actions, fork execution and in first child do first action, in second do second, etc.

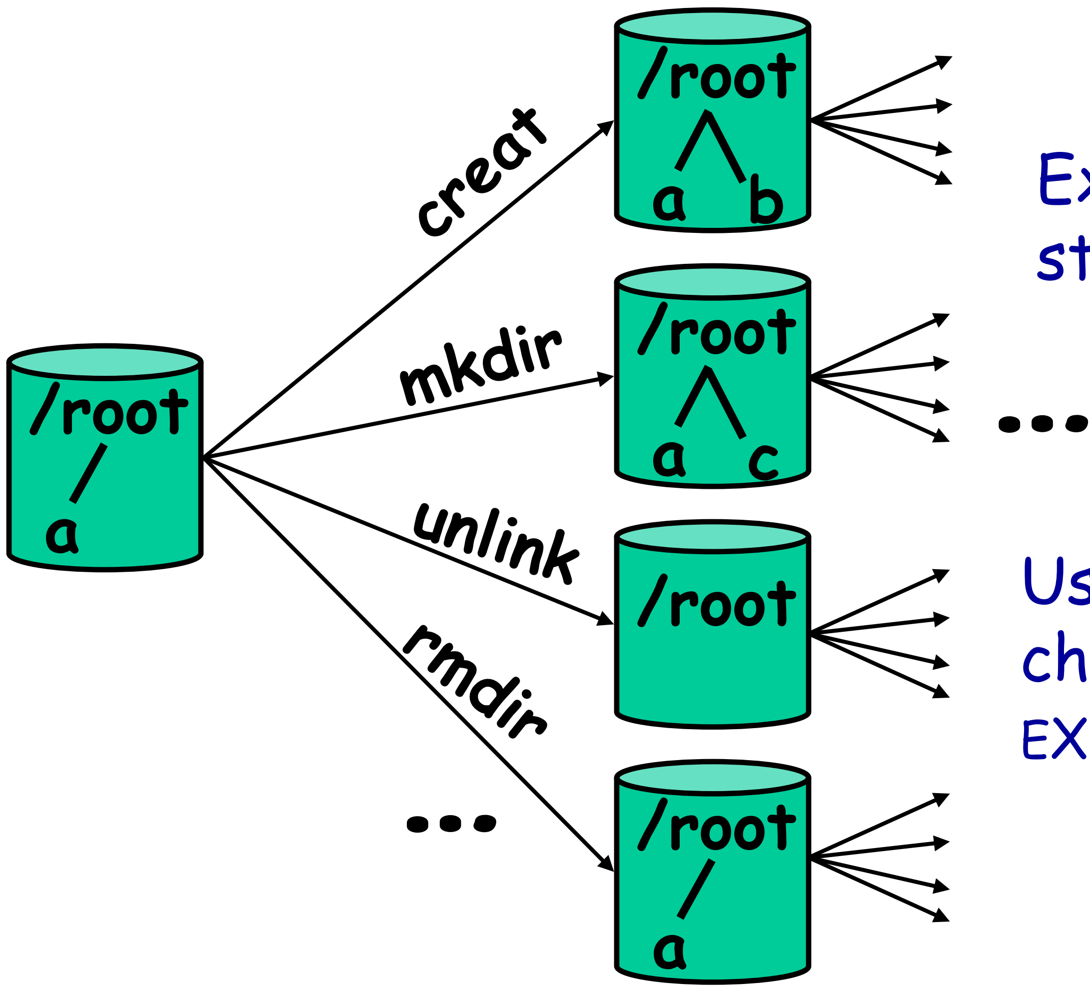
Result: rare events appear as often as common ones

Crashes (Overview slide revisit)



External choices

- Fork and do every possible operation

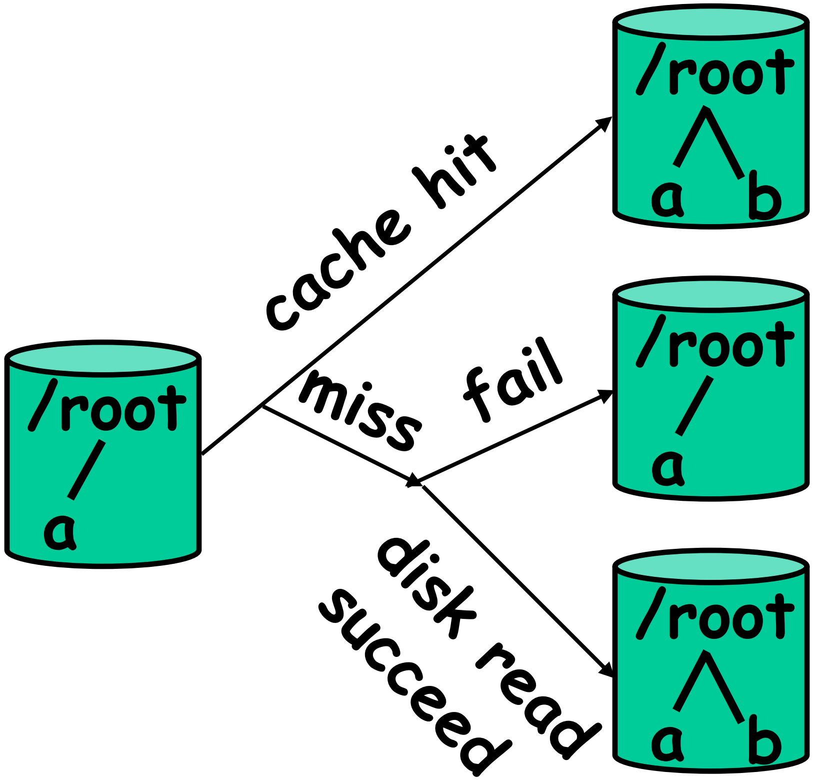


Explore generated states as well

Users write code to check FS valid.
EXPLODE "amplifies"

Internal choices

- Fork and explore all internal choices




```
struct block* read_block (int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        return b;
    return disk_read (i);
}
```

Users expose choices using choose(N)

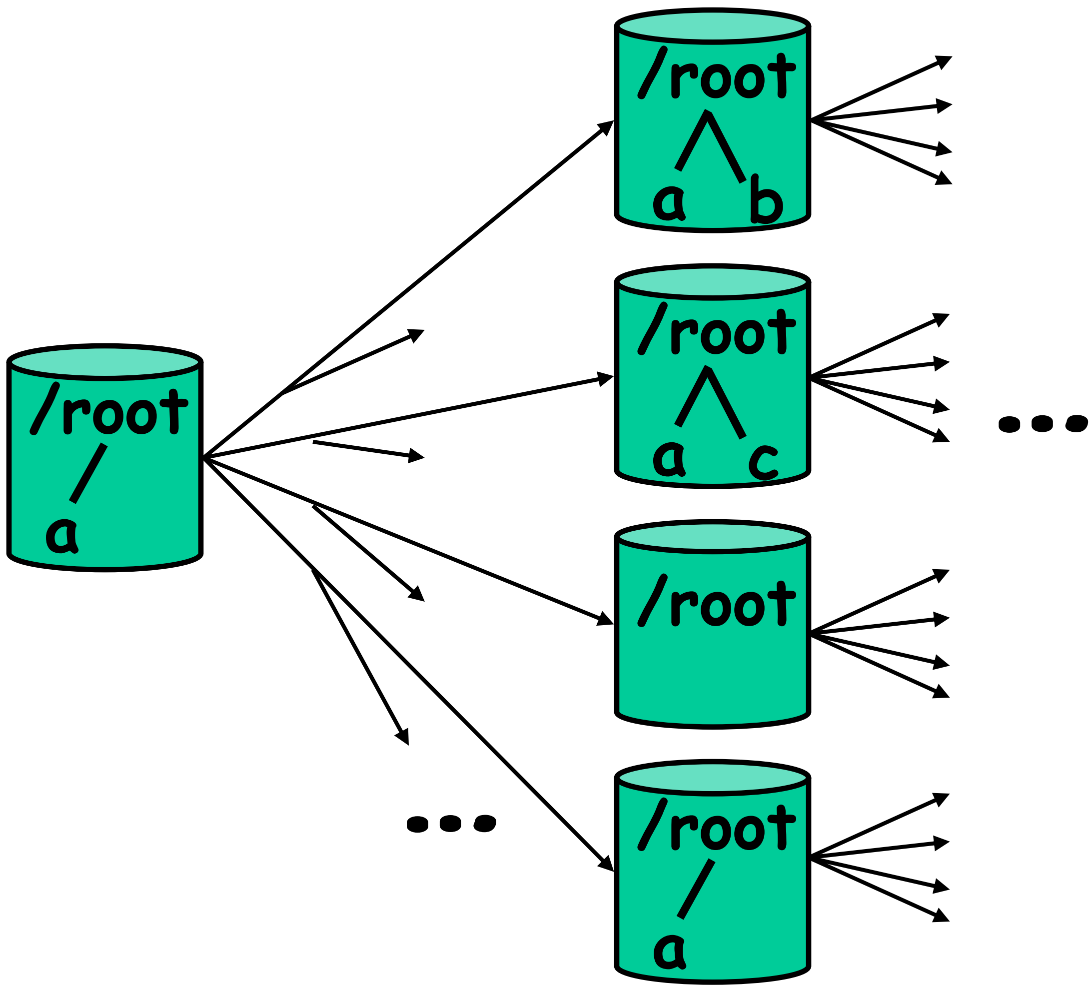
- To explore N-choice point, users instrument code using choose(N) (also used in other model checkers)
- choose(N): N-way fork, return K in K'th kid

```
struct block* read_block (int i) {
    struct block *b;
    if ((b = cache_lookup(i)))
        return b;
    return disk_read (i);
}
```

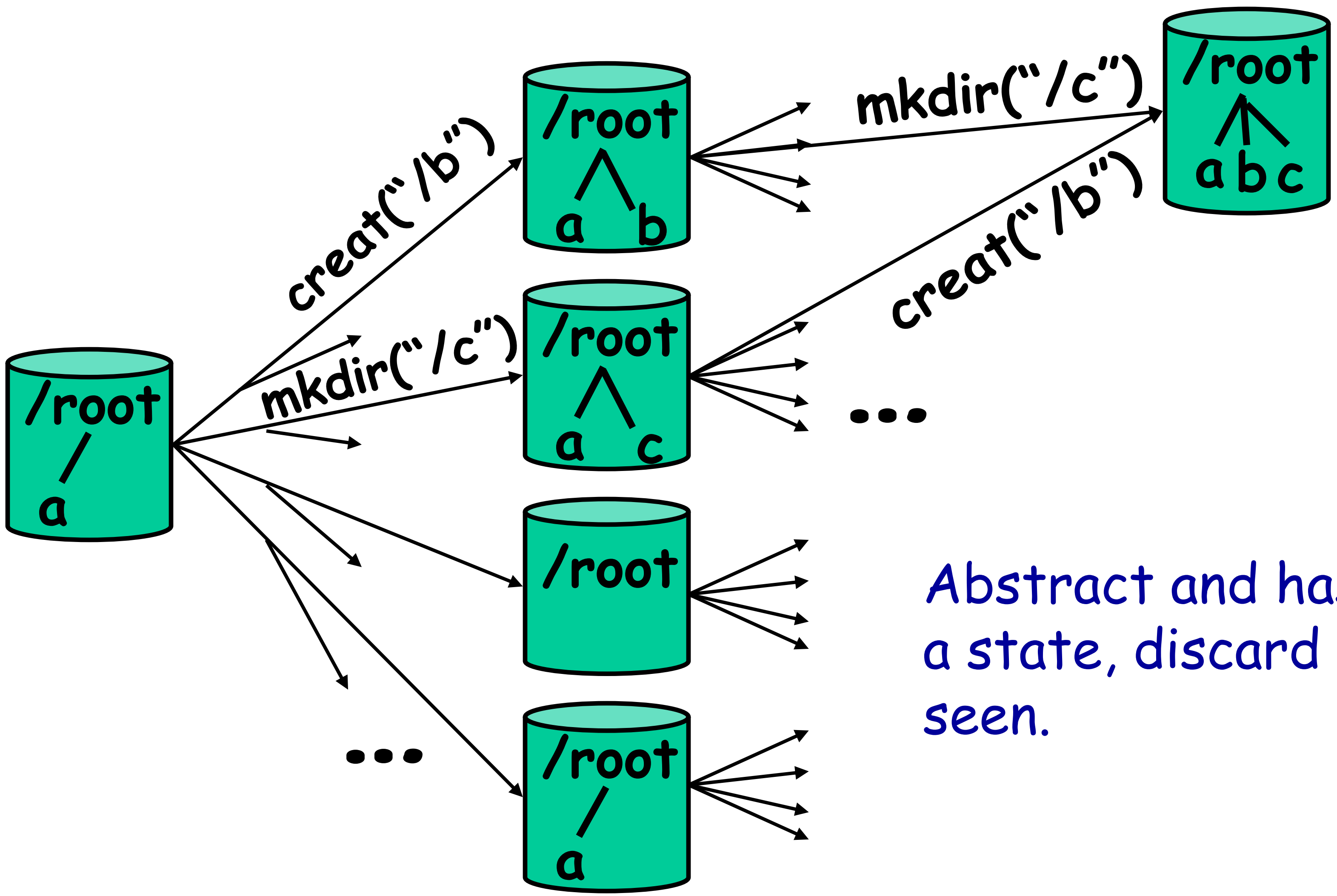
```
cache_lookup (int i) {
    if(choose(2) == 0)
        return NULL;
    // normal lookup
    …
}
```

- Optional. Instrumented only 7 places in Linux

Crash X External X Internal



Speed: skip same states

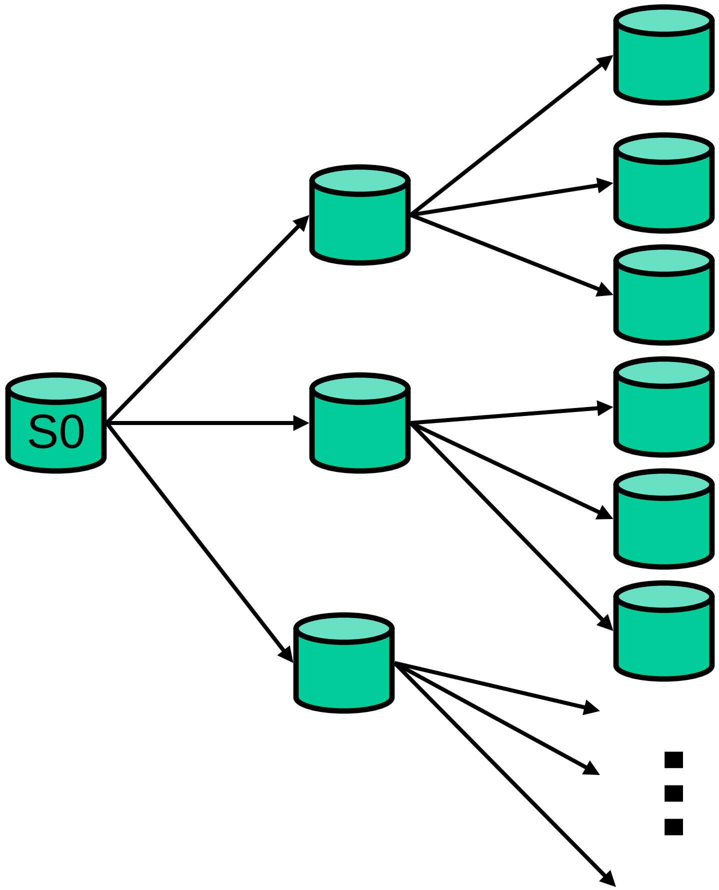


Abstract and hash a state, discard if seen.

Outline

- Overview
- Checking process
- Implementation
  - FiSC, File System Checker, [OSDI04], best paper
  - EXPLODE, storage system checker, [OSDI06]
- Example check: crashes during recovery are recoverable
- Results

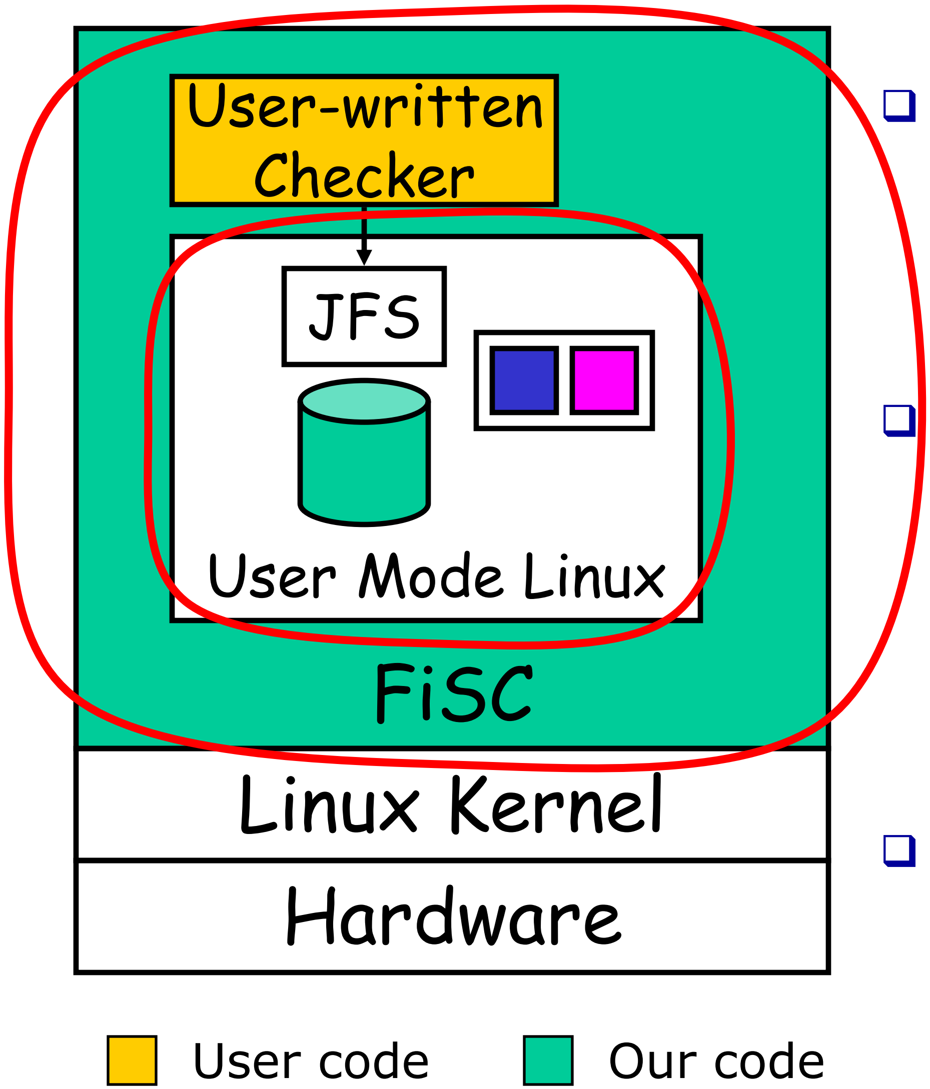
Checking process




```
S0 = checkpoint()
enqueue(S0)
while(queue not empty){
    S = dequeue()
    for each action in S {
      restore(S)
      do action
      S' = checkpoint()
      if(S' is new)
            enqueue(S')
    }
}
```

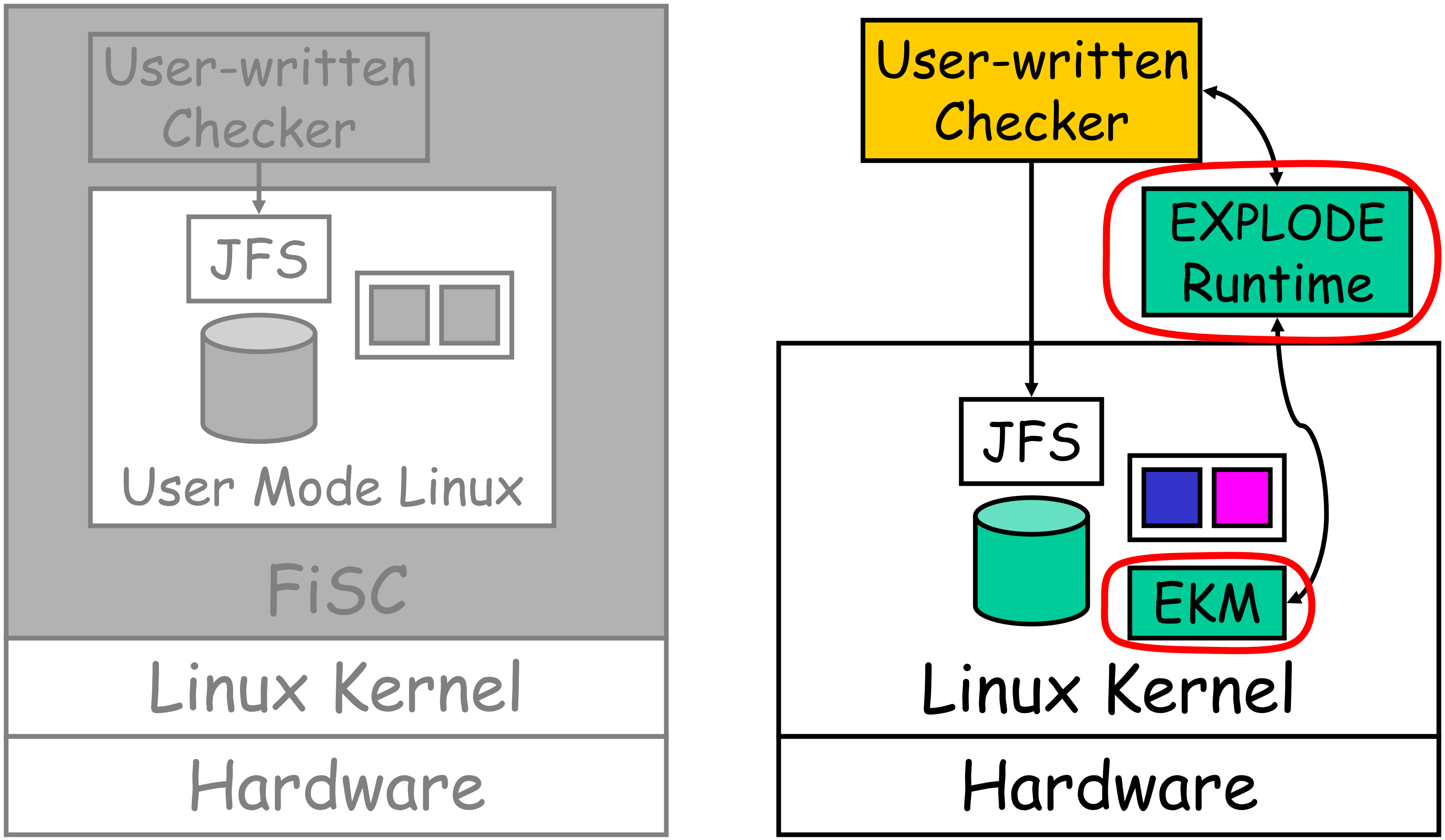
How to checkpoint and restore a live OS?

FiSC: jam OS into tool



- Pros
  - Comprehensive, effective
  - No model, check code
  - Checkpoint and restore: easy
- Cons
  - Intrusive. Build fake environment. Hard to check anything new. Months for new OS, 1 week for new FS
- Many tricks, so complicated that we won best paper OSDI 04

EXPLODE: jam tool into OS



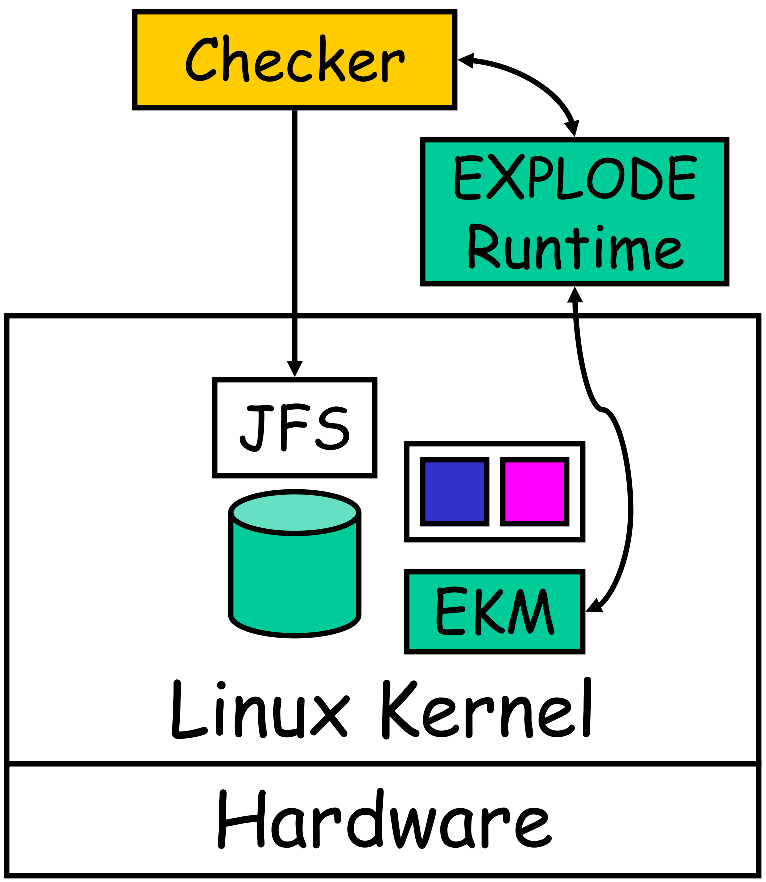
User code   Our code   EKM = EXPLODE kernel module

EKM lines of code

OS	Lines of code
Linux 2.6	1,915
FreeBSD 6.0	1,210

EXPLODE kernel modules (EKM) are small and easy to write

How to checkpoint and restore a live OS kernel?



- Hard to checkpoint **live** kernel memory
- Virtual machine? No
  - VMware: no source
  - Xen: not portable
  - heavyweight
- There's a better solution for storage systems

Checkpoint: save actions instead of bits

state = list of actions
checkpoint S = save (creat, cache miss)
restore = re-initialize, creat, cache miss

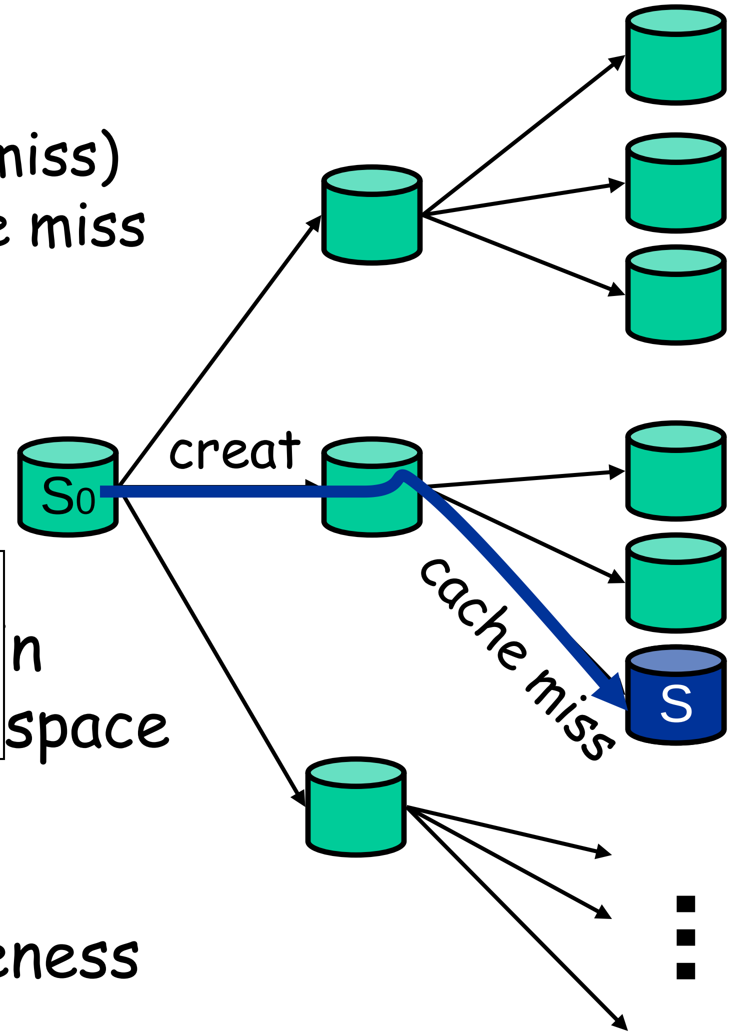
re-initialize = unmount, mkfs

Utility that clears in-mem state of a storage system

Utility to create an empty FS

tate n
trac space
(stateless search).

We use it only to reduce intrusiveness



Deterministic replay

- Storage system: isolated subsystem

- Non-deterministic kernel scheduling decision
  - Opportunistic fix: priorities
- Non-deterministic interrupt
  - Fix: use RAM disks, no interrupt for checked system
- Non-deterministic kernel choose() calls by other code
  - Fix: filter by thread IDs. No choose() in interrupt

- Worked well in practice
  - Mostly deterministic
  - Worst case: auto-detect & ignore non-repeatable errors

Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable
- Results

Why check crashes during recovery?

- Crashes are highly correlated
  - Often caused by kernel bugs, hardware errors
  - Reboot, hit same bug/error

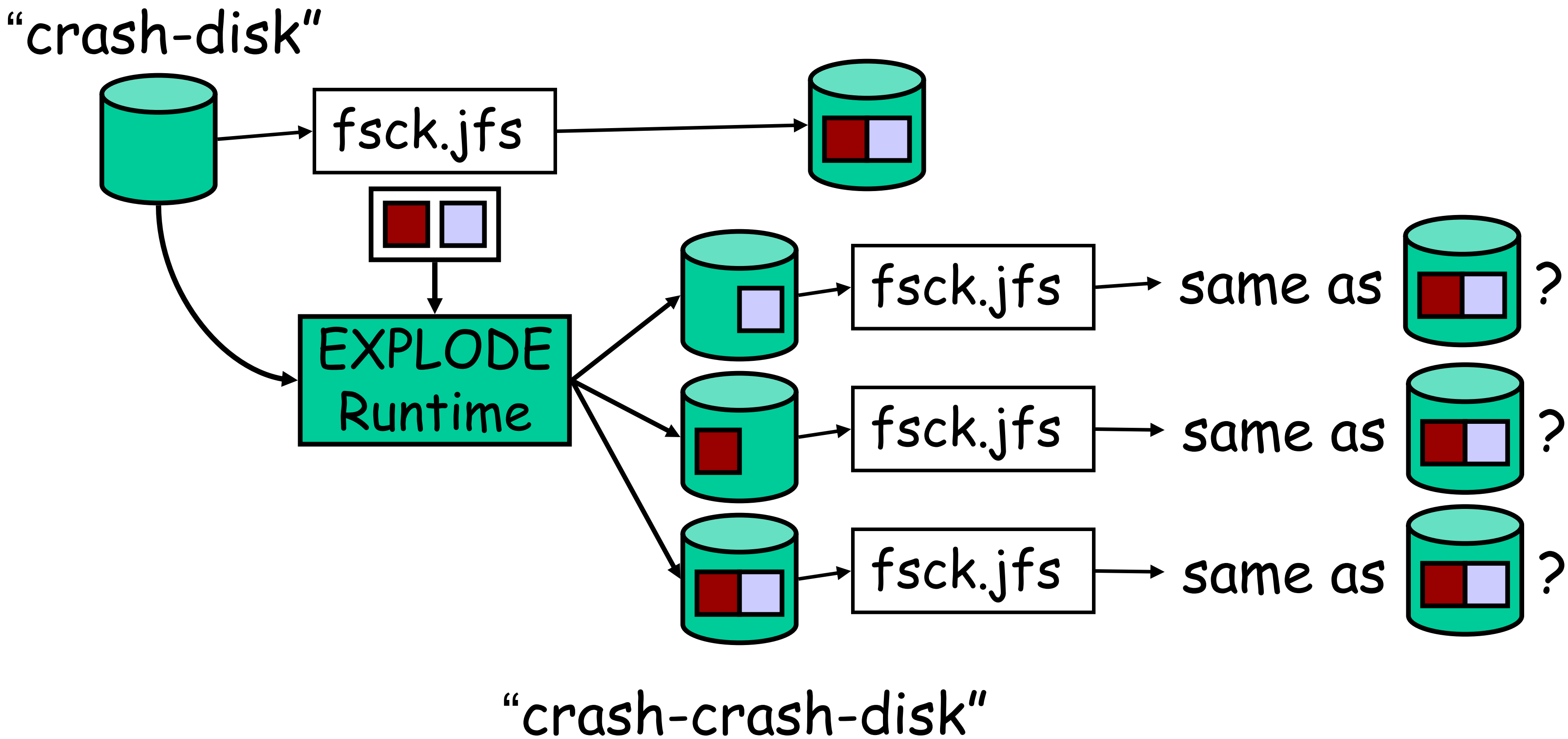
What to check?

- fsck once == fsck & crash, re-run fsck
  - fsck(crash-disk) to completion, "/a" recovered
  - fsck(crash-disk) and crash, fsck, "/a" gone

  Bug!

- Powerful heuristic, found interesting bugs (wait until results)

How to check crashes during recovery?



Problem: N blocks → 2^N crash-crash-disks.
Too many! Can prune many crash-crash-disks

Simplified example

- 3-block disk, B1, B2, B3
- each block is either 0 or 1
- crash-disk = 000 (B1 to B3)

fsck(000)

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

buffer cache: B2=1

buffer cache: B2=1, B3=1

buffer cache: B2=1, B3=1, B1=1

fsck(000) = 111

Naïve strategy: 7 crash-crash-disks

crash-disk = 000

fsck(000) = 111

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

buffer cache: B2=1, B3=1, B1=1

000 + {B2=1}

fsck(010) == 111?

fsck(001) == 111?

fsck(011) == 111?

fsck(100) == 111?

fsck(110) == 111?

fsck(101) == 111?

fsck(111) == 111?

Optimization: exploiting determinism

crash-disk = 000

fsck(000) = 111

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

- For all practical purposes, fsck is deterministic
  - read same blocks → write same blocks

000 + {B2=1}

- fsck(010) == 111?
- fsck(000) doesn't read B2
- So, fsck(010) = 111

What blocks does fsck(000) actually read?

crash-disk = 000

fsck(000) = 111

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

Read of B3 will get what we just wrote. Can't depend on B3

fsck(000) reads/depends only on B1. It doesn't matter what we write to the other blocks.

fsck(0**) = 111

Prune crash-crash-disks matching 0**

crash-disk = 000

fsck(000) = 111

Read(B1) = 0
Write(B2, 1)
Write(B3, 1)
Read(B3) = 1
Write(B1, 1)

buffer cache: B2=1, B3=1, B1=1

~~fsck(010) == 111?~~

~~fsck(001) == 111?~~

~~fsck(011) == 111?~~

fsck(100) == 111?

fsck(110) == 111?

fsck(101) == 111?

fsck(111) == 111?

Can further optimize using this and other ideas

Outline

- Overview
- Checking process
- Implementation
- Example check: crashes during recovery are recoverable
- Results

Bugs caused by crashes during recovery

- Found data-loss bugs in all three FS that use logging (ext3, JFS, ReiserFS), total 5

- Strict order under normal operation:
  - First, write operation to log, commit
  - Second, apply operation to actual file system

- Strict (reverse) order during recovery:
  - First, replay log to patch actual file system
  - Second, clear log
  - No order → corrupted FS and no log to patch it!

Bug in fsck.ext3

```
recover_ext3_journal(...) {
    // ...
    retval = -journal_recover(journal);
    // ...
    // clear the journal
    e2fsck_journal_release(...)
    // ...
}
```

```
journal_recover(...) {
    // replay the journal
    //...
    // sync modifications to disk
    fsync_no_super (...)
}
```

```
// Error! Empty macro, doesn't sync data!
#define fsync_no_super(dev)          do {} while (0)
```

- Code directly adapted from the kernel
- But, fsync_no_super defined as NOP: "hard to implement"

FiSC Results (can reproduce in EXPLODE)

Error Type	VFS	ext2	ext3	JFS	ReiserFS	total
Data loss	N/A	N/A	1	8	1	10
False clean	N/A	N/A	1	1		2
Security		2	2	1		3 + 2
Crashes	1			10	1	12
Other	1		1	1		3
Total	2	2	5	21	2	32

32 in total, 21 fixed, 9 of the remaining 11 confirmed

EXPLODE checkers lines of code and errors found

Storage System Checked		Checker	Bugs
10 file systems		5,477	18
Storage applications	CVS	68	1
	Subversion	69	1
	"EXPENSIVE"	124	3
	Berkeley DB	202	6
Transparent subsystems	RAID	FS + 137	2
	NFS	FS	4
	VMware GSX/Linux	FS	1
Total		6,008	36

6 bugs per 1,000 lines of checker code

Related work

- FS Testing
- Static (compile-time) analysis
- Software model checking

Conclusion

- EXPLODE
  - Comprehensive: adapt ideas from model checking
  - General, real: check live systems in situ, w/o source code
  - Fast, easy: simple C++ checking interface

- Results
  - Checked 17 widely-used, well-tested, real-world storage systems: 10 Linux FS, Linux NFS, Soft-RAID, 3 version control, Berkeley DB, VMware
  - Found serious data-loss bugs in all, over 70 bugs in total
  - Many bug reports led to immediate kernel patches