Recon - weryfikacja poprawności systemu plików podczas jego działania

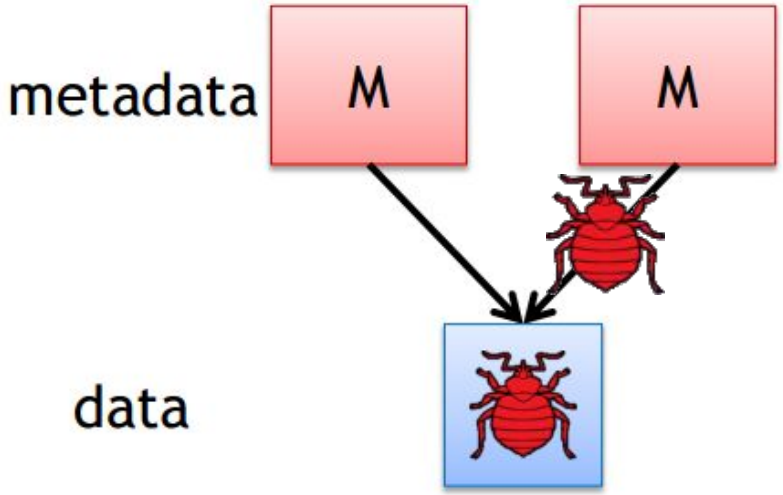
autor: Andrzej Jackowski

Popularne metody zapewnienia niezawodności systemu plików

1. Księgowanie (journal)
   a. Kopiowanie podczas zapisu (copy on write)
   b. Lekkie aktualizacje (soft updates)
2. Sumy kontrolne, kody korekcyjne
3. RAID
4. Replikacja
5. Powyższe metody pomagają nam unikać błędów które dzieją się poniżej systemu plików

Kiedy to za mało?

1. Weźmy system plików w którym jeden blok z metadanymi wskazuje na jeden blok z danymi
2. Jeden blok wskazuje na dane które do niego należą
3. Drugi blok na skutek błędu w kodzie systemu plików (np. korupcja pamięci, wyścig) również wskazuje na ten sam blok
4. W rezultacie blok z danymi zawiera uszkodzone dane
5. Te same uszkodzone dane mamy w dzienniku, na wszystkich dyskach macierzy RAID, na replice itd.
6. O problemie możemy dowiedzieć o wiele za późno (np. przy próbie odczytu)
7. Problem może mieć jeszcze bardziej przykre konsekwencje, gdy np. zaczniemy pisać danymi użytkownika po blokach z metadanymi.



Czy powinniśmy się tym przejmować?

1. Systemy plików są zazwyczaj powszechnie używane, wiec są dobrze przetestowane i stabilne
   a. ext2/ext3/ext4 - 1993/2001/2008
   b. ReiserFS / Reiser4 - 2001/2004
   c. btrfs - unstable 2009 / stable 2013

W systemach plików są błędy!

FS	Source	Bug Title	Closed
ext3	http://lwn.net/Articles/2663/	ext3 corruption fix	2002-06
ext3	kerneltrap.org/node/515	Linux: Data corrupting ext3 bug in 2.4.20	2002-12
ext3	Redhat, #311301	panic/ext3 fs corruption with RHEL4-U6-re20070927.0	2007-11
ext3	https://lkml.org/lkml/2008/12/6/88	Re: [2.6.27] filesystem (ext3) corruption (access beyond end)	2008-06
ext3	Debian, #425534	linux-2.6: ext3 filesystem corruption	2008-09
ext3	Debian, #533616	linux-image-2.6.29-2-amd64: occasional ext3 filesystem corruption	2009-06
ext3	Redhat, #515529	ENOSPC during fsstress leads to filesystem corruption on ext2, ext3, and ext4	2010-03
ext3	https://lkml.org/lkml/2011/6/16/99	ext3: Fix fs corruption when make_indexed_dir() fails	2011-06
ext3	Redhat, #658391	Data corruption: resume from hibernate always ends up with EXT3 fs errors	Not yet
btrfs	https://lkml.org/lkml/2009/8/21/45	btrfs rb corruption fix	2009-08
btrfs	https://lkml.org/lkml/2010/2/25/376	[2.6.33 regression] btrfs mount causes memory corruption	2010-02
btrfs	https://lkml.org/lkml/2010/11/8/248	DM-CRYPT: Scale to multiple CPUs v3 on 2.6.37-rc* ?	2010-09
btrfs	https://lkml.org/lkml/2011/2/9/172	[PATCH] btrfs: prevent heap corruption in btrfs_ioctl_space_info()	2011-02
btrfs	https://lkml.org/lkml/2011/4/26/304	btrfs updates (slab corruption in btrfs fitrim support)	2011-04

Niedawny błąd

1. W maju tego roku był poważny błąd związany z systemem plików EXT4 (Red Hat Bug #1223332, “corruption caused by unwritten and delayed extents”)
2. Dotyczył on kernela w wersji 4.0.2, który był w tym czasie używany m.in. w Fedorze i Archu
3. Błąd powodował nadpisanie zerami istniejącego bloku z danymi
4. Przeglądając komentarze do powyższego błędu na Red Hat Bug można znaleźć odnośniki do kolejnych błędów związanych z EXT4 i kernelem 4.0, wiec należy zaakceptować, że błędy w systemach plików po prostu są.

Znane rozwiązanie - FSCK

1. FSCK to znane narzędzie pozwalające na weryfikację stanu metadanych systemu plików
2. Wymaga odmontowania systemu plików
    a. Jeśli zapisujemy dane do systemu plików, to jego stan się zmienia, więc fsck może zwracać informację o błędach tam gdzie ich nie ma
    b. Jeśli pozwolimy fsck na naprawianie błędów w takie sytuacji, to możemy być pewni, że popsujemy sobie system plików
    c. Jeśli mamy zamontowany system plików jako read-only, to sprawdzenie jego stanu przy użyciu fsck ma sens, ale jeśli zaczniemy go reperować, to możemy odczytywać uszkodzone dane

*“If **e2fsck** asks whether or not you should check a filesystem which is mounted, the only correct answer is "no". Only experts who really know what they are doing should consider answering this question in any other way.”* - http://linux.die.net/man/8/e2fsck

Znane rozwiązanie - FSCK

3. Jest wolne
    a. W zależności systemu plików, ilości danych i błędów może trwać godziny, albo nawet kilka dni
    b. Nie wiadomo ile będzie trwać, bo nie wiadomo jak dużo plików musimy naprawić
    c. Będzie trwało jeszcze dłużej (obrazek po prawej)
4. Może okazać się, że w momencie użycia FSCK jest za późno - dane użytkownika są bezpowrotnie uszkodzone
5. Ogólne podejście - używać systemu pliku tak długo jak działa, w razie problemów wgrywać kopię zapasową

	2006	2009	2013	Change
Capacity (GB)	500	2000	8000	16x
Bandwidth (Mb/s)	1000	2000	5000	5x
Seek time (ms)	8	7.2	6.5	1.2x

Pojemność dysków vs ich szybkość

Znane rozwiązanie - kopia zapasowa

1. Znowu długo - wymaga **przynajmniej** fizycznej wymiany nośników, w praktyce dane trzeba skopiować z dysku na dysk
2. Kopia zapasowa nie zawiera najnowszych danych
3. Kopia zapasowa może zawierać błędy, jeśli problemy z systemem plików pojawiły się przed jej utworzeniem

Nowe rozwiązanie - system Recon

1. Dodajmy warstwę pośrednią między systemem plików i dyskami twardymi
2. Utwórzmy zbiór **niezmienników**, który zapewnia, że wszystkie weryfikacje wykonywane przez fsck potwierdzą poprawność systemu plików
3. Przed każdą modyfikacją metadanych sprawdźmy (w dodanej warstwie pośredniej), czy zmieniony system plików ciągle spełnia swoje niezmienniki
4. Jeśli modyfikacja nie popsuje systemu plików, to ją zapisujemy

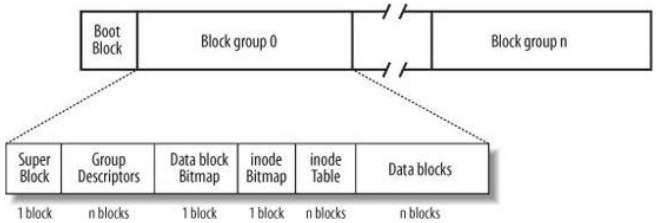
Przykład niezmiennika

“block pointer ustawiamy z 0 na N ↔ zapalimy bit N w Data block Bitmap”

Zmiana metadanych w formacie:

[Typ, ID, Pole, Stara wartość, Nowa wartość]

Figure 18-1. Layouts of an Ext2 partition and of an Ext2 block group



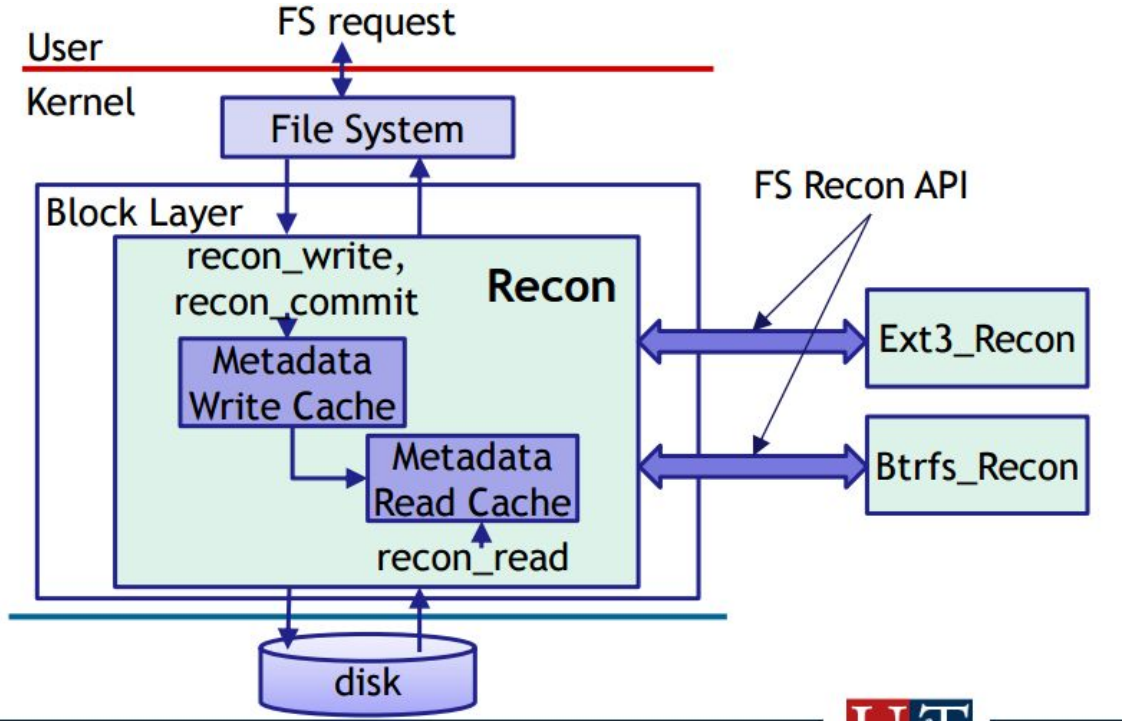
Zalety niezmienników

1. Wiemy o uszkodzeniu systemu plików zanim błędne dane trafią na dysk
2. Musimy sprawdzać jedynie wycinek danych ulegający zmianie i jego okolice, a nie cały system plików
3. System przez cały czas może działać (nie musimy go odmontowywać)

Co jeśli niezmiennik nie zostaje spełniony?

1. W zależności od tego do czego używamy systemu możemy obsłużyć błąd w dowolny sposób, przykładowo:
    a. Zalogować błąd/wysłać maila do administratora i działać dalej
    b. Natychmiast przerwać wszystkie zapisy i przejść w tryb read-only
    c. Odmontować system plików
    d. Spróbować naprawić metadane (tak jak to robi fsck)
    e. Zrobić migawkę (snapshot) systemu plików i pracować dalej
    f. Zrestartować urządzenie
    g. ...

Architektura systemu Recon



Kiedy wykonywać sprawdzenie?

1. Metadane dostarczane przez system plików mogą być chwilowo niepoprawne - możemy dostawać informacje o zmianie metadanych w kilku paczkach
2. Na szczęście współczesne systemy plików zazwyczaj operują na transakcjach, które są potrzebne do realizacji dziennika lub kopiowania przy zapisie
3. Możemy więc przechowywać zmiany w pamięci i weryfikować ich poprawność jedynie podczas operacji zatwierdzenia transakcji

Wykonywanie sprawdzenia - problemy w EXT3

1. Transakcje w EXT3 mogą mieć następujący scenariusz:
   a. Zapisz metadane do dziennika
   b. Zatwierdź transakcje (zapisz *commit block*)
   c. Skopiuj metadane z dziennika na dysk
   d. Usuń metadane z dziennika
2. Jeśli wykonamy sprawdzenie przed punktem **b**, to dane mogą zostać uszkodzone podczas kopiowania w punkcie **c**
3. Dlatego musimy przeprowadzić weryfikację dwukrotnie - podczas zatwierdzenia transakcji i podczas kopiowania metadanych z dziennika

Jak odczytywać metadane?

1. Zauważmy, że system plików przysyła do Recona prośby o odczyty / zapisy plików bez określania jakiego typu danych one dotyczą
2. Możemy jednak sobie z tym poradzić, jeśli znamy zachowanie systemu plików
3. Używamy więc reguł w stylu “blok ze wskaźnikami wczytujemy zanim wczytamy dane spod wskaźnika”, aby określić typ operacji
4. Dzięki temu faktycznie możemy sobie stworzyć bazę z opisem zmian metadanych dla każdej transakcji

Przykład niezmiennika

“block pointer ustawiamy z 0 na N ↔ zapalimy bit N w Data block Bitmap”

Zmiana metadanych w formacie:

[Typ, ID, Pole, Stara wartość, Nowa wartość]

Luka w odczytywaniu metadanych

1. Załóżmy, że użytkownik chciał wykonać następujący zapis:
   a. Zapisz dane do bloku 22345
2. Z powodu błędu w systemie plików do Recona przyszło jednak co innego:
   a. Zapisz dane do bloku 12345
3. W bloku 12345 nieszczęśliwie znajdują się metadane (np. wskaźnik na blok umieszczony w i-węźle)
4. Recon nie rozpozna, że jest to blok z metadanymi - nie zaszła reguła “po dodaniu wskaźnika na blok do i-węzła pojawi się zapis do podanego bloku”
5. W niektórych systemach plików ten problem nie jest możliwy, w innych niewiele można z tym zrobić (chociaż można np. dodać kod do systemu plików, który umożliwi identyfikacje problemowych bloków)

Oprogramowanie do sprawdzania systemu plików też może mieć błędy!

1. Kod e2fsck ma prawie 30,000 linii
2. Kod ext2 ma 10,000 linii, a ext4 ma 50,000 linii
3. Oprogramowanie do sprawdzania systemu plików może mieć więc tyle samo błędów, co sam system plików
4. Jaki jest więc sens go używać?

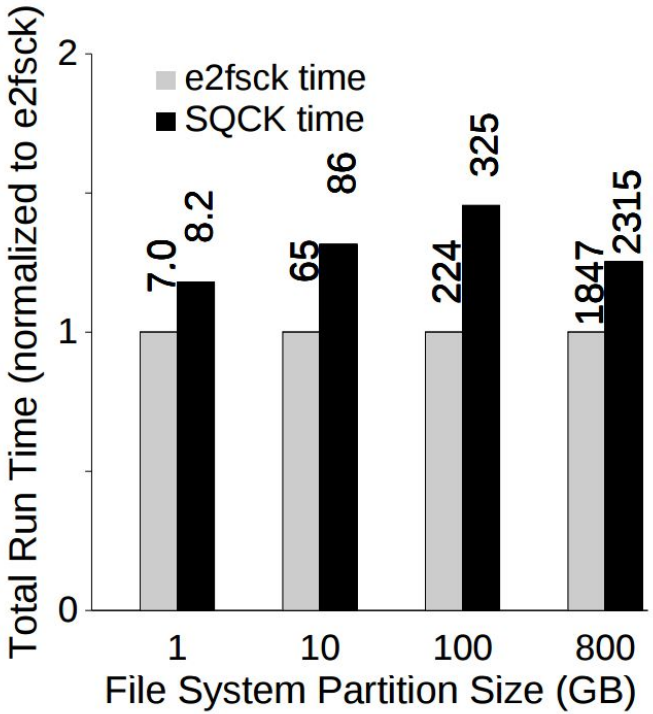
SQCK - fsck można napisać lepiej!

1. Zamiast utrzymywać ciężki w zrozumieniu kod napisany w C, zapiszmy warunki gwarantujące poprawność systemu plików w języku deklaratywnym
2. Zgrajmy więc w całości metadane systemu plików do relacyjnej bazy danych (np. MySQL)
3. Następnie zapiszmy kod weryfikujący poprawność systemu plików w postaci wielu krótkich zapytań do bazy danych
4. Zapisanie wszystkich weryfikacji z e2fsck zajęło około 1100 linii SQL, rozbitego na 121 zapytań
5. Dla porównania implementacja w fsck:
    a. Fazy 1. zajmuje 4000 linii (najdłuższa funkcja 957 linii)
    b. Fazy 2. 1945 linii (najdłuższa funkcja 699 linii)

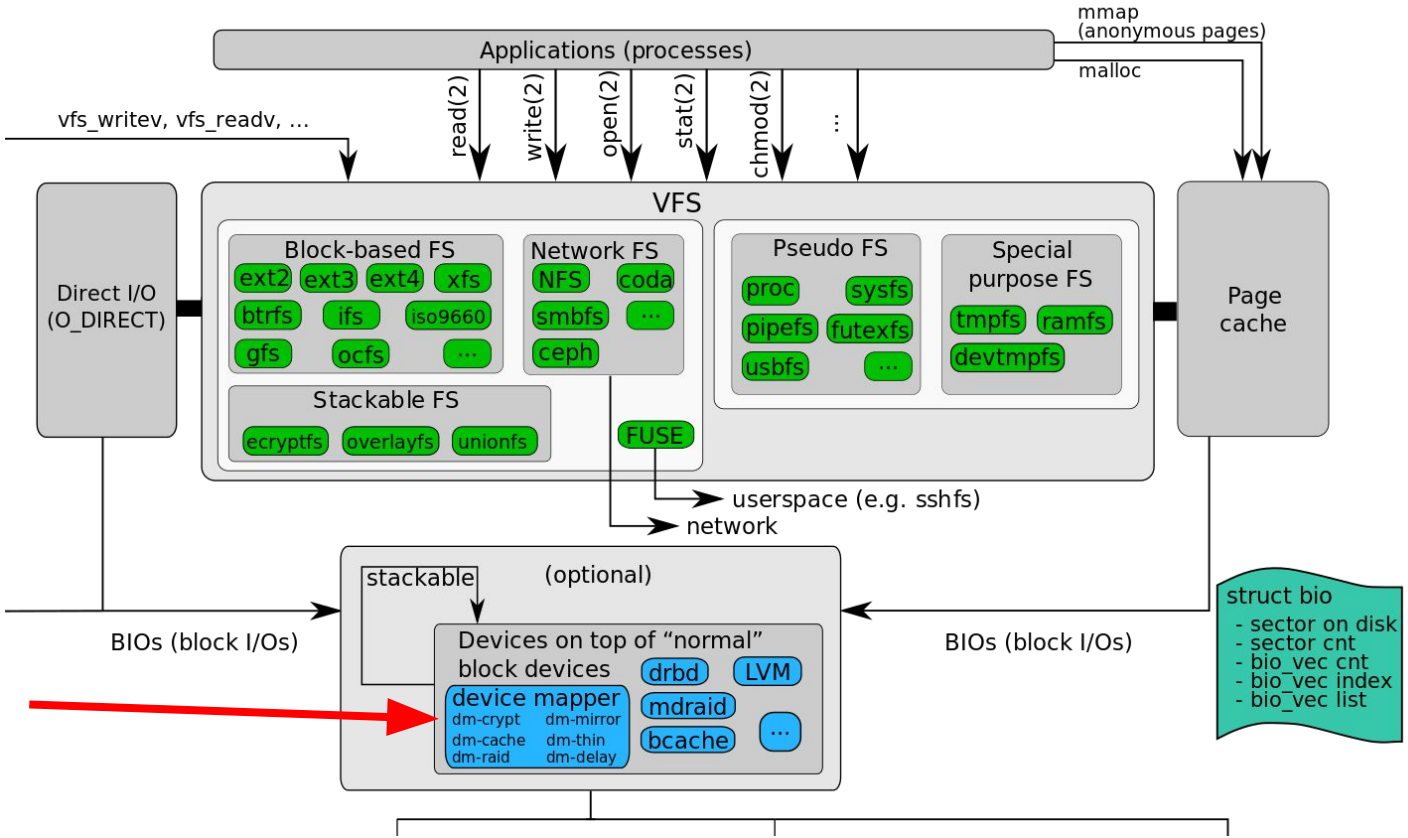
#	Checks Performed
28	**Phase 0: Check consistency in the superblock**
23	*Field check:* Check all superblock fields (*e.g.*, fs size, inode count, groups count, mount/write time)
3	*Range check:* Ensure pointers to block bitmap, inode bitmap, inode table are in the group
2	*Special feature:* Check resize inode feature
35	**Phase 1: Scan and check inodes and block pointers**
9	*Bad block:* Check fields of bad-block inode; ensure superblocks and group descriptors in healthy blocks
18	*Inode structure:* Check fields (*e.g.*, mode, time, size) of different inodes (*e.g.*, root, reserved, boot load)
1	*Range check:* Ensure direct and indirect pointers point within the file system
7	*Conflicts:* Ensure no conflict among block pointers (*e.g.*, two inodes should not share a block)
38	**Phase 2: Scan and check all directory entries**
16	*Directory:* Check each has '.' and '..' entry, '.' points to itself, does not have missing block, fields of dir inode consistent (*e.g.*, acl, fragment size)
9	*Dir Entry:* Check each entry has correct name length, each points to an in-range inode, record length valid, filename contains legal characters
5	*Pathnames:* Each entry points to used inode, does not point to self, does not point to inode in bad block, does not point to root, dir has only one parent
8	*Special inodes:* Check device inodes and symlinks
6	**Phase 3: Ensure all directories are connected to the file system tree**
3	*lost+found*: Ensure lost+found directory is valid and ready to be populated
3	*Reattach*: Reattach orphan directory to lost+found
3	**Phase 4: Fix reference counts and reattach zero-linked file to lost+found**
11	**Phase 5: Check block and inode bitmaps against on-disk bitmaps**
121	**Total**

SQCK - fsck można napisać lepiej!

6. Po zastosowaniu wielu optymalizacji SQCK okazało się niemalże równie szybkie co e2fsck
7. Twórcy Recona postanowili wykorzystać pomysł twórców SQCK - do napisania niezmienników wykorzystali opracowane do SQCK zapytania



Implementacja - Recon to device mapper



Recon

Implementacja - Recon to device mapper

1. Device mapper otrzymuje zapytania o wykonanie operacji w postaci struktury `struct bio`
2. `struct bio` zawiera między innymi:
   a. Informację, czy chcemy wykonać odczyt czy zapis
   b. Wskaźnik na `struct block_device`
   c. Sektor w którym chcemy wykonać operację
3. Na podstawie `struct` bio i naszych założeń (o czym było wcześniej) jakie metadane są zmieniane

Rezultaty - implementacja

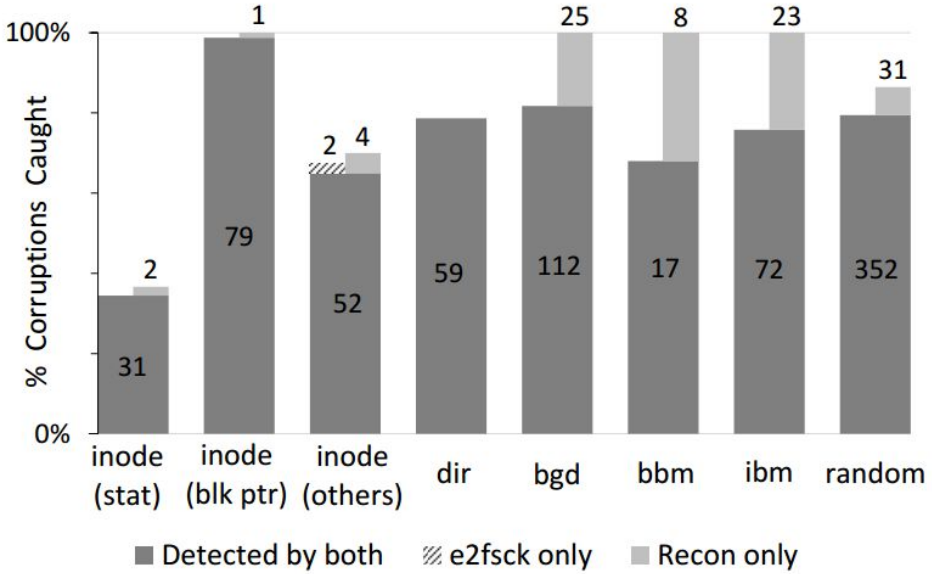
1. System Recon został zaimplementowany dla ext3
2. Implementacja obejmowała weryfikację poprawności systemu, pomijając możliwości ex2fsck do ewentualnego naprawiania systemu
3. Okazało się, że wystarczy zaimplementować jedynie 31 niezmienników - łatwiej jest weryfikować poprawność systemu podczas działania
    a. Przykładowo Superblock zawiera pola, które nie powinny się zmieniać. Zamiast weryfikować ich poprawność możemy dodać niezmiennik “te pole nie może się zmienić”
4. Cały projekt zajął 3,800 linii kodu w C
    a. 1,500 linii jest wspólnych dla różnych systemów plików
    b. 1,500 linii kodu pozwala interpretować metadane ext3
    c. Jedynie 800 linii implementuje niezmienniki ext3 (średnio 26 linii na niezmiennik)

Rezultaty - sprawdzanie poprawności

1. Testowanie na znanych błędach okazało się zbyt trudne - były one trudne do reprodukcji, zależne od konkretnej wersji kernela, załadowanych modułów itd.
2. Zamiast tego postanowiono po prostu wstrzykiwać błędne dane:
   a. W konkretne miejsca w metadanych
   b. Losowo
3. Za każdym razem dane były wstrzykiwane na chwilę przed umieszczeniem ich w dzienniku
4. Następnie dane były commitowane - jeśli Recon zauważył błąd to go logował
5. W kolejnym kroku system plików był odmontowywany i puszczono na nim e2fsck
6. Korupcja danych uruchamiała się 20-90 sekund po rozpoczęciu testu
7. Test konkretnie wykonywał naprzemiennie kompilację kernela / make clean

Rezultaty - sprawdzenie poprawności

1. inode (stat) - struktura zawiera takie pola jak timestampy, więc ciężko wykryć w nich błędy
2. inode (blk ptr) - e2fsck ignoruje nieużywane i-węzły
3. inode(others) - Recon nie dwukrotnie nie wykrył niepoprawnego ustawienia flagi, wykrytego przez e2fsck
4. dir - nie wykryto błędów w nazwach katalogów
5. block group descriptor, block bitmap, inode bitmap - Recon działa wyraźnie lepiej:
    a. gwarantuje, że niektóre części struktur nie mogą się zmieniać
    b. wykrywa zmiany w nieużywanych częściach struktur danych



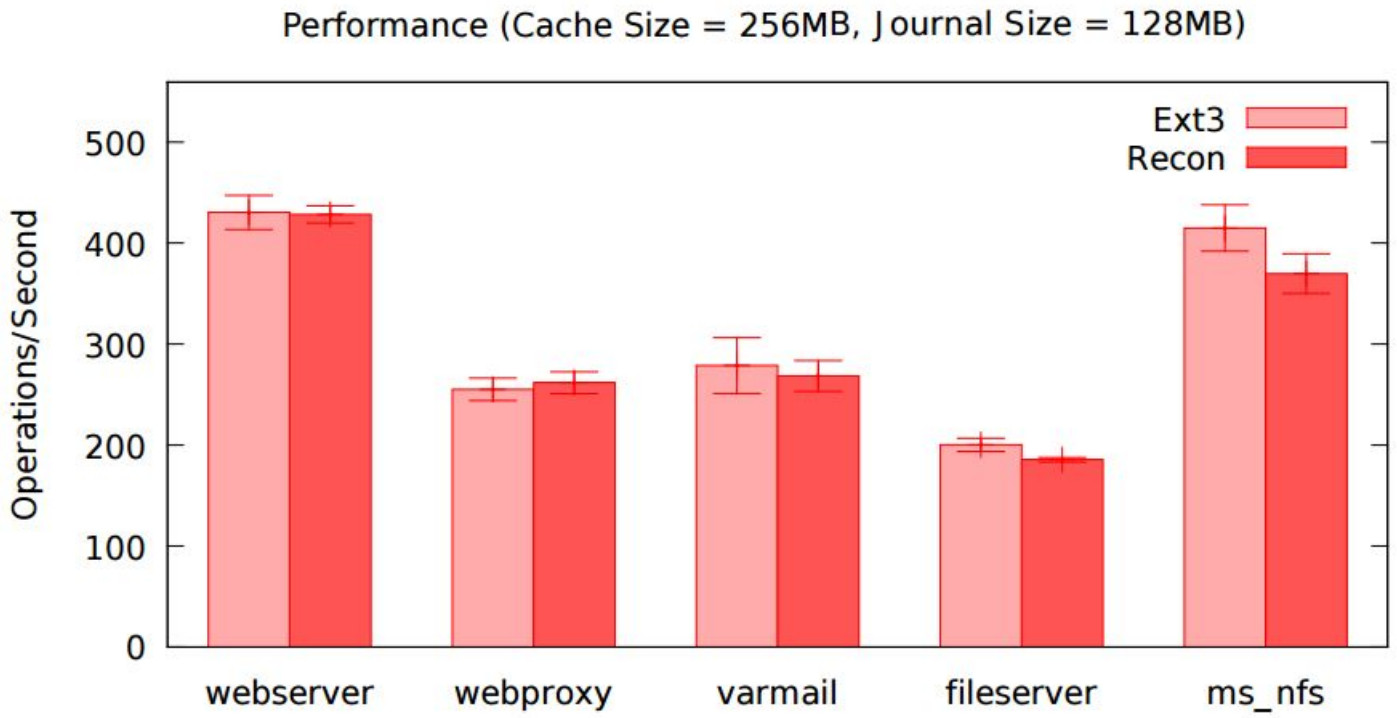
Efekt uboczny badania - kilka słów o e2fsck

1. Podczas 28 z 731 testów e2fsck wykrywał błędy w systemach plików nawet po tym gdy je naprawił
2. Dwa z tych przypadków nastąpiły w przypadku zmiany jedynie jednego bajtu metadanych
3. Widzimy więc, że nawet drobne uszkodzenie metadanych może całkowicie zepsuć nam system plików, a przecież liczba błędów rośnie im dłużej uszkodzony system działa

Rezultaty - wydajność

1. Eksperyment został przeprowadzony w następujących warunkach:
    a. Procesor: 2x Intel XEON 3Ghz
    b. Pamięć: 2GB RAM
    c. Dysk twardy: 1TB, sformatowany ext3
    d. Aplikacja: FileBench (wersja 1.4.8.fsl.0.8)
    e. Test trwał godzinę

Rezultaty - wydajność



Krótkie podsumowanie

1. System Recon pokazał, że da się weryfikować poprawność metadanych podczas działania systemu plików
2. Rezultaty są lepsze niż w przypadku uruchomienia narzędzia e2fsck
3. Narzut związany z działaniem weryfikacji jest znikomy
4. Niestety nie istnieje żadna produkcyjna implementacja Recona, więc na razie musimy żyć bez niego

Co dalej?

1. W 2012 roku autorzy poinformowali, że są w trakcie implementacji wersji współpracującej z systemem btrfs, nie znalazłem żadnej informacji, czy projekt został skończony
2. Praca została cytowana 31 razy (według Google Scholar), jednak nie znalazłem pracy, która nawiązywałaby do Recona dłużej niż jednym zdaniem.
3. Dlatego nie ma co czekać, tylko trzeba wykorzystywać ten udany pomysł w swoich projektach!

Czas na pytania!

Dziękuję za uwagę!