Vol.2014-DBS-160 No.6
Vol.2014-OS-131 No.6
Vol.2014-EMB-35 No.6
2014/11/18

ファイルシステムテストツールのカバレッジ調査

青田直大[1]　吉村剛[1]　河野健二[1]

概要：ソフトウェアが仕様通りに実装されていることを確認するため，テストコードによる動作検証が一般的に行われている．テストコードによって検査されるコードのカバレッジは検査後のソフトウェアの品質に直結するため，テストコードのカバレッジをあらかじめ検証しておく必要がある．本発表では，Linux のファイルシステムのテストツールである xfstests のカバレッジ調査の結果を報告する．xfstests は XFS だけではなく ext4 や btrfs などのさまざまなファイルシステムに対応した汎用的なテストツールとなっている．ハードウェアのパフォーマンスモニタリング機能を利用し，効率的に xfstests 実行時のコードカバレッジを取得し，その結果を元に各ファイルシステムでどの部分のコードがどの程度テストされているのか調査した結果を示す．

キーワード：ファイルシステム，ソフトウェアテスト，テストカバレッジ

1. はじめに

ローカルファイルシステムはユーザの様々なデータを保管するシステムの重要なコンポーネントである．ファイルシステムのバグはデータ破損などの深刻な被害をもたらすため, 特に注意して避けなければならない．

しかし, 現実には Linux kernel の近年のバージョンではファイルシステム部分に多くのバグがあることが報告されている [1]. バグが増えた要因には, 既存のファイルシステムへの新しい機能の導入と Btrfs [2] や F2FS [3] といった新しいファイルシステムの登場が理由として挙げられている．これらのファイルシステムは複数ディスクを 1 つのファイルシステムとして用いることを可能にする, あるいはフラッシュストレージ上での用途に最適化された設計を持っている．これらのファイルシステムが備える機能は, ビッグデータに代表されるような大容量のデータを保管するニーズの増加や, スマートフォンの普及によるフラッシュストレージの使用頻度の増加などストレージの状況の変化から必要とされているものである．現在多く使われている ext4 や XFS といったファイルシステムは単一のハードディスク上にファイルシステムを構築することを想定した設計となっており, ストレージの変化に対応しきれてはいないので, ストレージの変化に対応した新しい機能を実装しつつも, バグの導入は避けていくことがが求められている．

[1] 慶應義塾大学
Keio Uniersity

こうした機能追加によるバグの導入を検出する手段として, 一般にテストコードによる動作検証が行なわれている．Linux kernel においても, Linux 全体のテストを行なう Linux Test Project [4], ファイルシステムのテストを行なう xfstests [5], デバイスマッパーサブシステムのテストを行なう Device Mapper Test Suite [6], また Linux kernel の起動時に CRC32 や NMI などのコードが正しく動いていることを検証するセルフテストと呼ばれるコードが導入されている．

テストコードを改善することで, Linux kernel のコードも改善され, バグの数を減らすことができると期待される．テストコードを改善するためには, そのテストがコードのどの部分をテストしているのかを把握することが必要となる．しかし, Linux kernel のような大規模なソフトウェアのカバレッジを取得する場合, カバレッジ取得によるオーバヘッドが大きくなるという問題点がある．

一般的にカバレッジ取得に用いられている instrumentation という手法には 50% から 200% のテスト実行時間のオーバヘッドがあると言われている [7]. Linux kernel のように大規模なシステムのテストでは, そのテスト時間も長くなることが予想され, それだけオーバヘッドによる失われる時間も大きくなる．しかし, 長時間の開発時間のロスは Linux kernel のような活発な開発が行なわれているシステムにとって望ましくない．

instrumentation に対して, 近年の CPU の機能を用いたハードウェアによるカバレッジ取得手法も研究されている [8, 9]. これらの方法は instrumentation に対して, より

低いオーバヘッドでカバレッジの取得を可能とするが, その手法の制限から正確なカバレッジを取得できないという問題がある.

これまでのハードウェアによるカバレッジ取得手法の研究 [8–10] は, その対象をベンチマークソフトとしていた. これらのソフトウェアは主に CPU に依存したワークロードを行なうため, ファイルシステムテストツールのワークロードとは異なっている. そこで本研究では, これら 2 種類のカバレッジ取得方法をファイルシステムのテストツールである xfstests に適用し, 2 つの手法のトレードオフについて調査する. さらに, 取得されたカバレッジ結果を分析し xfstests の各ファイルシステムにおけるテスト状況について報告する.

本論文の構成を以下に示す. 2 章ではカバレッジ取得に用いられる手法について述べる. 3 章ではテストツール xfstests について述べる. 4 章では の評価方法について説明する. 5 章では調査結果とその結果についての考察を述べる. 6 章ではまとめを述べる.

2. カバレッジ取得方法

2.1 instrumentation

instrumentation の手法では, まず測定対象のプログラムのバイナリまたはソースコードを編集し, 計測用のコードを埋め込む. 対象プログラムが実行されると, この計測用のコージが実行されることで, 対象プログラム中のどの部分が実行されたのかが記録される.

この手法は Squish Coco [11] や gcov [12] でも用いられている一般的な手法である. 特に gcov は, Linux kernel でも使用することがようになっている [13].

instrumentation を使えば, 確実にカバレッジを取得できるが, 実行速度に平均して 25%のオーバヘッドが発生する [11]. また, カバレッジ用にバイナリやソースコードを編集, または特別なコンパイルフラグを付けてコンパイルする必要があるという問題がある. 特に Linux kernel では, kernel 全体を instrumentation してしまうか, あるいはカバレッジを測定したいディレクトリごとに Makefile を編集する必要がある.

2.2 ハードウェアを用いたカバレッジ取得

近年の CPU にはハードウェアモニタリングという, プログラムのパフォーマンス解析やデバッグに用いることができる機能が備わっている. その中の Branch Trace Buffer (BTB) や Last Branch Record (LBR) を用いたカバレッジ取得手法が研究されている [8, 9]. BTB と LBR はいずれもリングバッファのように機能するレジスタ群に最後に実行した n 個のブランチを記録する機能のことである. n は BTB では 4 であり, LBR では CPU アーキテクチャによって異なるが Intel Nehalem プロセッサでは 16 である [14]. これらのレジスタを適当なサンプリング間隔で読みとることで, その時点での最後のいくつかの実行ブランチを知ることができるので, そこからカバレッジを計測することができる.

ハードウェアを用いたカバレッジ取得には instrumentation よりもオーバヘッドが小さくなること, コードやバイナリの変更といったカバレッジ専用のコンパイルが必要ないというメリットがある. しかし, 2 つの問題がこの方法での完全なカバレッジ情報の取得を困難にしている.

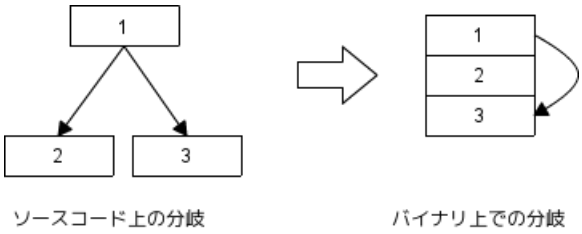

図 1 BTB, LBR は 1 → 3 の場合のみを記録する

1 つ目の問題は fall-through ブランチの問題である. 図 1 の左側にあるようなソースコード上の分岐は, コンパイルによって同図右のようなバイナリになる. この時分岐は 1 から 3 にジャンプした場合にのみ起き, 1 から 2 にそのまま実行が遷移した場合にはその情報は LBR などには記録されない. これが fall-through ブランチの問題である. この問題は INST_RETIRED イベントを監視する, またはコードの静的解析を行なう, あるいは 1 から 2 の間に無条件のジャンプ命令を挿入することで解決することができる [8].

2 つ目の問題は実行ブランチのとりこぼしの問題である. サンプリングとサンプリングの間に多くのブランチ命令が実行された場合, LBR に前回のサンプリングから今回のサンプリングの間に実行された全てのブランチが記録されないことがありうる. この場合, 得られたカバレッジ結果は不完全なものとなってしまう. サンプリング間隔を短くすれば, この問題を削減することができるがその間隔によっては instrumentation よりもオーバヘッドが悪化してしまう [8].

他のハードウェアを用いたカバレッジ取得手法として, Branch Trace Store (BTS) を用いたものがある [15]. BTS も LBR と同様に実行したブランチを記録する機能であるが, 記録する場所がメモリ上であること, 記録用の領域が少なくなった場合に割り込みを発生させることができることの 2 点で LBR とは異なっている. こうした違いから BTS では LBR と違って完全なブランチを取得することができ, 完全なカバレッジを取得できると考えられる. しかし, BTS を有効にした時に元の実行速度よりも 40 倍遅くなった例も報告されている [10].

Vol.2014-DBS-160 No.6
Vol.2014-OS-131 No.6
Vol.2014-EMB-35 No.6
2014/11/18

3. xfstests

xfstests は Linux のファイルシステムを対象としたテストツールである. もともとは Linux のファイルシステムの 1 つである XFS のみを対象としたテストツールであったが, いまでは XFS 以外のファイルシステムにも動作させることができる一般的なテストツールとなっている.

表 1 xfstests のテストの内訳

対象	テスト数
generic	149
shared	6
btrfs	77
ext4	12
udf	3
xfs	202

xfs のテストは表 1 に示すように, 複数のファイルシステムに同じように適用できるテストと, 各ファイルシステムに固有の機能のためのテストの 2 種類に分かれている. また, 各テストは 1 つまたは複数のグループに属している. このグループ名を用いて特定の機能をテストするテストだけを実行することができる.

各テストはシェルスクリプトによって記述されており, スクリプトの実行が成功したかどうか, 成功した場合はその出力が正しいものであるかどうかがテストされる. さらに, テストが以前にも実行されていた場合前回のテストに必要とした秒数と, 今回のテストに必要とした秒数が出力される.

4. 評価方法

ファイルシステムのテストに対しての instrumentation によるカバレッジ取得と, ハードウェアを用いたカバレッジ取得とのオーバヘッドおよび取得度合の比較を行なうため Linux kernel 組み込みの gcov 機能によると, LBR の読みとりによるカバレッジ取得の 2 つの手法を比較した. BTS については, 現在の Linux kernel ではカーネル空間を対象とすることができないようにされているため今回は評価対象としていない [16]. また, 対象のファイルシステムは, ファイルシステム固有のテストを持っている btrfs, ext4, XFS の 3 種類とした.

gcov を用いた場合は, 対象となる 3 つのファイルシステムのディレクトリの Makefile を編集し, それらの領域だけに限り instrumentation が行なわれるようにした. LBR の読みとりには Linux kernel 付属のパフォーマンス測定ツールである perf [17] を用いた. また, サンプリング間隔は 5000CPU サイクルとし, 読みとったブランチ情報からのカバレッジ解析には btrax [15] を用いた.

5. 調査結果

5.1 カバレッジ

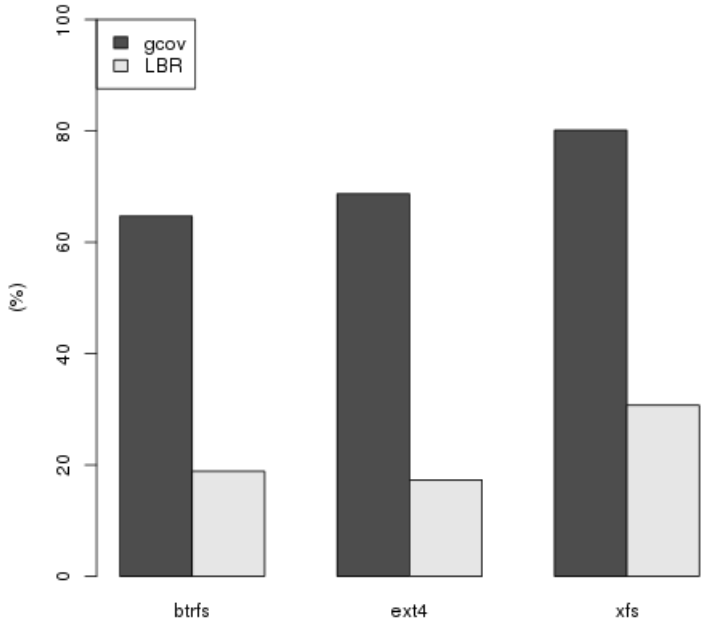

図 2 各ファイルシステムにおける gcov と LBR のカバレッジ比較

図 2 は btrfs, ext4, XFS の xfstests によるコードカバレッジを gcov による instrumentation と LBR の読みとりによる 2 つの方法で比較したものである. gcov によるカバレッジ取得は正確なものであると考えられることから, 今回の LBR の読みとりによるカバレッジ計測ではとりこぼしている部分が多くあることがわかる.

5.2 パフォーマンス

図 3 は, XFS における gcov と LBR のパフォーマンス比較を行なったものである. 左側は各テストの所要秒数を横軸を LBR の場合, 縦軸を gcov の場合としてプロットしたものである. 直線よりも上の点は, gcov の方が LBR よりも時間がかかっていて遅いテストであり, 直線よりも下の点は逆に gcov の方が速いテストである. 右側は各テストの LBR の実行時間に対する gcov の実行時間の比率を小さいものから順に並べてプロットしたものである. すなわち, 右側ほど gcov の方が遅いということになる. ほとんどのテストが gcov と LBR であまり変わらないパフォーマンスとなっているが, 最悪ケースでは gcov の方が LBR よりも 31 倍遅いという結果となった.

図 4 は, ext4 における gcov と LBR のパフォーマンス比較を行なったものである. こちらもほとんどのテストが

Vol.2014-DBS-160 No.6
Vol.2014-OS-131 No.6
Vol.2014-EMB-35 No.6
2014/11/18

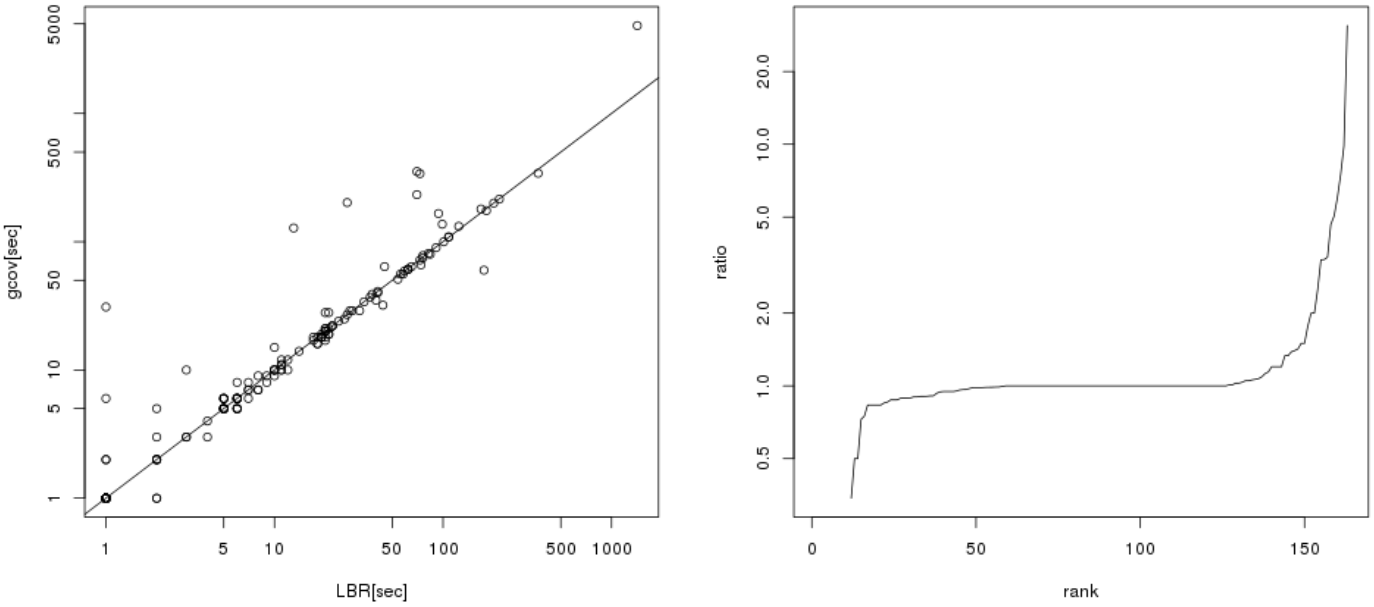

図 3　XFS における gcov と LBR のパフォーマンス比較

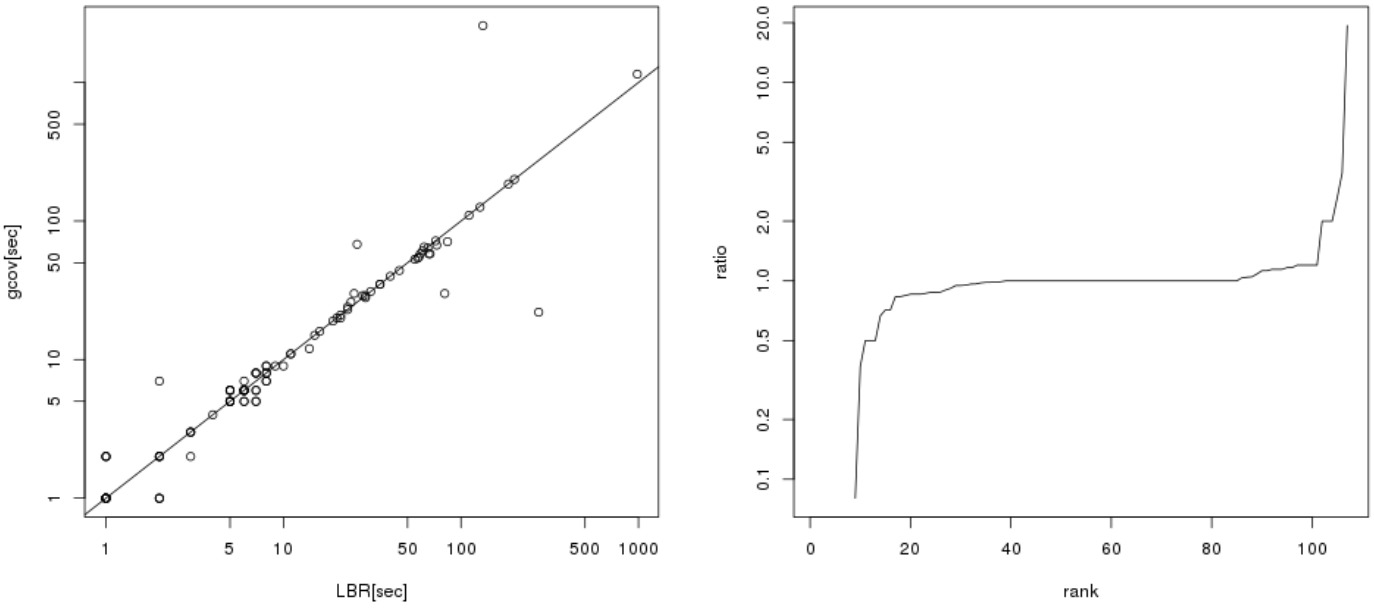

図 4　ext4 における gcov と LBR のパフォーマンス比較

gcov と LBR であまり変わらないパフォーマンスとなっている. 最悪ケースでは gcov の方が LBR よりも 19 倍遅いという結果となっていた. ただし, このテストケースはランダムな IO を発行するものであったのでより詳細な分析が必要だろう.

5.3　詳細なカバレッジ結果

表 2 に, XFS における xfstests のカバレッジの詳細な結果を示す. 行カバレッジはその行を実行したかどうか, 関数カバレッジはその関数を実行したかどうかを現す. この結果を元に XFS のどの機能がテストされていないか, どの機能にテストを追加すべきかを解析することができる. たとえば, 単純には xfs_log_recover.c というログのリカバリという重要な作業を行なうと思われるコードが 64% 弱のカバレッジとなっているので, 具体的にどの部分がカバーされていないのかを見ていく必要がある. または, xfs_sysfs.c

の部分にはテストが書かれていないが, これはこのファイルが比較的最近 (2014 年 7 月) に書かれたファイルであるかではないかと推察できる.

6. まとめ

本研究では Linux のファイルシステムの品質向上に, テストツールの改善からアプローチする. テストツールの改善のためには, テストツールのカバレッジを知ることが必要である. しかし, Linux のような大規模システムではカバレッジ取得のオーバヘッドが大きくなるという問題がある. そこで近年研究されている CPU の機能を使ったハードウェアによるカバレッジ取得を Linux のファイルシステムのカバレッジ取得に適用し, そのパフォーマンスとカバレッジの正確さを計測した. LBR を用いた場合, 現状ではその正確さに問題があること, そして多くのテストケースでパフォーマンスがあまり変わらないことを見た. これからカバレッジ結果を詳細に解析し, btrfs, ext4, XFS などの Linux の有名なファイルシステムにおいて, 何がテストされていないのか, どうすればテストカバレッジを向上できるのかを明らかにする.

参考文献

[1] Palix, N., Thomas, G., Saha, S., Calvès, C., Muller, G. and Lawall, J.: Faults in Linux 2.6, *ACM Transactions on Computer Systems*, Vol. 32, No. 2, pp. 1–40 (2014).
[2] Rodeh, O., Bacik, J. and Mason, C.: BTRFS: The Linux B-Tree Filesystem, *ACM Transactions on Storage*, Vol. 9, No. 3, pp. 1–32 (online), DOI: 10.1145/2501620.2501623 (2013).
[3] LWN.net: f2fs: introduce flash-friendly file system [LWN.net], https://lwn.net/Articles/518718/.
[4] LTP: LTP - Linux Test Project, http://linux-test-project.github.io/.
[5] sgi: oss.sgi.com Git - xfs/cmds/xfstests.git/summary, http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git;a=summary.
[6] Thornber, J.: jthornber/device-mapper-test-suite, https://github.com/jthornber/device-mapper-test-suite.
[7] Tikir, M. M. and Hollingsworth, J. K.: Efficient Instrumentation for Code Coverage Testing, *SIGSOFT Softw. Eng. Notes*, Vol. 27, No. 4, pp. 86–96 (online), DOI: 10.1145/566171.566186 (2002).
[8] Walcott-Justice, K., Mars, J. and Soffa, M. L.: THeME: a system for testing by hardware monitoring events, *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*, New York, New York, USA, ACM Press, pp. 12–22 (online), DOI: 10.1145/2338965.2336755 (2012).
[9] Shye, A., Iyer, M., Reddi, V. J. and Connors, D. A.: Code Coverage Testing Using Hardware Performance Monitoring Support, *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, New York, NY, USA, ACM, pp. 159–163 (online), DOI: 10.1145/1085130.1085151 (2005).
[10] Soffa, M. L., Walcott, K. R. and Mars, J.: Exploiting hardware advances for software testing and debugging, *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, New York, New York, USA, ACM Press, pp. 888–891 (online), DOI: 10.1145/1985793.1985935 (2011).
[11] froglogic: froglogic Squish Coco, http://www.froglogic.com/squish/coco/.
[12] Foundation, F. S.: Gcov - Using the GNU Compiler Collection (GCC), https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[13] Project, L. T.: Linux Test Project - Coverage gcov, http://ltp.sourceforge.net/coverage/gcov.php.
[14] Intel: Intel 64 and IA-32 Architectures Software Developer Manuals, http://www.intel.com/content/www/us/en/processors/architectures
software-developer-manuals.html.
[15] Hitachi: Branch Tracer for Linux, http://btrax.sourceforge.net/.
[16] Molnar, I.: x86, perf_counter, bts: Do not allow kernel BTS tracing for now, http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=1653192f510bd8114b7b133d7289e6e5c3e9504
[17] Perf: Perf Wiki, https://perf.wiki.kernel.org/index.php/Main_Page.

Vol.2014-DBS-160 No.6
Vol.2014-OS-131 No.6
Vol.2014-EMB-35 No.6
2014/11/18

表 2　XFS の gcov によるカバレッジの詳細 (抜粋)

ファイル名	行カバレッジ (%)	行カバレッジ	関数カバレッジ (%)	関数カバレッジ
kmem.c	67.4	29 / 43	100.0	6 / 6
kmem.h	60.0	12 / 20	50.0	3 / 6
uuid.c	100.0	11 / 11	100.0	3 / 3
xfs_acl.c	77.9	74 / 95	88.9	8 / 9
xfs_aops.c	88.5	479 / 541	95.3	41 / 43
xfs_bit.c	100.0	38 / 38	100.0	3 / 3
xfs_bmap_util.c	75.4	468 / 621	85.7	18 / 21
xfs_buf.c	88.2	559 / 634	94.7	54 / 57
xfs_buf_item.c	91.1	266 / 292	100.0	24 / 24
xfs_dir2_readdir.c	94.5	189 / 200	100.0	6 / 6
xfs_discard.c	77.3	51 / 66	66.7	2 / 3
xfs_dquot.c	79.9	271 / 339	82.4	14 / 17
xfs_dquot_item.c	97.9	92 / 94	95.0	19 / 20
xfs_error.c	45.5	10 / 22	33.3	1 / 3
xfs_export.c	33.9	21 / 62	50.0	3 / 6
xfs_extent_busy.c	81.8	112 / 137	100.0	8 / 8
xfs_extent_busy.h	100.0	3 / 3	100.0	1 / 1
xfs_extfree_item.c	74.4	99 / 133	91.3	21 / 23
xfs_file.c	92.5	420 / 454	96.2	25 / 26
xfs_filestream.c	85.6	113 / 132	90.9	10 / 11
xfs_fsops.c	84.1	290 / 345	90.0	9 / 10
xfs_icache.c	93.3	376 / 403	100.0	31 / 31
xfs_icreate_item.c	87.2	34 / 39	77.8	7 / 9
xfs_inode.c	85.2	822 / 965	94.9	37 / 39
xfs_inode.h	93.3	42 / 45	88.9	8 / 9
xfs_inode_item.c	86.3	221 / 256	94.4	17 / 18
xfs_ioctl.c	76.0	465 / 612	87.5	28 / 32
xfs_ioctl32.c	0.0	0 / 231	0.0	0 / 17
xfs_iomap.c	87.0	235 / 270	90.9	10 / 11
xfs_iops.c	93.0	384 / 413	92.0	23 / 25
xfs_itable.c	97.9	235 / 240	100.0	10 / 10
xfs_linux.h	93.3	14 / 15	100.0	4 / 4
xfs_log.c	90.2	941 / 1043	98.4	63 / 64
xfs_log.h	100.0	19 / 19	100.0	1 / 1
xfs_log_cil.c	95.6	259 / 271	100.0	17 / 17
xfs_log_recover.c	63.6	814 / 1280	89.2	58 / 65
xfs_message.c	34.4	11 / 32	50.0	6 / 12
xfs_mount.c	74.8	514 / 687	95.3	41 / 43
xfs_mount.h	71.4	10 / 14	100.0	2 / 2
xfs_mru_cache.c	88.2	112 / 127	85.7	12 / 14
xfs_qm.c	79.2	513 / 648	84.8	28 / 33
xfs_quotaops.c	86.3	44 / 51	85.7	6 / 7
xfs_rtalloc.c	3.8	14 / 372	20.0	3 / 15
xfs_stats.c	41.3	19 / 46	33.3	3 / 9
xfs_super.c	58.4	404 / 692	73.8	31 / 42
xfs_symlink.c	89.6	173 / 193	100.0	5 / 5
xfs_sysctl.c	0.0	0 / 26	0.0	0 / 4
xfs_sysfs.c	0.0	0 / 17	0.0	0 / 6
xfs_sysfs.h	100.0	11 / 11	100.0	3 / 3
xfs_trace.h	72.4	301 / 416	55.3	288 / 521
xfs_trans.c	78.8	272 / 345	100.0	18 / 18
xfs_trans.h	100.0	6 / 6	100.0	2 / 2
xfs_xattr.c	78.9	56 / 71	83.3	5 / 6