|  |
| --- |
| Improving restore speed for backup systems that use inline chunkbased deduplication |
| 场景：   * 很多备份系统中一个文件的chunks分散在不同的备份中，造成碎片化，不利于restore * 碎片化随着时间的推移逐渐加深。这是因为在新的备份中，一个文件的chunks分散在过去不同的备份内。新的备份的碎片化更加严重   解决方案：   * Container capping: 复原一个文件时需要访问的containers越少越好。因此采用了container capping。该方法将stream分解为segments，每一个segment利用最多T个旧containers去重。 * Forward assembly area(FFA)：   + 由于File recipe的存在，我们可以轻易知道未来需要哪些chunks。   + 内存中开辟一块区域叫做FFA，用以组装文件一定范围内的数据。每次从内存中取出一个container后，根据file recipe从container中取出该范围内需要的所有chunks。   + 可以采用rolling的方式管理FFA   File recipe example: |
| 不足：   * container还是通过文件系统存储的，因此会有很多额外的开销(see page 3) |
| Data set:   * HP Storage * cycle30：只保留30天的数据，每增加一个新备份数据就要删除往前数第30天的备份数据 |

|  |
| --- |
| SAR SSD Assisted Restore Optimization for Deduplication-Based Storage Systems in the Cloud |
| 背景：   * 提升文件去重系统的restore性能 * SSD每GB的成本约为HDD的20倍，因此不能全部用SSD替代   方案   * 由于SSD的随机读性能很好，而HDD的随机读性能很差，并且SSD的能耗较低，因此可以将部分chunks存在SSD上，以减少random read以及能源消耗。这些chunks的特点是high reference count, small size，of non-sequential in access patterns * SSD中存有chunks with high reference count，因为reference count越高，越可能被访问到。除此以外，该系统还记录有哪些chunks经常被访问到，这样popular chunks with low reference count也可以被选择记录到SSD中 |
| Datasets:   * R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In FAST’10, Feb. 2010. * Virtual machines for restore |
| 缺点：   * 感觉可以优化read\_list使得homes traces的效果提高 |

|  |
| --- |
| ALACC Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching |
| 背景：   * 在restore的过程中，是可以知道将来需要哪些chunks的(通过Look-ahead window) * Look-ahead window(LAW)常被用于指导cache中chunks/containers的进出。其中chunk based cache在LAW的指导下可以取得更好的性能 * workload会不断变化，如何根据workload调整cache将影响restore的性能   restore时的方式：   * Container based cache: 进出对象为一个container中含有的所有chunks；先读到container read buffer中，再直接加入到container based cache内。   + 由于container中含有在未来restore的过程中不会用到的chunks，因此浪费了很多存储空间   + 将被换出的container有可能含有未来需要用到的chunk，此时增加了不命中率 * 并不是说container based cache一定比chunk based cache差：When most data chunks in the container are used very shortly and there is **very high temporal based locality**, the cache hit ratio of caching containers can be better than that of caching chunks. * Chunk based cache：进出对象对单个的chunk。先读到container read buffer中，再把chunks拷贝到chunk based cache。比container based cache多了一重复制。 * Forward assembly area: 见前面文章。   作者的方法：LAW assisted chunk based cache+ FAA   * 如figure3所示。LAW比FAA大。FAA的大小为多个container size，每一个叫做FAB(Forward Assembly Block)。当第一个FAB复原完成以后，保存并在FAA的末尾增加一个FAB * 当FAB遇到一个未被填过的chunk时。首先查找container based cache，若找到，则在所有的FAB中该chunk出现的位置填入该chunk。若未找到，则将含有该chunk的container读出(称该container为read-in container)，将该container含有的chunks与FAA一一比对，并插入FAA对应的位置。同时根据LAW和FAA的信息，判断read-in container种的其余chunks是否插入container based cache。 * chunk based cache的组织   + 由于LAW比FAA大，此时可以将read-in container中的chunks分为三类。第一类，U-chunk(Unused chunk)，既不再FAA中出现，也不在LAW-FAA中出现的。第二类，P-chunk(Probably used chunk)，仅在FAA中出现，未在LAW-FAA中出现。第三类，F-chunk(Future used chunk)，在LAW-FAA中出现的chunk。   + 将cache分为两段，一段为F-cache，用于存储F-chunks，将未来最先被使用的F-chunk放在前面。一段为P-cache，用于存储P-chunks，用LRU的方式组织。F-cache中被使用过的F-chunk作为MRU移入P-cache。   + 当需要evict的时候，P-cache优先，然后是F-cache的F-chunk of lowest priority * 动态调节FAA，LAW，chunk based cache的大小：   + 每restore一个FAB就调整一次三者大小 |
| Traces:   * EMC 1, FSL 1 |

|  |
| --- |
| Chunk Fragmentation Level: An Effective Indicator for Read Performance Degradation in Deduplication Storage |
| 背景：   * Chunk可能可以在四类地方被发现：   + The current version of its own data stream   + A previous version of its own data stream   + The current version of other data streams   + A previous version of other data streams * 提出了衡量fragmentation的标准 |

|  |
| --- |
| Reducing impact of data fragmentation caused by in-line deduplication |
| 场景：   * 备份系统是一定会发生碎片化的。因为同一个文件的各个部分可能存储于硬盘的不同地方，例如不同的containers中。 * 碎片化随着时间的推移逐渐加深。这是因为在新的备份中，一个文件的chunks分散在**过去不同的备份**内。新的备份的碎片化会变得越来越严重 * The latest version of backup is most likely to restore, so its restore speed should be guaranteed.   提升restore速度的时候需要考虑的因素:   * eliminate reduction in restore bandwidth for the latest backup; * introduce very little write performance drop for ongoing backups; * not degrade deduplication effectiveness and use little additional space; * not require much additional resources like disk bandwidth, spindles and processing power;   Context的定义   * stream context of a block: it is defined as a set of blocks written in this stream immediately after this block * disk context of a block: blocks immediately following this block on disk   Context的影响   * 如果stream context与disk context很接近，那么说明fragmentation很小，读速度会很快。反之则碎片化严重，需要多次寻道并且查找，影响读速度 * 由于去重的原因，对于current backup，其一定会有一部分数据存在于过去的backups中。随着时间的推移。stream context与disk context的差距越来越大，碎片化愈加严重   重写chunks策略：根据minimal rewrite utility、current rewrite utility以及rewrite limit决定是否将一个duplicate重写  Rewrite utility: 反映了disk context与stream context的不相似度. It is defined as the size of the blocks in the disk context of **this duplicate** which do not belong to its stream context, relative to the total size of this disk context.   * Minimal rewrite utility:即以stream context与disk context的**不相似度**作为重写duplicates的thresh hold(see 5.3.2)   + 当一个duplicate的stream context与disk context很不一样时，重写一个duplicate的收益很高，因为可以一次从硬盘中多读出一个有效的chunk，减少一次寻道时间   + 当两者很接近时，出现边际效应，重写一个chunk带来的收益不高，反而降低了去重率，增加了存储成本 * Current rewrite utility(see 5.3.3)：duplicates中前5%的最高的rewrite utility * Rewrite limit: 用以限制总的重写的duplicates的数量 |
| 总结：   * 通过降低去重率来提高读的效率，牺牲了一部分写性能(见下) * 比较stream context与disk context的时候需要从硬盘上读取数据，有额外的I/O开销 * 底层的文件系统造成更大的碎片化 |