本文是F2DFS的测试说明，其中内容包括F2DFS各项测试的测试流程、测试结果和结果的分析。

1. **基础环境说明**

我们在Linux 5.10.50版本内核中实现了F2DFS，配置为CPU Intel Core i5-10505 @3.20GHz，16GB内存，512G nvme SSD，其中参与测试的分区大小都为50GB。

1. **基本IO性能测试**

在线去重场景下，我们使用了标准的IO性能测试工具FIO对原生F2FS和F2DFS进行了性能测试。此处对使用到的一些参数进行说明：

-ioengine：FIO使用的IO引擎，本次测试使用的有同步IO引擎sync、psync和异步IO引擎io\_uring

-rw：IO的类型，包括read、write、randread、randwrite等。

-bs：单次发起的IO大小。

-size：单个线程读写的IO总量。

-numjobs：测试时使用的线程数。

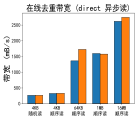
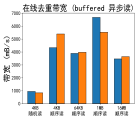
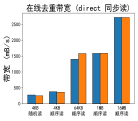
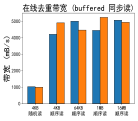
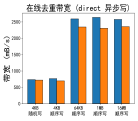
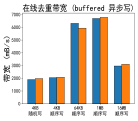
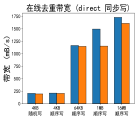
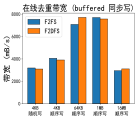
-dedupe\_percentage：重复率。

-buffered/direct：是否绕过page cache，直写硬盘。其中-buffered=1和-direct=0是等效的，反之亦然。

**2.1 测试参数**

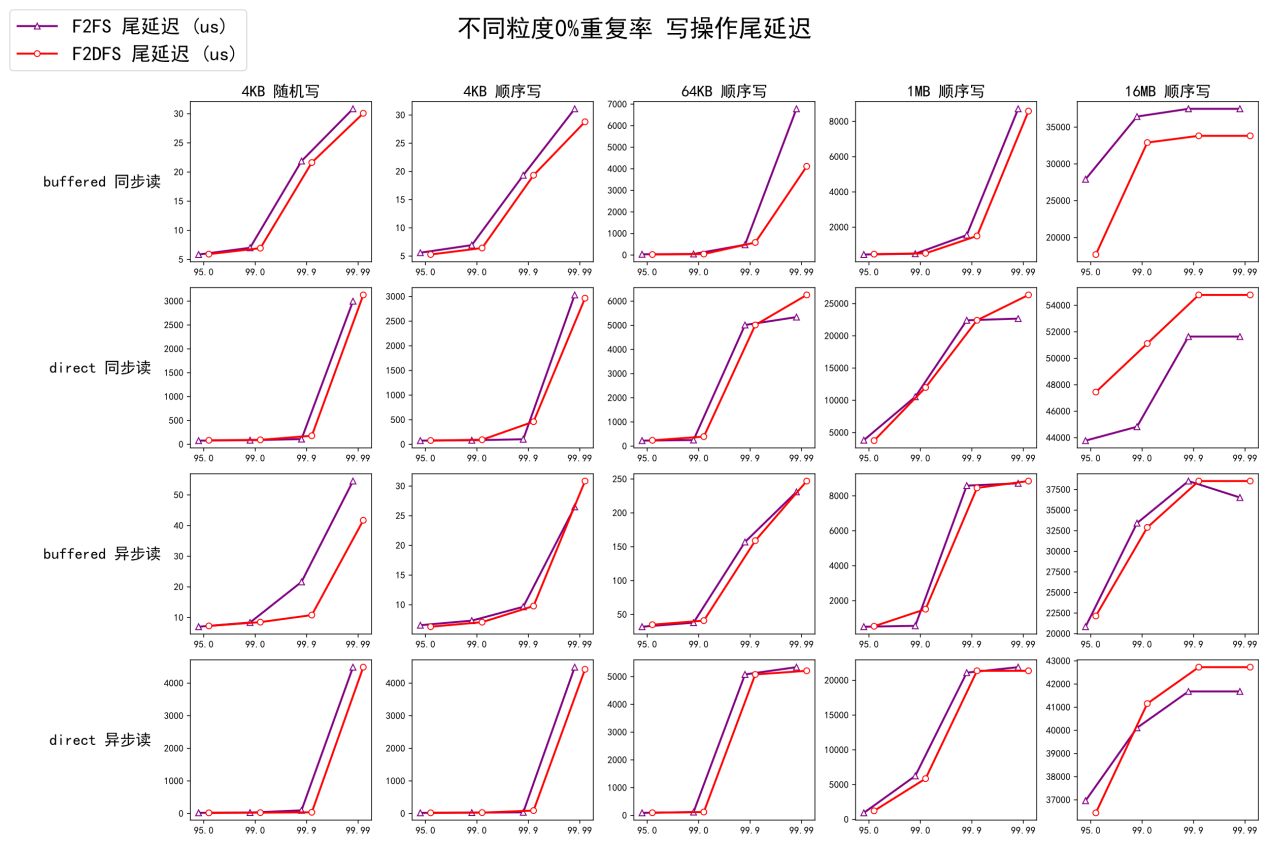
本组测试中使用的默认参数有：-dedupe\_percentage=0 -iodepth 1 -size=1G -numjobs=4。变量参数有： -ioengine=sync/io\_uring，分别对应同步和异步IO；-bs= 4K / 64K / 1M / 16M；-buffered=1/-direct=1，分别对应非直写和直写硬盘；以及-rw=read/write/randread/randwrite。

**2.2 测试结果及分析**

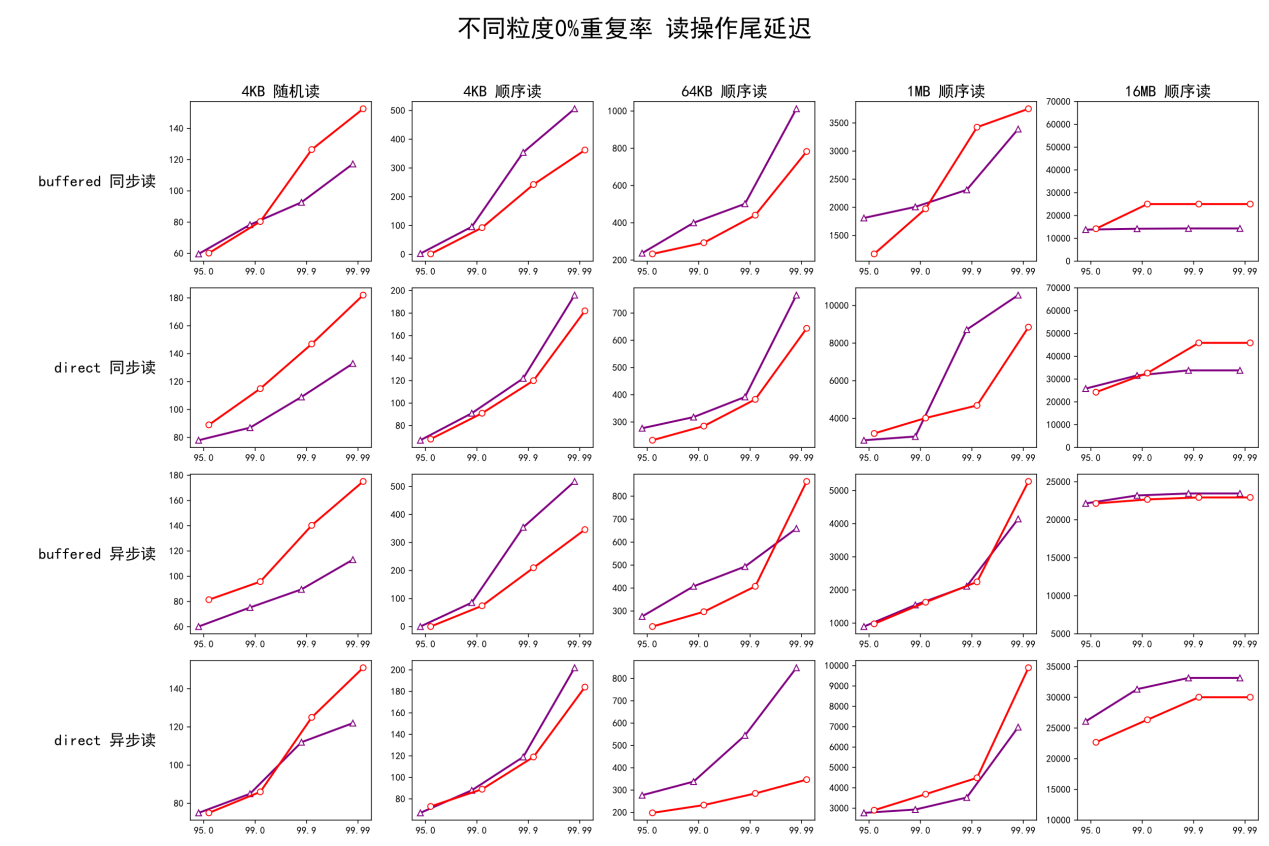


**图1 - F2DFS基本IO带宽测试，对比F2FS**

如图1所示，各张图标题显示了测试条件，横坐标则是测试的类型和操作的粒度，分别是4KB的随机读写和4KB~16MB的顺序读写，而纵坐标则是测试的带宽结果。平均而言，F2DFS的写操作带宽是原生F2FS的96%，相应的读操作带宽则为102%。

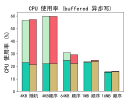
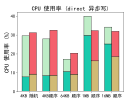
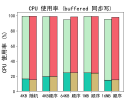
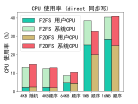
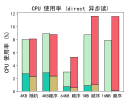
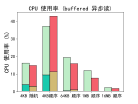
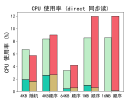
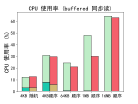


**图2** **- F2DFS写操作尾延迟测试，对比F2FS**



**图3 - F2DFS读操作尾延迟测试，对比F2FS**

图2、3是F2DFS在0%重复率时的读写操作的尾延迟情况测试，其中横坐标是尾延迟的百分比，纵坐标则是尾延迟时间。选择使用0%重复率进行测试的原因是此时F2DFS对所有的数据都需要进行完整的在线去重和读写流程，尾延迟相较于高重复率场景下是最高的。在此情况下，F2DFS相较原生F2FS，无论是读写尾延迟都没有明显增长。



**图4** **- F2DFS 在线去重CPU使用率测试，对比F2FS**

图4是F2DFS在进行在线去重场景下读写测试的CPU使用率情况，其中横坐标是不同的测试场景，纵坐标是CPU使用率。平均而言， F2DFS的写操作CPU使用率对比F2FS为102%，而读操作则为106%。需要说明的是，CPU使用率是一项与时间强相关的概念，在同样工作负载的条件下，更短时间内完成的程序会有更高的CPU使用率，反之亦然。

1. **垃圾回收时间测试**

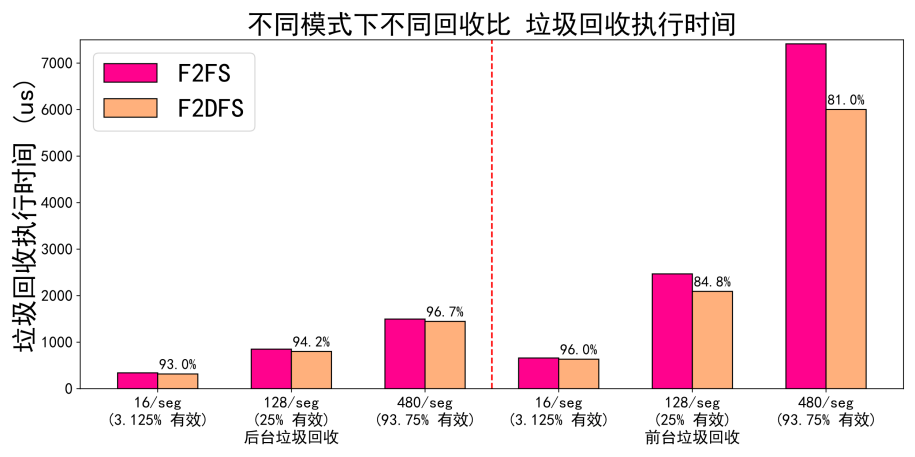
**3.1 测试流程**

F2FS垃圾回收的单位是Section，每个Section对应一个Segment，大小是2MB；在垃圾回收测试中，首先使用脚本写入特定大小的文件并删除其中部分文件，实现在每个Segment中都有特定数量的有效块，使文件系统老化到一致状态。其次使用IOCTL触发垃圾回收，并在F2DFS垃圾回收的主函数do\_garbage\_collect的首尾通过时间戳的方式获取垃圾回收流程的执行时间。

具体而言，文件系统老化使用的脚本是\*\*\*random\_generate.sh，其中星号代表每个Segment中有效块的数量，其原理如下：首先，一个Segment中有512个Block，每个Block的大小是4KB。假设要使每个Segment中都有16个有效块， 可以通过连续写入多个16\*4=64KB的文件，并对文件进行编号。文件写入后，将多余文件删除，使每个Segment对应的大小中只保留有特定有效块数量的文件；例如，需要16个有效块时，可以每32个文件中保留1个；需要480个有效块时，可以将单个文件大小改为128KB，并在16个文件中删除1个。

将文件系统老化到特定状态后，需要通过IOCTL触发垃圾回收。F2FS/F2DFS垃圾回收对应的IOCTL是ioctl(fd,\_IOW(0xf5, 6, unsigned int),&a)，其中fd是文件系统下的一个文件，\_IOW(0xf5, 6, unsigned int)是一个宏，使函数调用到文件系统垃圾回收的代码，而a是区分前后台垃圾回收的flag，a=0时触发后台垃圾回收，a=1时则触发前台垃圾回收。

**3.2 测试结果及分析**



**图5 - F2DFS垃圾回收执行时间测试，对比F2FS**

图5展示了F2DFS垃圾回收的执行时间测试，图的左右两部分分别是后台和前台垃圾回收，横坐标中的数字是每个Segment中有效块的数量，而纵坐标则是对这个Segment执行垃圾回收的平均时间。

F2DFS的垃圾回收执行时间平均下降至F2FS的91%。通过引入虚拟地址层，F2DFS避免了在引入去重后垃圾回收的巨大开销，并通过利用虚拟地址层实现了相对原生F2FS垃圾回收的优化。需要说明的是，引入了去重后文件系统本身的垃圾回收触发频率会降低，但在此测试中为了公平对比，上面的有效块数量是指物理块的有效数量。

**3.2 项目收集用例trace replay**

**3.2.1 trace情况离线分析**

本部分所描述的trace收集自vfs层，应用场景有视频、音乐、安装应用、文件拷贝等。我们将完整的trace分为了6个部分，称为trace1到trace6，分别进行测试。

表2 项目收集测试用例Trace相关信息表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **trace1** | **trace2** | **trace3** | **trace4** | **trace5** | **trace6** |
| 总写入量 | 3682.332 | 3820.605 | 3753.504 | 3865.488 | 3881.48 | 887.0039 |
| 总读取量 | 4142.527 | 1181.988 | 2281.457 | 593.1094 | 225.1953 | 461.543 |
| fsync次数 | 5601 | 10928 | 7301 | 5780 | 10334 | 1754 |
| 重复数据占比 | 20.92% | 40.98% | 4.41% | 0.19% | 1.50% | 2.39% |

表2是对收集到的trace的离线分析结果。除trace6外，其他trace的写入量都大概在3700 MB到3900MB间，读取量trace1较高，达到了4GB以上，其他trace的读取量不等。重复率方面，trace2拥有较多的重复数据，达到了40%左右，trace1次之，大约20%，而trace3-trace6的重复率都比较低，不到5%。

**3.2.2 不开启fsync感知情况下的结果及分析**

图7是在不开启fsync感知情况下F2DFS与F2FS执行时间的对比图，无fsync代表replay程序遇到应该fsync指令时直接跳过，执行时间以F2FS为100%归一化。从图中可以看到，有fsync场景下，F2DFS的执行时间普遍达到了F2FS的120%左右；而无fysnc场景下，这个数值大概在100%到110%。有fsync场景下，F2DFS对比F2FS的性能下降比无fsync场景下更明显。

**图 7 项目收集用例 F2DFS对比F2FS执行时间**

**图 8 项目收集用例 F2DFS对比F2FS新增落盘数据量**

图8是新增落盘数据量的情况。新增落盘数据量是指测试前后系统中文件占用大小的差值，反应了系统去重的情况，但是其中假如有对已有数据的update，则不会体现在最终落盘数据量上。总体而言，此处的结果符合上述中对trace的分析：trace2有最高的重复率，trace1少于trace2，其他trace的重复率很低，导致最终落盘数据量方面F2FS与F2DFS差异极小。

**图 9 项目收集用例 F2DFS对比F2FS各操作执行时间**

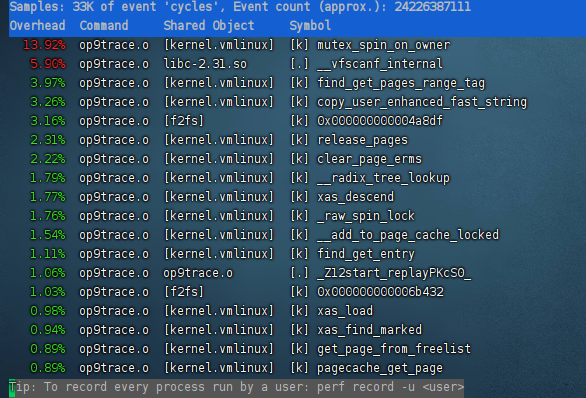
具体而言，我们将有fsync场景下的各个部分操作时间都进行了统计，结果展示在图9。可以看到read+write的时间在两者间的变化并不大，总时间的增长主要集中在fsync所使用的时间。进而我们统计了sync写在各个trace中的占比，结果如下表4：

表3项目收集测试用例Trace同步写占比信息表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| trace | trace1 | trace2 | trace3 | trace4 | trace5 | trace6 |
| write | 942677 | 978075 | 960897 | 989565 | 993659 | 227073 |
| Sync write | 552248 | 468190 | 867524 | 980294 | 937210 | 218885 |
| 同步写占比 | 58.58% | 47.87% | 90.28% | 99.06% | 94.32% | 96.39% |

表中可以发现，除了trace1和trace2的同步写占比稍低以外，其他trace的同步写占比都达到了90%以上，是比较极端的场景。这也意味着在不开启fsync感知情况下，F2DFS相对于F2FS的20%执行时间增长几乎是worst case下的情况。

通过性能分析工具perf，我们查找到了此时的性能瓶颈——锁：

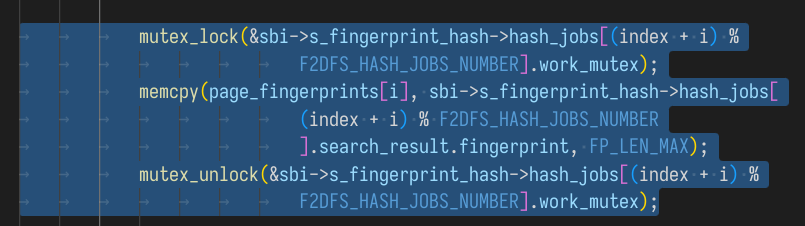


**图 10 项目收集用例 F2DFS执行replay时各函数占用情况**

进一步地，通过ftrace跟踪fsync相关函数执行时间，可以看到如下情况：

表4项目收集测试用例F2FS/F2DFS跟踪fsync耗时情况

|  |  |  |
| --- | --- | --- |
|  | **f2fs\_write\_data\_pages()** | Fsync总耗时 |
| F2FS | ~50us | ~850us |
| F2DFS | ~130us | ~790us |

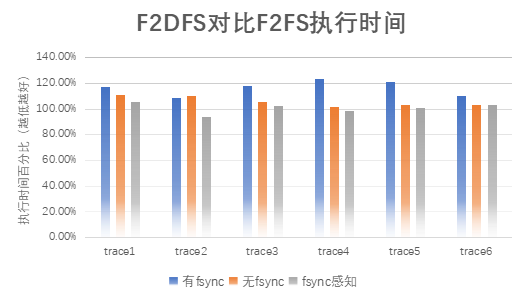
fsync总耗时变化不大的情况下，f2fs\_write\_data\_pages()函数的执行时间增长了一倍以上，通过跟踪函数调用关系可以发现如下调用路径：f2fs\_write\_data\_pages()->f2fs\_write\_cache\_pages()->mutex\_lock()->\_\_mutex\_lock\_slowpath()-> mutex\_spin\_on\_owner()，路径最后的mutex\_spin\_on\_owner()函数就是上述造成性能瓶颈的原因。检查代码中的相关字段，这个锁保护的代码是并行哈希相关的函数。

**图 11 项目收集用例F2DFS执行replay时性能瓶颈代码**

综上所述可以得出结论：F2DFS在有fsync场景下的性能下降主要原因是fsync要求的同步机制与并行哈希的尾延迟间的相互影响。这也是我们开启fsync感知能力的motivation。

**3.2.3 开启fsync感知情况下的结果及分析**

fsync感知指的是在sync路径下不分配虚拟地址而直接分配物理地址，也不进行在线去重，以尽可能减少对性能的影响。



F2FS

**图 12项目收集用例 F2DFS对比F2FS执行时间（fsync感知）**

图12是加入了带fsync感知的F2DFS的执行时间对比图，可以看到在加入fsync感知的情况下，F2DFS的执行时间回落到接近F2FS的水平，6个trace平均执行时间仅为F2FS的100.77%。但是如图13，相应的代价是由于不对sync路径上的写进行去重，再加上绝大多数数据是以sync形式写入，所以落盘数据量较之前更多了，特别是trace1和trace2。

**图 13 项目收集用例 F2DFS对比F2FS新增落盘数据量（fsync感知）**

表5项目收集测试用例Trace异步写重复数据占比信息表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **trace1** | **trace2** | **trace3** | **trace4** | **trace5** | **trace6** |
| 重复数据占比 | 20.92% | 40.98% | 4.41% | 0.19% | 1.50% | 2.39% |
| 重复数据占比（仅异步写) | 3.90% | 17.50% | 0.97% | 0.06% | 0.40% | 0.99% |

表5展示了各个trace中的重复数据占比。通过分析在异步写中的重复数据占整体的比例，可以看到除trace2外的其他trace都很低，符合最终落盘数据的情况。trace2在异步写中有17.5%的重复数据可删除，但是由于update并没有体现在落盘数据量上。但由于在线重复数据删除的减少数据写入量的效果，trace2的执行时间甚至相较F2FS还要更低，达到了约94%的水平。

**3.3 公开用例trace replay**

针对移动端场景，我们选择了SmartDedup上开源的trace进行重放测试，trace的下载网址为http://visa.lab.asu.edu/web/resources/traces/traces-smartphone/。由于完整trace的测试量过大且其中包含大量额外信息，所以我们选择了原trace中的3个片段并进行了修剪，仅保留了我们需要的读写以及同步操作。修剪后的trace信息如下：

表6 公开测试用例Trace相关信息表

|  |  |  |  |
| --- | --- | --- | --- |
| **基本信息** | **Trace 1** | **Trace 2** | **Trace 3** |
| 总写入量（含update） | 2.15GB | 0.77GB | 3.02GB |
| fsync次数 | 9959 | 25083 | 11862 |
| 磁盘占用 (F2FS) | 548.74MB | 36.13MB | 35.34MB |
| 磁盘占用 (F2DFS) | 538.14MB | 28.38MB | 31.88MB |
| 磁盘占用 (F2DFS\_fsync感知) | 538.17MB | 28.97MB | 31.89MB |
| 总体重复率 | 45.50% | 53.09% | 25.45% |

总体重复率是通过统计trace中写操作的FP个数除以总写操作次数获得的，磁盘占用是指trace重放完成后最终落盘的数据量。执行时间测试结果如图7所示，结果均以F2FS结果为100%归一化。在上述测试场景中，F2DFS的执行时间优于F2FS，但F2DFS（fsync感知）则较差。这是由于上述测试用例中的重复率较高，F2DFS在开启sync写在线去重的情况下可以有效减少写入数据，而开启fsync感知后则做不到这点，导致需要写入更多的数据，进而产生性能的下降。

**图 14 Linux 5.10 F2DFS在限制内存条件下的写操作尾延迟测试**

**3.4 块级去重对比文件级去重的收益**

本部分统计了几个场景中块级去重对比文件级去重的收益情况，结果展示在表7中。表中上下分别是文件级去重和块级去重的收益表，其中dup为重复数据量，total为总数据量。

表7 项目收集测试用例Trace相关信息表

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | C盘 | D盘 | /home | Android | Kernel | VM |
| File | dup | 3.85G | 775.84M | 276.90 M | 31.84 M | 3.77G | 0 |
| total | 66G | 75.8G | 32.2G | 3.4G | 9.6G | 39.7G |
| Block | dup | 33.57G | 22.50G | 1.81G | 164.98M | 4.71G | 7.26G |
| total | 66.25G | 75.74G | 31.78G | 3.41G | 8.17G | 39.66G |

C/D盘分别是Windows系统下的系统分区和文件分区；其中，C盘中主要的数据是Windows系统、Windows自带应用（如Office、Edge浏览器等）、少量第三方应用以及一些系统运行文件（如pagefile）等；D盘中主要存放的数据是虚拟机文件、系统安装镜像，还有部分文档等占据了较小的空间。

/home是Linux下/home目录的结果，该目录中的主要文件有流媒体音乐软件的缓存、部分编译完的软件（如qemu）、部分linux软件的安装包等。

Android是手机内部共享存储空间目录下的镜像结果。由于Android的应用隔离和权限保护机制，不方便读取系统和其他应用空间，所以只能在此目录下测试。其中的主要数据是音乐和图片。

以上均是个人数据，其特征涉及到使用习惯等其他因素。

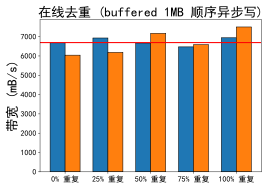
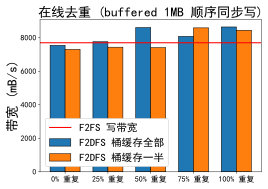
Kernel是linux 2.6.0 -2.6.39共40个版本的linux内核源码，在版本更新时没有被改动的文件均会被计算为重复文件，所以文件级去重就有比较好的表现。但在使用块级去重后，可探查的重复率进一步上升。

VM是[http://cloud-images-archive.ubuntu.com/releases/focal/](http://cloud-images-archive.ubuntu.com/releases/focal/" \o "http://cloud-images-archive.ubuntu.com/releases/focal/) 网站中提供的ubuntu server镜像文件，涉及11个版本，共77个镜像。由于是整个完整的镜像文件，所以文件级去重在此时无法发挥作用。块级去重扫描后大约有18%的重复率。

1. **在线去重测试**

**4.1 限制内存使用量场景下的写测试**

由于F2DFS的主要目标使用场景是内存资源有限的移动设备，所以我们进行了限制内存使用量时的写性能测试。具体而言，F2DFS依赖内存中的桶结构进行在线去重，我们通过将单个桶内的slot数量减少50%，在桶的缓存量只有原测试条件的一半的情况下再进行测试，同时还同步对比了原生F2FS。



**图6 - F2DFS在限制内存条件下的写带宽测试**

图6展示了在限制内存条件下F2DFS在不同重复率下对比原生F2FS和不限制内存用量条件下的F2DFS的顺序写测试结果，其中横坐标是不同的重复率，纵坐标则是带宽；可以看到，即使在限制内存的条件下，F2DFS仅在低重复率场景下有轻微带宽下降，平均而言，限制内存的F2DFS的写操作带宽为不限制内存场景下的97%。

图表, 折线图

描述已自动生成

**图7 - F2DFS在限制内存条件下的写操作尾延迟测试**

图7展示了F2DFS在限制内存条件下的写操作尾延迟结果，对比原生F2FS和桶缓存全部slot的F2DFS，桶缓存一半slot的F2DFS在1MB粒度下，尾延迟与上述两个方案相近，不存在较大的尾延迟下降问题。

**4.2 内存开销评估**

**表1 F2DFS对比SmartDedup的内存代价**

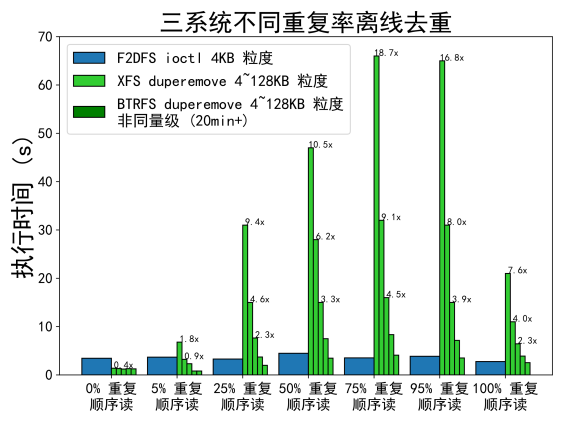
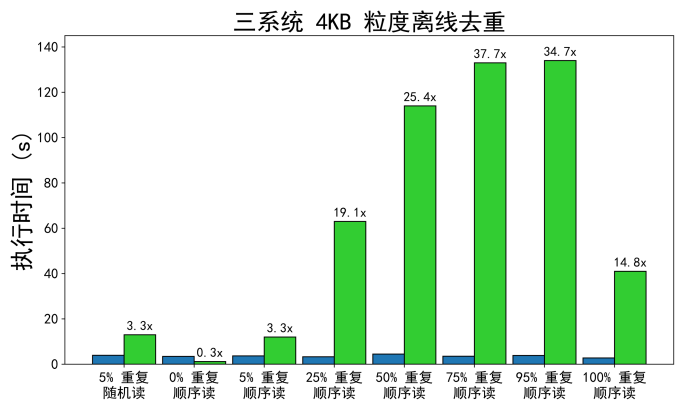
|  |  |  |
| --- | --- | --- |
| **内存代价** | **F2DFS** | **SmartDedup** |
| 在线去重（桶） | **SHA1:** 6.25MB/ 1GB | **MD5:** 4.6MB/ 1GB |
| 离线去重（普通索引） | 254KB/ 100GB | 200MB/ 100GB |
| 离线去重（碎片索引） | 2.5MB/ 1GB | 2MB/ 1GB |

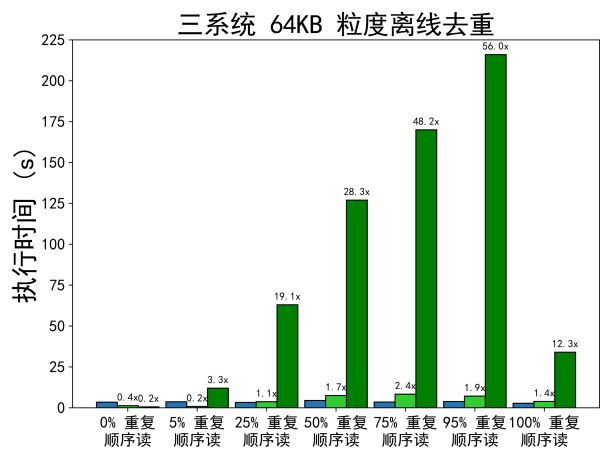
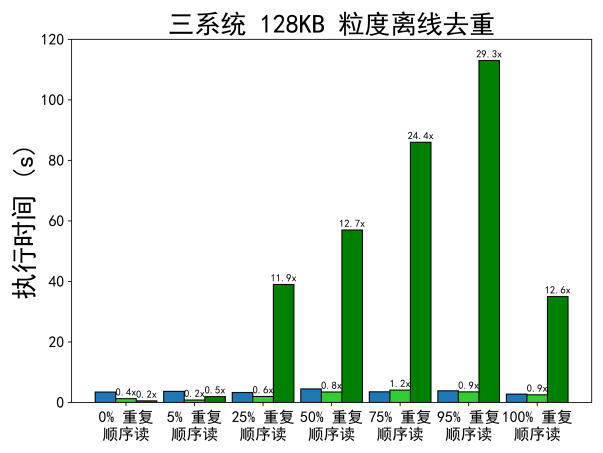
SmartDedup是近年提出的专门针对资源受限设备的内存开销友好设计工作，与其对比，F2DFS在在线去重场景下由于使用了抗碰撞能力更强，指纹长度更长的SHA1哈希，所以内存代价略高于SmartDedup。而在离线去重场景下，通过引入离线缓冲区、离线位图等结构，F2DFS离线去重内存索引开销显著优于对方。具体结果如表1所示，在线去重场景下，F2DFS的内存开销为6.25MB/GB，而SmartDedup的内存开销为4.6MB/GB。现有的支持在线去重的个人用户文件系统一般选用的是MD5（16字节/块）或SHA1（20字节/块），企业级归档文件系统选用的是SHA1或SHA256（32字节/块）。

离线去重场景下，针对普通索引内容，F2DFS的内存开销为254KB/ 100GB，而SmartDedup的内存开销为200MB/ 100GB。离线去重场景下，针对碎片索引内容，F2DFS的内存开销为2.5MB/ 1GB，而SmartDedup的内存开销为2MB/ 1GB。

1. **离线去重测试**

在离线去重的测试中，我们的对比对象是BtrFS/XFS+duperemove。基本的测试文件是由FIO生成的不同重复率的1GB文件，写入后F2DFS中通过IOCTL触发离线去重，而BtrFS/XFS则手动调用duperemove完成离线去重。以上测试都通过time命令监控执行时间和CPU占用情况，测试结果如下：

**图8 - F2DFS离线去重执行时间测试，对比XFS/BtrFS +duperemove**

由于BtrFS+duperemove的离线去重测试时间已经达到了20min+，对比F2DFS和XFS +duperemove已经并非同一数量级，所以并未在上图中进行展示。从图8中可以看到，在同为4KB的去重粒度下，XFS +duperemove仅在重复率0%的场景下对比F2DFS有所领先，其他场景下F2DFS均有不同的领先。

究其原因，主要是XFS/BtrFS+duperemove在执行离线去重时所用的时间主要集中在修改文件映射关系上。在0%重复率时，由于没有重复数据，所以不需要修改映射关系，XFS/BtrFS采用顺序读方式，而F2DFS则采取4KB粒度读方式进行离线去重。因此，XFS/BtrFS凭借更高的读速度获得相较F2DFS的优势。而在一定重复率范围内，随着重复率的提高，重复数据的映射关系变得更加复杂，XFS/BtrFS+duperemove就需要将大量时间花费在修改文件映射关系上；而当重复率极高时，由于重复数据大多集中到一部分块上，所以重复数据的比对和修改映射效率又会有回升，综上所述出现了duperemove离线去重性能先升后降的的情况。

在将去重粒度提高到64KB~128KB的情况下，XFS/BtrFS +duperemove的离线去重执行时间达到了接近F2DFS在4KB粒度下的同量级水平。但由于此时的去重粒度较大，已经无法发现更小的重复块，将会导致系统的去重率下降。这意味着F2DFS可以在同样的去重粒度下执行时间更短，或者在同样的执行时间内进行更精细的离线去重。

图表, 条形图

描述已自动生成图表, 条形图

描述已自动生成

**图9 - F2DFS不同粒度下离线去重CPU占用量测试，对比XFS/BtrFS +duperemove**

图9展示了不同重复率情况下，三种系统离线去重的CPU利用率情况。结果显示，F2DFS的CPU利用率均低于BTRFS，除了0%重复率情况下，均低于XFS。在0%重率情况下，相比于XFS，F2DFS的高CPU利用率取决于F2DFS的整体系统执行时间大于XFS，从而使得CPU利用率升高。

手机屏幕截图

中度可信度描述已自动生成

**图10 - F2DFS离线去重最大内存驻留量测试，对比XFS/BtrFS +duperemove**

同时我们还测试了离线去重时三系统在不同重复率和去重粒度下的最大内存驻留量，结果如图10。需要说明的是，此处测试的内存量是离线去重时读入并驻留在内存中的文件数据量，并非整个文件系统占用的数据量。F2DFS在不同重复率下都保持了大约4MB的内存驻留量，而XFS/BtrFS+duperemove的内存驻留量随着重复率的增加而整体呈现增加的趋势，同时在去重粒度更精细的情况下同样会增加内存驻留量。但无论在何种测试条件下，F2DFS的内存驻留量都优于XFS/BtrFS+duperemove，即使在duperemove使用较大的去重粒度，无法完成精细化去重的情况下，F2DFS的内存驻留量仍然更小。这说明F2DFS在离线去重的过程中使用的数据结构对内存占用量更加友好。

1. **其他测试**

**6.1 同步路径感知后的性能**

**图表, 条形图

描述已自动生成**

**图11 - F2DFS同步路径感知的自适应优化**

在F2DFS的设计方案中，为了确保断电或崩溃后的数据一致性，fsync触发的同步写流程不会经历虚拟地址的分配与在线去重（包含CPU密集的哈希指纹计算），而是依照原生F2FS的读写流程执行。针对这一场景，我们分别测试了10GB无重复数据写入场景下fsync触发（物理地址）和非fsync触发（虚拟地址）的写带宽变化。如图11，相较于F2FS，F2DFS的同步和异步写入性能分别只下降了0.7%和6.2%。

**6.2 VAT回写开销**

图表, 条形图

描述已自动生成图表, 条形图

描述已自动生成

**图12 - F2DFS检查点建立元数据写回开销**

由于F2DFS新引入了虚拟地址机制，同样测试了由于VAT更新导致的开销，具体来说是检查点建立函数f2fs\_write\_checkpoint的处理时间。如图12，在无重复数据的最坏场景下，F2DFS的检查点冻结时间是原生F2FS的3倍，而随着重复率增加，这部分性能开销会得到弥补。

**6.3 trace replay**