海量存储技术 报告

《闪存上的重复数据删除》

摘要：闪存在移动设备、传感器和云服务器上的应用比较常见，它也经常被用于做主存储器的缓存，用以解决存储系统可扩展性问题。闪存设备的容量和续航能力有限，删除 I/O 及缓存的重复数据有利于节省存储空间并减少昂贵的闪存写入。本报告针对闪存及其缓存的重复数据删除主要介绍三种方式。第一种方法 OrderMergeDedup 设计了一种软更新式元数据写入顺序，可以保持存储数据的一致性，而不会产生额外的 I/O。此方法进一步探索 I/O 延迟和合并的机会，以减少元数据 I/O 写入。由于一般研究主要集中在运行时数据管理的效率上，没有考虑程序流程，因此第二种方法提出了一种新的应用感知重复数据删除方法 URD（Unmodified region detection）。URD 重构应用程序的循环块以避免重复的更新操作，旨在减少就地更新数据的大小 ;第三种方法针对闪存缓存提出了一种集成数据缓存和重复数据删除元数据（数据的源地址和指纹）并有效地管理这两个组件的新型体系结构 CacheDedup。此方法还提出了重复感知缓存替换算法（D-LRU，DARC）来优化缓存性能和耐用性。

1. 背景介绍

现在闪存存在的问题有（1）闪存的随机写入性能不佳可能会在写入密集型应用程序（如 DBMS，电子邮件服务器和文件服务器）时导致严重的性能下降;（2）闪存的有限耐用性会损害可靠性，因为磨损的单元数量擦除周期可能不会提供可信的数据。针对这个问题现在有三种解决方案。第一个是在 Flash 转换层（FTL）从文件系统中的逻辑地址到闪存中的物理地址之间的映射，主要关注如何有效地处理写操作。第二种提高闪存可靠性和性能的有效方法是是写缓冲区缓存（WBC）。WBC 的基本功能是临时存储来自上层的写入操作，然后将其发送到闪存。在这个过程中，也希望 WBC 具有设计良好的缓冲区内更新来生成对 Flash 友好的写入模式，例如在 FTL 中没有随机写入和频繁擦除。第三种，试图在闪存块中实现均匀擦除计数分布的磨损均衡是提高闪存耐久性的一种可行的方法。动态耗损均衡方案将每块数据放入新位置就地更新，而静态耗损均衡通过将静态数据重新定位到更常用的位置来进一步考虑对低使用率物理块的利用。

重复数据删除已经广泛地用于节省存储空间和 I/O 负载。I/O 重复数据删除对于数据中心的存储服务器以及个人设备和现场部署的传感系统都很有利。写闪存可能会发生在智能手机和平板电脑上软件包安装过程中或通过频繁使用 SQLite 事务处理时。在网络物理系统中，大量的数据可能被现场部署的摄像机捕获，并被智能交通等应用程序存储或处理。与机械磁盘相比，闪存具有更快的读写访问速度，更低的功耗，更好的抗冲击性和机械延迟。同样地，闪存缓存的应用也很广泛，它作为存储系统典型 I / O 堆栈中基于 DRAM 的主存储器和基于 HDD 的主存储器之间的缓存层，利用此层 I / O 中固有的局部性来提高应用程序的性能。首先，随着整合水平（就整合到单个主机的工作负载数量和整合到单个存储系统的主机数量而言）在典型的计算系统（如数据中心和云）中的增长，可扩展性存储系统成为一个严峻的问题。其次，基于闪存的存储设备的高性能使得闪存缓存成为解决这种可扩展性问题的有希望的选择：通过使用缓存数据为 I / O 提供服务，可以减少主存储上的负载并提高工作负载性能。

重复数据删除系统维护诸如逻辑到物理块映射，物理块参考计数器，块指纹等元数据。重复数据删除需要满足以下条件：1）在系统发生故障后，这些元数据和数据结构必须保持

一致。2）为了满足存储持久性语义，进行持久写入更为重要。3）在闪存上，I/O 重复数据删除还必须最大限度地减少由元数据管理导致的昂贵的闪存写入。对于重复数据删除系统，元数据管理要实现 1）故障一致性：故障发生后，必须保证数据/元数据的写入是快速可连续恢复的。2）高效性：写元数据导致的多余 I/O 操作应对重复数据删除导致的 I/O 操作影响较小。3）持久性：重复数据删除层不应破坏持久性语义地返回 I/O 写操作。

对于闪存重复数据删除，原来工作的主要目标是在运行时在存储器上保留一份重复数据，从性能的角度来看，其中大部分忽略了闪存重复数据删除的关键挑战。1）它们不会减少写入 FTL 的流量。因此，随着 FTL 写入流量的增加，检测冗余的成本也会成比例的增加。2）检测重复数据依赖于散列方法，因此可能导致散列冲突，需要额外的负担来解决。3）需要重新设计 FTL 以支持重复数据删除。但是在商业闪存中，FTL 内部设计不会公开。

2. I/O 重复数据删除——OrderMergeDedup

文章[1]介绍了一种新的 I/O 重复数据删除机制，可以满足重复数据删除的三个要求。具体来说，OrderMergeDedup 对重复元数据和数据进行排序，使得故障的发生不会产生其他数据不一致（持久存储）和未回收的垃圾。这种方法在概念上类似于基于软件更新的文件系统设计，不需额外的 I/O 流量来保持一致性（与日志记录或基于阴影的 I/O 原子性相反），效率很高。原始文件系统软更新存在依赖性循环和回滚问题，但是相对简单的重复数据删除存储结构使我们能够识别和删除所有可能的依赖循环，而不会影响性能。

Dedupv1 [4]和ChunkStash [5]也是通过存储闪存的元数据来实现重复数据删除的，因为元数据如哈希值可以用来查找重复数据，它们利用闪存减少现有的索引信息的延迟。但是他们主要专注于硬盘数据上的重复数据删除。OrderMergeDedup通过合并共享公用重复数据删除元数据块的多个逻辑写入，可以进一步降低元数据I/O开销。特别是，我们有时会延迟I/O操作以期合并未来的元数据I/O。如果由于数据持久性语义导致延迟的I/O被等待，则预期的I/O延迟和合并可能会延长对用户的响应。实验结果表明，当延迟时间被限制时，性能影响是轻微的。由于减少了I/O负载，它甚至可以改善应用程序延迟。本方法通过故障一致性I/O排序和预期合并来实现重复数据删除。

I/O 重复数据删除减少了 I/O 数据流的重复写操作。我们在设备层拦截所有的 I/O 块（4KB），并根据其内容计算哈希指纹，指纹用来识别存储块中的重复部分。重复数据删除系统会维持多余的元数据信息。逻辑到物理映射将逻辑块的访问映射到物理存储内容中。对每个物理块，相应的参考计数器记录了映射到它的逻辑块数量。指纹用来匹配块内容。

2.1 故障一致性的 I/O 排序

针对故障发生时的 I/O 排序问题有以下三种方法。

Journaling. 原子 I/O 操作在写入文件系统之前会被记录在 redo 日志中。当故障发生后，重启系统可以通过执行 redo 日志恢复。

Shadowing. 对现存文件的写操作使用 copy-on-write 技术处理临时的影子块。在投影中，对现有文件的写入以写入时复制的方式处理为临时阴影块。 最后通过一个原子 I / O 写入文件索引块来实现，该块指向更新的影子数据/索引块。故障发生时必须重新编写索引块（可能在多个层次级别）以创建完整的阴影。

软更新. 软更新方法会排序文件系统的写操作，以便任何中间操作故障总是使文件系统结构保持一致性（除了可能的临时写块中的空间泄露问题）。(缺点) 在普通操作中，它不需要 I/O 开销，回滚机制可以解决在块排序中的循环性依赖问题，但是会降低效果。

为了保持故障发生后数据的一致性，前两个方法都需要额外的写 I/O。由于相比文件系统，重复数据删除存储的语义较简单，本文设计了一个重复数据删除 I/O 路径来解决循环依

赖问题。

物理块参考计数器记录了逻辑块的参考数量和物理块的指纹参考。块指纹不用因为最后一个逻辑块对物理块的参考的移除而改变。将它们分成两个故障一致性事务减少了复杂度和循环相依性。它也会使指纹的回收被延迟，提高了效果，因为在一段时间的不存在后，同样的数据也可以被执行重复数据删除。

1) 在连接到逻辑地址或计算好的指纹之前，物理数据块应保持不变。故障可能导致一些数据块不可用，但不会导致逻辑地址或指纹指向错误的内容。
2) 我们保证当故障突发时，唯一可能导致的不一致性是某些物理块的参考计数器高于实际值。高于实际的参考计数器可能会产生垃圾（可以异步回收），而低于实际的参考计数器可能会导致过早的块删除的严重损坏。为了达到这个目的，从某个逻辑地址或指纹指向一个物理块的新链接必须在物理块的参考计数器增加之前，并且相应的解除链接操作必须在物理块的引用计数器减少之前。
3) 同时，逻辑块到物理块的映射和指纹更新可以并行处理，因为它们之间不存在故障一致的依赖关系。

图一解释了在不同写情况下的软更新写顺序。

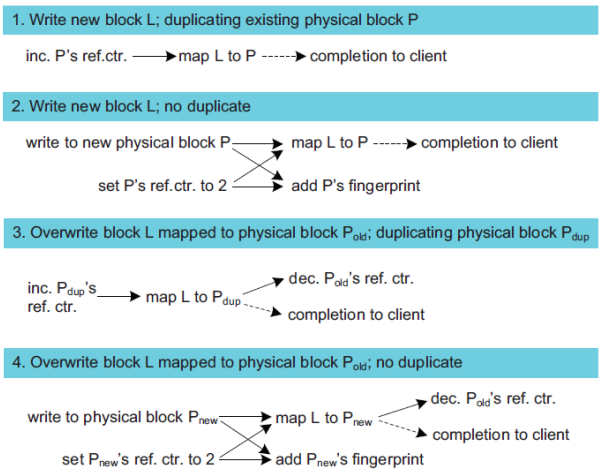

图 1 重复数据删除的 4 种情况

合并：对一个元数据块的写操作可被合并成一个 I/O 操作。这种合并明显减少了 I/O，但是可能导致循环依赖性和死锁。如图 1 中的 3 和 4 场景，若新块的参考计数器 Pdup/Pnew 和旧块 Pold 在同一个元数据块上，合并它们的参考计数器可能会导致死锁。这种情况应该不允许元数据写合并，

我们通过延迟在循环依赖中不重要的元数据的更新来解决这个问题。不重要的更新是指客户端完成信号不依赖的写操作，尤其是图 1 中 3 和 4 情况中 Pold 的减少。在它们的依赖被清除后延迟这些操作会消除依赖循环。因为延迟行为不是完成 I/O 操作的重要步骤，因此客户端 I/O 回应不会受到影响。

2.2 元数据 I/O 合并的有效性

通过延迟元数据更新 I/O 操作使得它们与未来元数据结合写的机会增加。

低持久性. 我们的重复数据删除系统支持两种具有不同性能折衷的持久性模型。具体而言，强大的持久性模型忠实地保留了底层设备的持久性支持——只有在从设备返回所有相应的物理 I/O 操作之后，重复数据消除层才会返回 I/O 操作。另一方面，弱持久性模型执行写入异步，在此之下，写入被提前返回，而相应的物理 I/O 操作可以延迟到下一个刷新或强制单

元访问挂起的请求。在持久性较弱的情况下，I/O 操作可以被延迟以增加元数据 I/O 合并机会。然而，这样的延迟可能会受到某些应用程序和数据库中的同步写入的阻碍。

不重要的 I/O 延迟和合并. 图 1 展示了通过一些不重要元数据更新的延迟而消除了循环依赖。这适用于所有的不重要元数据的写。图 1 中 2 和 4 场景中延迟 P/Pnew 的指纹嵌入操作也会消除循环依赖。突然的系统故障可能会使参考计数器值高于实际值，导致无法回收的垃圾，或丢失物理块重复数据删除机会的一些指纹，但不会出现其他严重的不一致情况。

预期 I/O 延迟和合并. 如果两个元数据执行的间隔时间比重复数据删除系统写入物理设备的典型周期长，那么两个写入同一物理块的元数据通常会导致单独的设备提交。如果这样的间隔很小，则可以给物理设备施加很短的空闲时间（通过停止向其写入数据）来产生更多的元数据写入合并机会。

预期 I / O 延迟和合并可以在高密度的写入请求下获得高收益，因为短暂的预期设备空闲时段将产生高度的合并。另一方面，轻负载提供的合并机会很少，所以预期的设备空闲只会延长 I / O 响应延迟。为了最大限度地发挥预期 I / O 延迟和合并的好处，我们应用一个简单的启发式提示作为指导——重复数据删除系统收到的传入写入请求的频率。直观地说，如果空闲期可以覆盖多个传入的写入请求，则可能发生元数据写入合并。

2.3 实验评估结论

OrderMergeDedup 使用多种移动和服务器工作负载(包安装和更新, BBench 网页浏览，车辆计数，Hadoop 和雅虎云服务基准）来评估基于 Linux 设备映射器的实施。实验表明 OrderMergeDedup 可以减少含有 23%-73%重复数据的工作中 18%-63%的写操作。与备选的基于 shadowing 的 I/O 重复数据删除相比，它的元数据写入开销要小得多。此方法对应用程序延迟有轻微的影响，甚至可能由于减少 I/O 负载而提高性能；

3. 应用程序感知的闪存缓存重复数据删除——URD

现有的重复数据删除方案存在一些限制。1）由于运行时重复识别过程，它们必然带来额外的运行成本。由于它们通常在识别重复操作时使用数据的哈希值，因此除非存在假想的完美哈希函数，否则会导致哈希碰撞。这意味着，为了识别重复数据删除，我们应该为每个写入操作额外地采用冲突解决的开销，对于写入密集型应用程序来说，这是不实际的。2）以前的工作只是假设修改 FTL 内部，这可能会使用地址映射算法与重复数据删除方案相互影响。而且对于不同类型的 FTL，这样的修改将会重复。

作为以前的 FTL 改进结果的补偿方法，文章[2]提出了一种新的应用感知重复数据删除方法 URD（Unmodified region detection），使用正交可执行的重复数据删除方案，而不是修改 FTL 内部。URD 重构应用程序的循环块以避免重复的更新操作，旨在减少就地更新数据的大小。为了减少运行时开销，URD 使用应用程序级方法，将给定的应用程序重写为具有相同语义和冗余写操作的新程序。为此，开发了用于优化编译器的 weexploit 循环优化技术。因此，通过 URD 转化的程序将需要更少的物理操作，而只需写入更新的页面和比原来的块更少的覆盖区域。

另一方面，传统研究主要专注于 FTL 上的重复数据删除。在发现写操作后，计算其哈希值并查看它是否已存在于哈希列表中，从而判断是否要丢弃该操作。为支持这种操作，一些重新设计的 FTL 允许多个逻辑地址映射到一个物理地址上，提高了 FTL 的可靠性和性能。但他们的工作主要集中在运行时数据管理的效率上，没有考虑程序流程。因为应用程序可以独立于 FTL 实现进行管理，因此文章[2]提出基于应用程序的在线重复数据删除方法 URD。在线方案指过滤重复请求，而离线方案消除已经在存储器上的重复数据。这两种方法都能帮助提高执行效果和空间利用率。

3.1 URD 基本设计

URD 利用应用层的优势，使用从应用程序分析的语义信息而不需要散列函数，并且不会修改 FTL 内部，利用寻呼按的重复执行模式来最大限度地提高性能。在 URD 中，通过利用循环平铺和展开等循环优化技术来重建循环，以便在固定大小的单元中执行重复数据删除。

写操作存在循环，通过循环将文件内容从存储器传递到缓冲区，经过处理后再传递到存储器。这种类型的循环通常用于图像处理应用，但是如果在此过程中没有修改操作，则存在重读写操作，导致耗时的擦除操作。URD 转换写循环为只写非重复数据。综上所述，原始循环结构 L 读取给定文件的数据，对其进行处理，并将整个数据覆盖回闪存。URD 将 L 转换成新的循环结构 L'，则 L'读取给定文件的数据，像 L 一样对其进行操作，但只将修改的部分覆盖回闪存。

我们根据重复数据删除单元的大小将 URD 分类为 URDP 和 URDB，分别执行页和块的重复数据删除。图 2 为 URDP 的一个例子。我们假设程序 P 读取由 8 个页面（＃0，＃1，…，＃6，＃7）组成的文件并修改文件的一些数据。程序 P 被 URDP 转换成程序 P'。程序 P'的循环成为循环嵌套，并且插入了用于重复数据删除的附加代码。

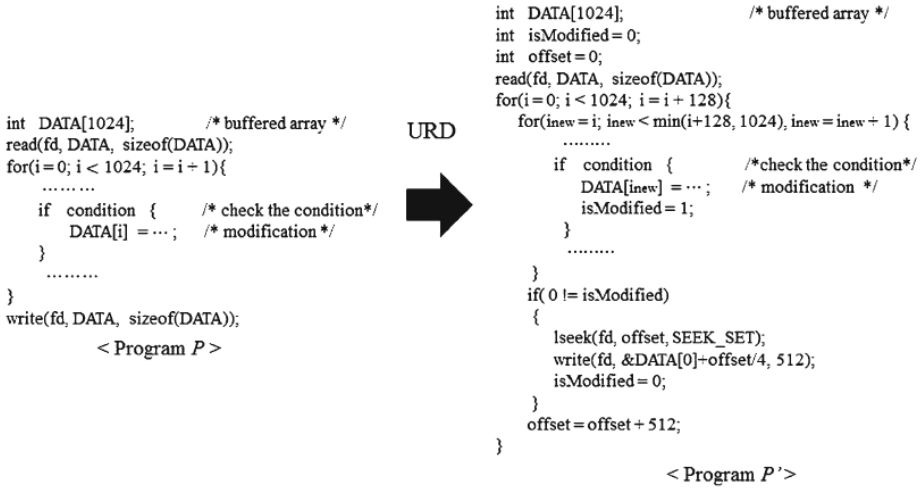

图 2：URDP 算法例子

图 3 显示了 URDP 对 512 字节覆盖页面的影响。我们假设给定文件的一半（＃0，＃2，＃4，＃6）被修改，其他（＃1，＃3，＃5，＃7）在运行时不被修改。即使页面＃1，＃3，＃5 和＃7 的数据在运行时期间没有被修改，程序 P 写入如图 b 所示的 4,096 字节的八个页面（＃0-＃7），程序 P 在循环执行之后立即写入所有数据。也就是说，页面＃1，＃3，＃5 和＃7 变成重复数据。与程序 P 不同，程序 P'只写入 2048 字节（＃0，＃2，＃4，＃6），如图 c 所示。程序 P'检查每个页面大小的数据是否被修改，并只写入修改的数据。URDP 之后的程序通过避免写入重复数据来减少写入的数据量。

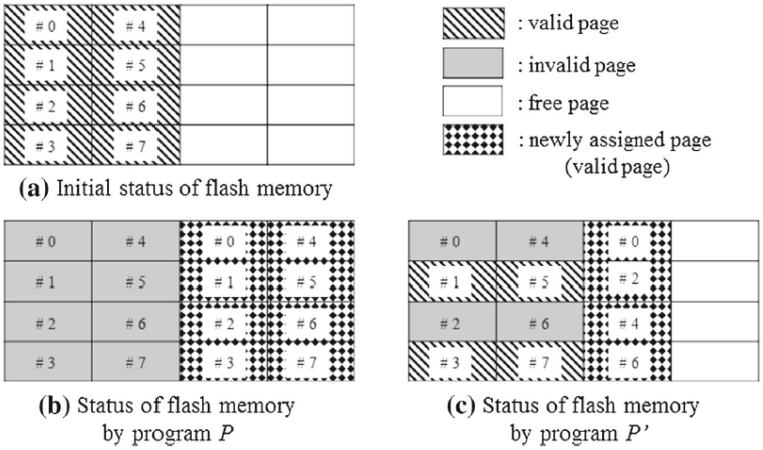

图 3：URDP 对覆盖页面的影响

3.2 URD 模型

循环检验重复数据的步骤如下：

1） 访问模式分析。探究循环是如何访问缓冲区数据。这一步决定了如何转换循环。

2） 循环模块转换。基于上一步来重新设置循环，以页为单位转换。

3） 选择性写的代码指令。为修改的以页为单位的数据插入代码框架，保证了不重复写。

访问模式分析. 计算嵌套循环中的循环个数和所占空间。如图 4 所示，这是一个两层的嵌套循环。L1 和 L2 循环的 movement_unitsize 分别为 400 字节和 4 字节，它们的循环大小分别为 4000 字节和 400 字节。

```
int DATA[10][100]                    //a buffered array
for(i1 = 0; i1 < 10; i1 = i1 + 1){   //L1
    for(i2 = 0; i2 < 100; i2 = i2 + 1){   //L2
        DATA[i1][i2]= … ;
    }
}
```

图 4：两层嵌套循环

循环模块转换. 包括两种循环优化技术：循环展开和循环平铺。

循环展开用几个副本替换循环体，并相应地调整循环控制代码。循环展开减少了迭代的次数，但增加了循环体的大小。:图 5 是一个简单的循环展开的例子

```
for( i = 1; i ≤ 100; i = i + 1 )
{
    s = s + a[i];
}
```

(a) Original loop

```
for( i = 1; i ≤ 99; i = i + 2 )
{
    s = s + a[i];
    s = s + a[i + 1];
}
```

(b) Result of unrolling the original loop by a factor of two

图 5：循环展开例子

循环平铺是一种增加嵌套循环深度的转换。给定一个深度为 n 的循环嵌套，循环平铺可以使它从深度 n + 1 到深度 2n 更深地嵌套。它重新安排迭代空间遍历来改善循环中的缓存重用。下图是循环平铺的一个简单示例。图 a 的一维环路在图 b 中变成一个二维的环形嵌

套，其大小为 2。分块大小是分块循环的迭代次数，即图 b 中的最内层循环，并且通过考虑高速缓存行的大小来调整以减少高速缓存未命中。URD 使用类似于循环平铺的方式将循环转换为循环嵌套，以便经过转换的循环以页为单位访问给定的文件。接着，循环平铺根据缓存大小修改迭代空间，而我们的技术不是通过迭代空间而是通过循环来更改完整执行的访问区域。URD 还像展开循环一样展开循环内的内部循环。

```
for( i = 1; i ≤ n; i = i + 1 )
{
    b[i] = a[i] / b[i];
    a[i+1] = a[i] + 1.0;
}
```

(a) Original loop

```
for( i = 1; i ≤ n; i = i + 2 )
{
    for( j=i; j ≤ min(i + 1, n); j = j + 1)
    {
        b[i] = a[i] / b[i];
        a[i+1] = a[i] + 1.0;
    }
}
```

(b) Result of tiling the original loop with a tile size of 2

图 6：增加嵌套循环深度的转换

选择性写的代码指令. 这一部分是只对修改的数据进行写入，从而在应用软件层执行重复数据删除。代码指令应该被放置在缓冲区数组被修改的页面中。

1） 选择 isModified 变量来标识页面是否被修改过。
2） 通过循环在运行时确定被修改数据的位置 offset，其值是页大小的整数倍。
3） 使用 lseek()和 write()来只写被修改的页面，而不是每个页面。

3.3 实验评估结论

在模拟环境下进行了实验和评估。 URD 通过避免重复数据的写入操作来减少来自应用程序的写入数据的大小。就 FTL 而言，基于页面的重复数据删除会产生耗时的合并操作，而基于块的重复数据删除可以产生具有成本效益的转换操作而不是合并操作。因此，无论写流量的重复程度如何，基于块的重复数据删除总是获得性能提升，而基于页面的重复数据删除的性能取决于写流量的重复程度。此结果与实际应用 GIMP 的仿真结果类似。

4. 闪存缓存重复数据删除——CacheDedup

闪存的有效缓存有以下几个关键的限制。1）随着现代工作负载数据密度的不断提高和系统整合的工作负载数量的增加，相对于商用闪存的容量，缓存容量的需求急剧增长。2）由于闪存会随着写入而磨损，所以使用闪存进行缓存会加重耐用性问题，因为工作负载固有的写入和错过缓存的读取都会导致磨损。

CacheDedup 是一种突破了上述限制的闪存重复数据删除的在线解决方案。首先，重复数据删除减少了工作负载的缓存占用空间，从而允许缓存更好地存储其工作集并减少容量错失。其次，重复数据删除技术减少了强制缺失和容量缺失导致的必要缓存插入次数，从而减少了闪存的磨损，提高了缓存的耐用性。尽管已经有针对基于闪存的主存储器在内的各种存储系统的重复数据删除技术研究，但 CacheDedup 采用了将重复数据删除技术和高速缓存技术整合在一起的新型技术。

高效的缓存重复数据删除需要缓存和重复数据删除管理的无缝集成。为了满足这一需求，CacheDedup 采用独立的数据高速缓存和元数据高速缓存，体现了一种集成数据高速缓存和重复数据删除元数据（数据源地址和指纹）的新颖架构。这个设计解决了两个关键问题。首先，它允许 CacheDedup 绑定元数据的空间使用情况，使其足够灵活，可以部署在存储系统的客户端或服务器端，并以软件或硬件实现。其次，它能够在数据从数据高速缓冲存储器中删除后优化在元数据高速缓冲存储器中缓存历史源地址和指纹。这些历史重复数据删除元数

据允许 CacheDedup 使用缓存指纹快速识别重复，并在再次引用这些源地址时生成缓存命中。

基于这种架构，我们进一步研究了可以利用重复数据删除来提高闪存缓存性能和耐用性的重复数据缓存替换算法。首先，我们介绍 LRU 的重复感知版本 D-LRU，可以通过在数据和元数据高速缓存上强制执行 LRU 策略来有效实现。其次，我们提供了一个 ARC 的重复感知版本 D-ARC，利用 ARC 的抗扫描能力来进一步提高缓存性能和耐用性。这两种算法不会导致数据和元数据缓存中的浪费，并且可以有效地利用空间。

4.1 需要集成的闪存重复数据消除

基于闪存的存储的出现极大地促进了在网络存储系统的客户端和服务器端采用闪存缓存。但是，闪存缓存仍然面临严重的容量和耐力限制。鉴于日益增加的数据密集型工作负载和不断增加的存储整合水平，商品闪存设备的规模相当有限。使用闪存进行缓存也会加剧闪存设备的磨损问题，因为不仅工作负载的写入会导致磨损，而且还会由于缓存未命中而插入到缓存中的所有读取。

重复数据删除是一种消除重复数据的技术，用于减少主存储的数据占用空间以及备份和归档存储。它经常使用抗碰撞密码散列函数来识别数据块的内容，并发现重复的数据块。重复数据删除有可能解决闪存缓存面临的上述限制。通过消除重复数据的高速缓存，能减少数据占用空间，允许闪存缓存更有效地捕获 I / O 工作负载的局部性并提高性能；它还减少了写入闪存设备的次数和相应的磨损。

尽管可以采用现有的闪存缓存和重复数据删除解决方案，并将它们堆叠在一起以实现缓存重复数据删除，但直接将这两个层面集成会导致效率低下。一方面，在重复数据删除层上堆叠缓存层是不可行的，因为前者不能利用后者实现的空间减少。另一方面，在缓存层上简单堆叠重复数据删除层也有严重的局限性。首先，去重复层必须管理整个主存储器的指纹，并且可能做出对高速缓存有害的指纹管理决定，例如，在良好的位置上去除属于数据的指纹并在高速缓存中造成重复的副本。其次，缓存层不能利用数据重复来改善缓存管理。相比之下，CacheDedup 采用集成设计来优化闪存缓存重复数据删除的性能和耐用性。

最近的工作 Nitro 研究了对服务器端闪存缓存使用重复数据删除和压缩。Cache-Dedup 是 Nitro 在其新的体系结构和用于重复感知缓存管理的算法中的补充。此外，我们的方法可以用于使缓存管理同时知道压缩和重复数据删除，并改进使用这两种技术的解决方案。重复数据删除可以在 I / O 路径中或线下执行。CacheDedup 进行在线重复数据删除，以防止任何重复块进入缓存，从而最大限度地减少数据占用和磨损。

4.2 感知重复数据删除的缓存管理

CacheDedup 存在一些缓存替换方案。具体而言，为突破时间局部性，广泛使用的 LRU 算法会删除缓存中最近最少使用的条目。从理论上讲，它已经被证明在最坏的情况下具有最好的保证。但它不是“抗扫描”的，即只访问过一次的条目会占满缓存，减少了重复访问条目的可用空间。ARC 是一种自适应算法，在缓存替换中考虑缓存被击中的时间和频率。这是“抗扫描”的，为许多工作负载提供更好的性能。

然而，高速缓存替换算法通常关注的是最大化命中率，忽视与硬件设备的寿命和磨损有关的任何问题，这对于基于闪存的高速缓存来说是不利的。有方法尝试通过绕过缓存来减少写入，但是会降低命中率。现在面临的挑战是如何在保持命中率接近最佳值和降低写操作次数之间找到“最佳点”。而 CacheDedup 通过优化 LRU 和 ARC 算法，并通过集成的重复数据删除和高速缓存管理体系结构来实现其重复感知缓存替换算法，从而解决了这一难题。

4.3 CacheDedup 结构

CacheDedup 将缓存的管理和重复数据删除结合起来，并为这两者的结合设计了新型结构。传统的缓存层需要管理从主存储器上的源地址到缓存设备块的缓存地址的映射。重复数据删除层需要追踪数据块的指纹从而鉴别重复数据块。

CacheDedup 解决了两个问题。1）不像传统缓存，源到缓存地址的映射数量与缓存大小无关，因为重复数据所以存在多对一的地址映射。2）即使高速缓存必须跟踪的指纹数量受到高速缓存大小的限制，CacheDedup 也可以跟踪当前存储在高速缓存中的数据的指纹。具体而言，保留已经从高速缓存中删除的块的历史指纹是有利的，以便当这些块再次被请求时，CacheDedup 不必再从主存储器中取出它们。当 CacheDedup 被用作客户端缓存时，这样的优化尤其重要，因为它可以减少昂贵的网络访问。但是，指纹存储仍需遵守 CacheDedup 的空间使用限制。

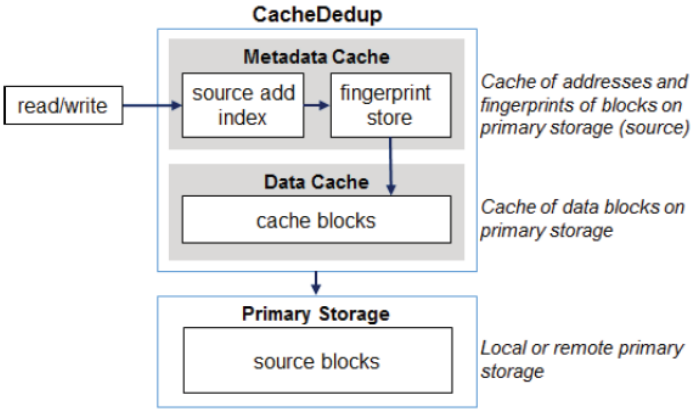

图 7：CacheDedup 结构示意图

CacheDedup 的结构如图 7 所示。Metadata Cache 数据结构是保存源地址及其指纹的缓存。这个设计解决了将元数据的管理作为缓存替换的问题，其中，Metadata Cache 和用于存储数据的 Data Cache 是分开的。Metadata Cache 包括两个重要的数据结构，第一个结构 source address index 是从主存源地址到 Metadata Cache 指纹 的一个映射。每个缓存源地址都与缓存指纹相关联。由于重复数据删除，多个源地址可被映射到同一个指纹上。Fingerprint store 存储着从指纹到 Data Cache 中的块地址的映射。它还包含历史指纹，其相应的块并没有存储在数据高速缓存中。当历史指纹指向的数据块被带回到数据缓存时，映射到该指纹的所有源地址在再次被引用时可以生成缓存命中。每个指纹都有一个引用计数来指示包含相同数据的源块的数量。当它下降到零时，指纹将从元数据缓存中移除。

4.4 实验结果及评估

实验结果表明，CacheDedup 引入的开销很小。重复数据删除可以在固定大小块或内容定义的可变大小块的粒度上完成，其中后者可以实现更大的空间减少但成本更高。CacheDedup 以缓存块的粒度来选择重复数据删除，这符合闪存缓存的结构，并有助于设计能够进行重复的缓存替换。研究结果也证实，使用固定大小的缓存重复数据删除技术可以达到很好的数据缩减水平。

与传统的缓存管理相比， CacheDedup 大大提高了 I/O 性能（丢失率降低了 20%，延迟降低了 51%）和闪存耐久性（发送写入降低高达 89%到高速缓存设备）。CacheDedup 对算法进行了严格的分析，证明它们不会浪费宝贵的缓存空间，并且在时间和空间使用方面效

率很高。这也表明，所提出的体系结构和算法可以扩展到支持压缩和重复数据删除的组合，以提高闪存缓存的性能和耐用性。

5. 结论

本报告通过主要阅读三种方案针对在闪存及其缓存上存在的重复删除数据问题进行研究探索。

第一个方案提出了一个新的 I/O 机制 OrderMergeDedup，设计软更新形式的元数据写入顺序，并使用预期 I/O 延迟来减少元数据 I/O 写入操作，保证故障一致性和高效率。

第二个研究 URD 是一个 FTL 层的重复数据删除方案。它利用循环代码建立循环结构，通过分析嵌套循环是如何访问缓冲区数组、允许变形循环以页或者块为单位访问缓冲区数组、通过嵌入代码框架来识别页或块是否被修改，从而消除闪存上的重复数据写操作。

第三种方案 CacheDedup 是第一个对闪存缓存重复数据进行管理的研究，他使用了新缓存替代策略 D-LRU 和 DARC，将元数据和数据缓存无缝连接在一起，提高了重复数据缓存管理效率和持久性。实验验证 CacheDedup 提高了缓存命中率，降低了 I/O 延迟，减少了发送至缓存的写操作数量。

参考文献（前三个为主要参考文献）：

[1] Chen Z, Shen K. OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash[C]//FAST. 2016: 291-299.

[2] Paik J Y, Chung T S, Cho E S. Application-aware deduplication for performance improvement of flash memory[J]. Design Automation for Embedded Systems, 2015, 19(1-2): 161-188.

[3] Li W, Jean-Baptise G, Riveros J, et al. CacheDedup: In-line Deduplication for Flash Caching[C]//FAST. 2016: 301-314.

[4] Meister D, Brinkmann A (2010) dedupv1: improving deduplication throughput using solid state drives (SSD). In: Proceeding of IEEE conference on massive data storage (MSST), pp1–6

[5] Debnath B, Sengupta S, Li J (2010) ChunkStash: speeding up inline storage deduplication using flash memory. In: Proceeding of USENIX conference on file and storage technologies (FAST)