DOI:10.11918/j.issn.0367-6234.201701022

FusionCache:采用闪存的 iSCSI 存储端融合缓存机制

孟祥辉[1,2], 曾学文[1], 陈 晓[1], 叶晓舟[1]

(1.中国科学院声学研究所 国家网络新媒体工程技术研究中心, 北京 100190; 2.中国科学院大学, 北京 100049)

摘 要: 针对原生的 iSCSI 目标端控制器缺乏独立的缓存模块问题,为了进一步提高存储区域网的整体性能,在 iSCSI target 软件中引入了一种基于闪存的融合缓存机制 FusionCache. FusionCache 利用闪存和 DRAM 组成统一的融合缓存架构,闪存充当 DRAM 的扩展空间,DRAM 分为缓存块元数据区和前端缓存区. 元数据区基于基数树管理缓存块元数据,用于加速缓存块的查找;前端缓存区基于回归拟合统计并预测缓存块访问热度,并吸收大量写入对闪存带来的冲击,只允许热点数据进入闪存. FusionCache 采用改进的 LRU 算法对缓存块进行替换,并且在写回过程中考虑 iSCSI 会话状态. 实验结果表明:FusionCache 能降低对后端磁盘设备的访问频率,提高 I/O 响应的速度和吞吐. 与只采用 DRAM 的缓存机制以及原生 iSCSI target 相比,FusionCache 的 I/O 访问延时分别降低了 33%和 60%,吞吐分别提高了 25%和 54%;相较于 Facebook 提出的 Flashcache 机制,FusionCache 的吞吐性能提高了 18%,延时降低了 27%;FusionCache 还具有良好的读缓存命中率;此外,FusionCache 能够减少闪存的写入次数,提高闪存使用寿命. FusionCache 提供良好的网络存储效率,并且降低了使用成本.

关键词: 网络存储性能;缓存机制;iSCSI target;闪存

中图分类号: TP393 **文献标志码**: A **文章编号**: 0367-6234(2017)11-0066-07

FusionCache: A Fusion Cache Mechanism for iSCSI Target Based on Flash Memory

MENG Xianghui[1,2], ZENG Xuewen[1], CHEN Xiao[1], YE Xiaozhou[1]

(1.National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China; 2.University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Focusing on the problem of lack of independent cache module of original iSCSI target controller, we introduce a fusion cache mechanism based on flash memory called FusionCache into the iSCSI target software to further improve the overall performance of the storage area network. FusionCache uses flash memory and DRAM to form a unified fusion cache architecture. The flash memory acts as DRAM's expansion space, and DRAM is divided into cache block metadata area (metadata cache) and front-end buffer area (front cache). The metadata cache manages cache block metadata based on radix tree in order to accelerate the cache block searching; the front cache tallies and predicts the access popularity of the cache block based on regressing fitting model, and absorbs the impact of massive writes on flash memory to ensure that only the hot data is allowed to enter the flash memory. FusionCache uses the improved LRU algorithm to do cache replacement. Besides, it takes iSCSI session's state into account during write-back. The experimental results show that: FusionCache is able to reduce access to backend disk devices, and improve I/O response speed. FusionCache reduces I/O access latency by 33% and 60%, and improves throughput by 25% and 54%, compared to cache mechanism with only DRAM and original iSCSI target, respectively. Compared with Flashcache proposed by Facebook, FusionCache improves throughput by 18% and reduces latency by 27%. FusionCache also has a good read cache hit rate. Besides, FusionCache reduces write amount of flash memory, thus extends its life. FusionCache provides good efficiency of network storage and reduces cost.

Keywords: network storage performance; cache mechanism; iSCSI target; flash memory

近年来存储介质技术的进步和个人云存储业务的迅速发展,使得网络存储系统再次成为学术界和工业界研究的热点,存储区域网(Storage Area Network,SAN)是解决数据密集型应用 I/O 性能瓶颈的重要手段,其中 IP SAN 采用 iSCSI[1] 协议. 在云计算和大数据时代,海量的网络数据尤其是视频流量爆炸性增长给存储系统带来了巨大的挑战. 如何在大规模和高负载的应用环境下,进一步提高网络存储系统的性能成为亟待解决的问题.

作为网络存储系统的一个重要部分,缓存系统一直以来是存储领域提高性能需要研究的关键课题. 文献[2-3]利用本地磁盘缓存远端目标设备上

收稿日期: 2017-01-05
基金项目: 中国科学院战略性先导科技专项课题(XDA06010302),中科院创新研究院前瞻项目(Y555021601)
作者简介: 孟祥辉(1990—),男,博士
通信作者: 叶晓舟,yexz@dsp.ac.cn

的数据，来降低访问延时，减轻存储服务器的负载. 然而，磁盘性能受限，固态硬盘（Solid State Drive，SSD）等基于闪存的设备更适合用作网络存储的缓存. 文献[4-5]利用 SSD 作为内存的补充来构建主机端的高速缓存架构，并在网络存储环境下，针对 SSD 缓存提出几种不同的改进 cache 策略，其性能提升比磁盘缓存优势明显；在云架构中，文献[6-7]利用 SSD 缓存提高虚拟机（Virtual Machine，VM）环境中的存储性能.

以上研究大多是基于客户端缓存进行研究，很少对 iSCSI target（即 IP SAN 的存储端）软件进行优化. 文献[8-9]基于 iSCSI target 端，从网络设备[8]缓存和 NUMA 节点[9]缓存的角度进行研究，但并未利用到高速的闪存设备进行加速. 文献[10-12]在存储端利用闪存构建高速缓存架构，保证响应时间的同时提高闪存设备的寿命，是针对操作系统通用块层或者文件系统级的优化，而 iSCSI target 控制器软件本身依然缺乏独立的缓存模块. 相关工作都是从 iSCSI target 软件之外的角度进行研究，很少关注 iSCSI target 软件本身. 针对以上问题，为进一步提高 SAN 的整体性能，本文为 iSCSI target 控制器软件提出独立缓存模块，即基于闪存的 iSCSI 存储端融合缓存机制 FusionCache. 与先前工作最大的不同是本文考虑 iSCSI target 自身的处理特征，着重研究为 iSCSI target 软件引入基于闪存的缓存机制. FusionCache 利用闪存和 DRAM 共同构成统一的块 I/O 缓存空间，闪存作为 DRAM 的扩展，而非传统的第二级缓存. DRAM 划分为两个区间，分别用于快速查找缓存块和提高闪存耐久性. 闪存的缓存空间利用改进的最近最少使用（Least Recently Used，LRU）原则组织. FusionCache 能降低对后端磁盘设备的访问频率，提高 I/O 响应的速度和吞吐.

1　缓存架构

传统使用 SSD 闪存的方法，一般把 SSD 当作二级缓存，或者 SSD 与 HDD 构成混合存储，对 DRAM 不命中的请求再下发到 SSD，但两级缓存的请求访问模式有很大不同. 本文考虑到这一点，在设计系统架构时，把 SSD 当作 DRAM 的扩展空间，DRAM 空间有限，还要保证对缓存命中与否的快速判断，所以只把缓存块（Cache Block）元数据单独存放在 DRAM，SSD 存储缓存块内容. DRAM 与 SSD 联合构成完整的缓存空间，对后端设备表现为统一缓存，并对客户端透明.

iSCSI Enterprise Target（IET）控制器软件分为用户层和内核层，对数据的处理工作大部分在内核层完成. IET 以 block I/O（即 bio）的形式对 I/O 请求进行处理，不经过虚拟文件系统以及 Linux Page Cache，软件本身没有针对 block I/O 的缓存模块. IET 对用户请求的封装结构为 tio（即 target I/O），因此需要在请求递交给通用块层之前截获 tio，并根据提出的缓存策略对请求进行处理.

基于以上考虑，在 IET 软件中提出的 FusionCache 架构见图 1. IET 软件本身包括 3 个组件：NTHREAD，WTHREAD 和 iSCSI Response，WTHREAD 负责 iSCSI 数据（即 tio）的读写流程，因此 FusionCache 最适合嵌入在 WTHREAD 中. 提出的缓存架构中，tio 解析模块负责提取出目标数据的位置信息，包括目标扇区和对应的内存地址；DRAM 管理模块负责管理 DRAM 上的 metadata cache 和 front cache 的空间；SSD 管理模块对闪存上的缓存块进行组织管理；LRU 链表使用改进的 LRU 算法对缓存块进行替换；bio 构造模块根据缓存策略向通用块层发出 bio 请求.

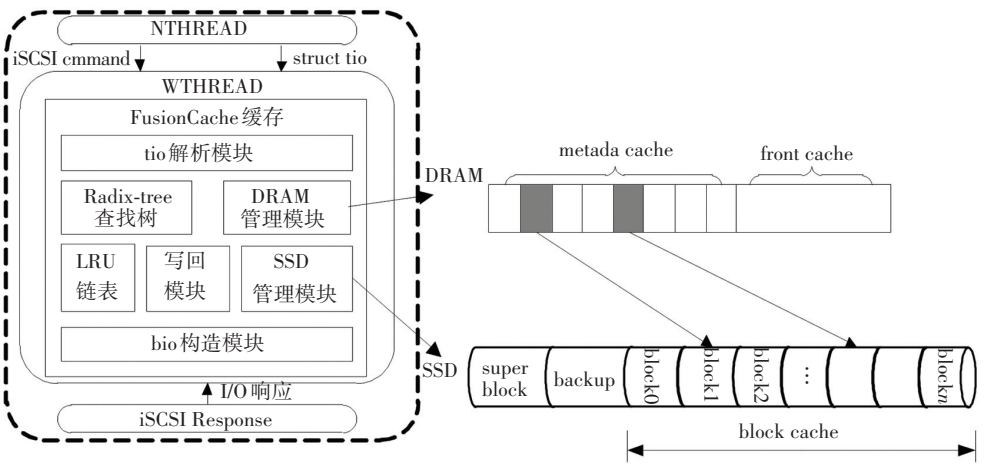


图 1　IET 缓存系统架构

Fig.1　IET cache system architecture

2 设计与实现

2.1 DRAM 中的缓存设计

DRAM 缓存空间包括 metadata cache 和 front cache,前者管理缓存块元数据,后者担当 SSD 的前端,对进入 SSD 缓存的数据进行筛选.

1) metadata 数据结构

为保证缓存算法的有效性和一致性,metadata 数据结构须至少包含以下信息,见图 2.

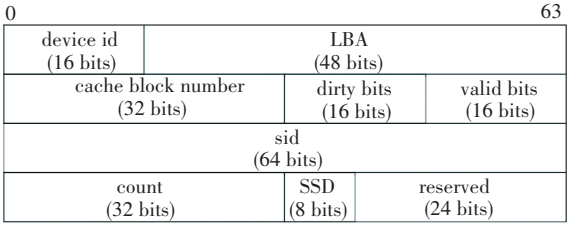


图 2 metadata 数据结构

Fig.2 Data structure of the metadata

元数据中每个字段代表的含义如下:

device id: 标识每一个后端磁盘设备或者 LUN (Logic Unit Number);

LBA:即 Logic Block Address,代表逻辑数据块 block 在一个 LUN 内的偏移或者编号;

cache block number: 标识 SSD 上一个缓存块的位置或者编号;

dirty bits: 脏数据标记,默认情况下每个缓存块包含 16 个扇区. 一个扇区在缓存中被修改而尚未同步到磁盘上时则为 dirty(脏)状态,对应的 dirty 位为 1,反之对应的 clean 状态为 0. 16 个扇区对应 16 个 dirty 位,只要有一个扇区为 dirty,则该缓存块也为 dirty.

valid bits: 与 dirty bits 类似,每一个 bit 代表缓存块中的一个扇区是否有效,1 代表有效. 考虑到某个请求对齐到磁盘逻辑块之后可能不满一个 block 块大小,部分扇区的读写则无意义,所以用 valid bits 标识有效数据,以免不必要的读写.

Sid:标识一个 iSCSI 会话,用于辅助 write back 过程;

count: 缓存块访问计数,用于热度预测;

SSD:标识当前缓存块是否在 SSD 上,因为在某个短暂时间内缓存块可能在 front cache 中;

reserved: 预留位,方便后续扩展功能.

2) 基于 radix tree 的缓存查找

考虑到 hashtable 存在冲突的情况,一旦冲突需要二次查找,而且 hashtable 的大小是固定值,不容易确定. FusionCache 基于 radix tree 快速查找闪存中的缓存块. 此外,radix tree 支持并行查找操作,可以方便地在多核平台上进行扩展和优化.

FusionCache 根据 16bits 的 device id 和 48bits 的 LBA 构成一个 64bits 的变量 index,并以此 index 作为查询的索引. 因此,radix tree 保存了目标逻辑块地址和缓存块元数据的映射关系. 图 3 表示一个简化的查询过程.

假设 index 的高位全为 0,低 18 位有效,每 6 位一组. 树高度为 3,每个节点 slot 数量为 64. 对于 index1,高位 000 000 对应第一层的第 0 号 slot,001 000 对应第二层的第 8 号 slot,低位 010 000 对应第三层的第 10 号 slot,叶子节点对应一个缓存块元数据,其 index 索引值即为 528.

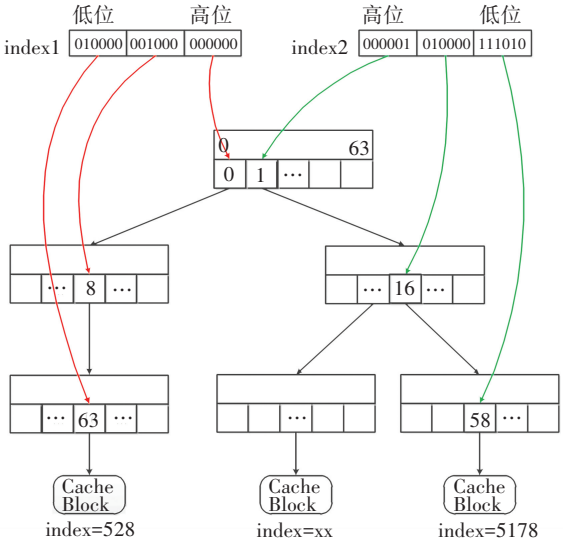


图 3 radix tree 查询过程

Fig.3 Searching process based on radix tree

3) 基于回归拟合的热度预测

由于 SSD 的写入寿命有限,需要考虑减少写入和擦除带来的损耗. FusionCache 使用内存中的一个区间构成 front cache,担当 SSD 的前端来吸收大量写入带来的冲击. 所有的缓存数据首先要进入 front cache,当 front cache 填满之后,根据 metadata 中的 count 信息统计其中缓存块的热度,并预测缓存块在未来被访问的概率. 只有热点数据才进入 SSD,冷数据直接写入后端磁盘.

相关研究表明,存储设备中的文件热度与访问情况之间符合 Zipf 分布[13]. 即存储设备上的 N 个文件依据热度(访问频率)降序排序,则第 k 个文件的访问频率为 $C/k^{1-\theta}$,参数 C 为

$$C = 1/(1 + 1/2 + 1/3 + \cdots + 1/N).$$

此即为 Zipf 分布定律

$$Z(k) = C/k^{1-q}, C = 1/(\sum_{k=1}^{N} 1/k^{1-\theta}). \quad (1)$$

式中 $Z(k)$ 为第 k 个文件被访问的概率. Zipf 分布表明,在一段时间内,热度越高的文件被再次访问到的

概率也越高. 文件的访问热度等于其所有数据块的热度，Zipf 定律同样适用于数据块.

设缓存块 b_i 的热度为 $h_i(t) = m_i(t)/N$，其中 m_i 为时间 t 之前缓存块 b_i 的访问次数，N 为所有缓存块的总的请求次数.

由于访问热度会随着时间动态改变，下一时刻的热度需要根据历史统计信息进行预测. 本文采用基于回归拟合的方法对缓存块的热度进行预测，具有较高的精度，而且复杂度较低. 根据预测结果对 front cache 中的缓存块进行预判，热度低的数据不允许进入 SSD.

本文以每次 front cache 填满作为一个周期统计各个缓存块的请求次数，一个周期内总共记录 N 次请求，连续统计 L 个周期，来预测第 $L+1$ 个周期的访问热度(注意前 $L-1$ 个周期没有进行预测). 在第 l 个周期，设各个缓存块访问次数为 $c_1^l, c_2^l, \cdots, c_{N_B}^l$，则 $N = \sum_{i=1}^{N_B} c_i^l$. L 个周期总计 $N \cdot L$ 次请求，代表了各个缓存块一段历史期的访问频率，同时也反映了访问热度的趋势. 根据这 L 个周期的统计信息，通过回归拟合的思想，可以预测到缓存块在未来一段时间内的热度. 具体方法为：

设缓存块 B_i 在前 L 个周期的访问记录为 $\boldsymbol{C}_i = (c_i^1, c_i^2, \cdots, c_i^L)^{\mathrm{T}}$，其中 c_i^l 为 B_i 在第 l 个周期的记录，$l = 1, 2, \cdots L$.

鉴于预测精度和计算量的均衡，相关研究表明周期数 L 在 5~10 之间比较合适[13]. 由于在 $L = 5 \sim 10$ 之间时，访问热度不会大幅波动，同时为了减少回归拟合计算的负担，选择二次回归便可以实现较高的预测精度.

FusionCache 的一元二次回归模型为

$$c_i^l = a_2 l^2 + a_1 l + b + \varepsilon, \varepsilon \text{ 服从 } N(0, \sigma^2). \quad (2)$$

式中：c_i^l 为访问频率，l 为周期数. 设 $l_2 = l^2, l_1 = l$，即把 l^2 和 l 看作 a_2, a_1 的系数，则上述模型可以转化成二元线性回归模型

$$c_i^l = l_2 a_2 + l_1 a_1 + b + \varepsilon. \quad (3)$$

其系数矩阵为 $\boldsymbol{T}$

$$\boldsymbol{T} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 2^2 \\ 1 & 3 & 3^2 \\ \cdots & \cdots & \cdots \\ 1 & L & L^2 \end{pmatrix}. \quad (4)$$

设模型的预测参数为 $\boldsymbol{P}_i = (b, a_1, a_2)^{\mathrm{T}}$，则 $\boldsymbol{P}_i$ 的预估值 $\overline{\boldsymbol{P}_i} = (\boldsymbol{T}^{\mathrm{T}} \cdot \boldsymbol{T})^{-1} \cdot \boldsymbol{T}^{\mathrm{T}} \cdot \boldsymbol{C}_i$，设 $\boldsymbol{A} = (\boldsymbol{T}^{\mathrm{T}} \cdot \boldsymbol{T})^{-1} \cdot \boldsymbol{T}^{\mathrm{T}}$，则 $\overline{\boldsymbol{P}_i} = \boldsymbol{A} \cdot \boldsymbol{C}_i$. 矩阵 $\boldsymbol{T}$ 仅与周期数 L 有关，则 L 确定后，矩阵 $\boldsymbol{T}$ 与参数 A 可以在 FusionCache 的初始化阶段离线计算，以减轻实时计算负载. 然后，可以预计 B_i 在第 $L+1$ 个周期的访问频率为

$$c_i^{L+1} = (1, (L+1), (L+1)^2) \cdot \overline{\boldsymbol{P}_i} = (1, (L+1), (L+1)^2) \cdot (\boldsymbol{A} \cdot \boldsymbol{C}_i), \quad (5)$$

其中，$(1, (L+1), (L+1)^2)$ 和 $\boldsymbol{A}$ 都能在初始化过程中离线计算得到. 最后可以得出访问热度的动态预测值：$h_i^{L+1} = c_i^{L+1}/N$.

根据以上对访问热度的统计和预测，设定一个热度阈值 $H_{\text{threshold}}$，对 front cache 缓存块划分两个队列 q_1, q_2：

$$\begin{cases} h_i \geqslant H_{\text{threshold}}, q_1 = \{B_i, \cdots, B_m\} \\ h_j < H_{\text{threshold}}, q_2 = \{B_j, \cdots, B_n\} \end{cases}$$

q_1 中的数据写入到 SSD，q_2 中的数据直接写到后端磁盘.

2.2　闪存中的缓存设计

2.2.1　闪存数据布局

如图 1 所示，SSD 可以划分为 3 个区域，分别为 superblock 区，metadata backup 区，以及存放缓存块的 cache block 区. 其中，superblock 是整个 SSD 缓存的“元数据”，保存 SSD 缓存的配置信息，比如缓存使用情况、缓存块大小等. backup 区是 DRAM 中 metadata cache 的备份，保证缓存数据的非易失性或者持久性，在服务器故障重启时，凭借 SSD 中的 metadata 备份即可恢复原有的缓存数据到 DRAM 中，不需要从后端磁盘重新读取重复数据. Cache block 区则存放实际的缓存块.

缓存操作的基本单位是 cache block，如果设置太小，则缓存块元数据会占据太多空间；如果 block 太大，则算法精度会降低，算法会失真. 不同的应用场景，I/O 访问模式不同，即便是同样的数据库应用，I/O 大小也会变化. 因此缓存块大小的设定没有普适性的值. Flashcache 和 dm-cache 的默认缓存块大小为 4K，SQL Server 和 Oracle 默认的块大小都是 8KB. 本文默认使用 8K 大小的缓存块，占用两个内存 page，有利于数据预读和减小元数据所占空间. 当然，也可以根据特定应用场景的 I/O 访问模式自定义缓存块大小并保存到 superblock 中.

2.2.2　改进的 LRU 算法

SSD 缓存根据数据访问的时间局部性原则，不仅考虑缓存块上一次访问的时间，同时结合前文所述的访问热度进行缓存替换.

进入 SSD 的缓存根据热度分为 3 个级别：hot(热点数据)，warm(暖数据)和 cold(冷数据)，算法

分别使用 3 个不同级别的 LRU 队列（Q_{hot}，Q_{warm}，Q_{cold}）管理对应的 cache 数据，见图 4.

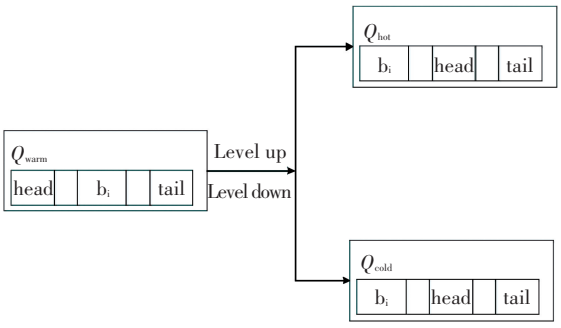


图 4 改进的 LRU

Fig.4 Improved LRU

热度高的队列中 cache 比热度低的队列 cache 块生存时间更久，Q_{warm} 和 Q_{cold} 队列中的 cache 块被命中时，则分别将其升级到更高一级队列中，Q_{hot} 和 Q_{warm} 队列中 cache 块在一段时间内没有被命中则会降级到更低一级的队列中. 当 SSD 缓存空间用尽发生替换时，只在 Q_{cold} 中进行，不需要 DRAM 的 cache 信息进行判断. 所以，cache 块的访问频率能够在不同级别的 LRU 队列得到体现.

2.2.3 write-back 过程

iSCSI 协议允许 initiator 和 target 之间建立多个会话，每个会话允许使用多个连接. 所以，SSD 缓存块的写回过程应该考虑会话状态.

SSD 缓存空间较大，发生写回的频率较低，cold 数据可能较长时间停留在 SSD 缓存，尤其是一个会话结束以后，相应的缓存块应该被写回到磁盘上. 因此，write-back 策略根据 metadata 中的 sid 判断缓存块是否属于一个结束的会话，然后把属于已经结束的会话的全部缓存块都写回到磁盘. 若缓存已满，需要发生替换时，首先需要检查被替换的缓存块的 dirty bits，对于 dirty 的缓存块应立即写回磁盘再替换，而 clean 状态的缓存块可以立即替换.

2.3 读写流程

图 5 展示了读请求的处理流程，写过程与读相似. 需要注意的两点：一是无论对 SSD 缓存命中与否，都需要构造 bio 请求，因为对 SSD 和 HDD 磁盘的访问都属于块请求；二是无论是读还是写过程，都需要图 5 左下角的写前端缓存区过程，右半部分虚线框内为写前端缓存区子过程.

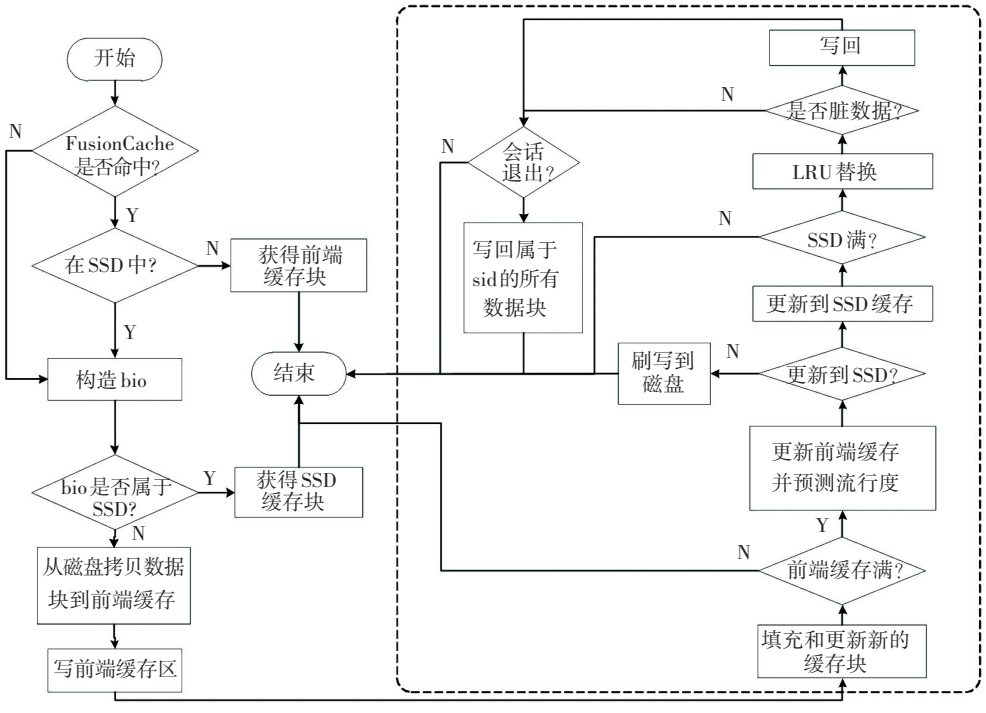


图 5 缓存读流程图

Fig.5 Cache read flow diagram

当一个读请求到达 target 端，FusionCache 首先检查是否命中缓存，若命中则存在两种可能：命中的缓存在 front cache，则 cache block 将立即返回；命中的缓存在 SSD 中，则需要构造到 SSD 的 bio 请求，然后返回相应的 cache block. 若未命中，则需要构造到 HDD 的 bio 请求，然后缺失的 cache block 会从 HDD 拷贝到 front cache，即写前端缓存区过程：首先分配一个新的 cache block 并填充相关数据，若 front

cache 此时未满则此次请求处理结束，继续处理其他请求；若 front cache 已满，则启动上述基于回归拟合的热度预测算法. 根据算法结果，热数据移动到 SSD，冷数据直接进入 HDD. 之后检测 SSD 状态，若 SSD 未满则此子过程返回；若 SSD 已满，则利用改进的 LRU 替换策略驱逐 Q_{cold} 中的 cache block. 在驱逐 cache block 时，FusionCache 同时考虑 dirty 位状态和 iSCSI 会话状态. 若 dirty 位被置 1，被驱逐的 cache block 须先同步到 HDD；若检测到某个会话已经 logout，那么所有属于该会话的 cache block 将会被写回到 HDD.

3　实验与分析

使用 IOMeter 测试 FusionCache 性能，作为对比，测试原生的以及只使用内存 DRAM 作为缓存的 iSCSI target，同时对比 Facebook 开源的 Flashcache 在 iSCSI target 端的表现. 此外，使用两种实际应用场景负载记录（workload trace）进行测试，I/O trace 分别为 Websearch[14] 和 Ads[15]. 测试环境见表 1.

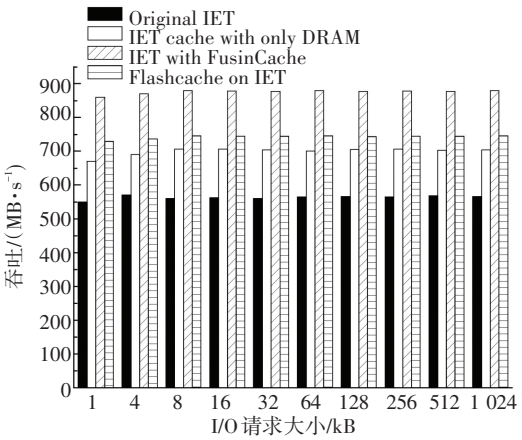
表 1　测试环境配置

Tab.1　Experiment environment

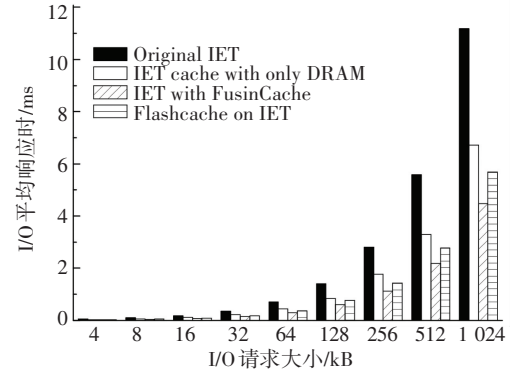
配置	iSCSI initiator 端	iSCSI target 端
CPU	Intel Xeon E5-2620	Intel Xeon E5620
内存	8G	16G
SSD	-	PCIe256G
磁盘阵列	-	RAID0:6LUN
操作系统	Ubunbu-12.04	CentOS-5.11
软件版本	openiscsi-2.0	iscsitarget-1.4.20

客户端上 IOMeter 测试结果见图 6. 图 6(a)展示几种机制在不同 I/O 请求块大小的吞吐性能，原生的 IET 默认块大小 4K，所以在请求大小为 4K 时性能最高，只使用 DRAM 做缓存和使用 FusionCache 的 IET 默认块大小都是 8K，所以请求大小在 8K 左右性能最高，继续增大块大小不会提高性能. FusionCache 相对原生 IET 和只使用 DRAM 的缓存提升分别为 54% 和 25% 左右，比 Flashcache 提高 18% 左右. 由于 SSD 空间远大于 DRAM，所以 Flashcache 的平均性能要稍好于 FusionCache 架构中只使用 DRAM 的情况.

图 6(b)显示，随着请求块大小成倍增大，I/O 平均响应时间整体呈指数上升. 集成缓存模块的 IET，允许 I/O 请求在缓存中获得请求数据而无须到磁盘请求，所以响应时间大大缩短；但是 DRAM 空间不大，缓存的数据量有限，而 SSD 缓存的空间增大几十倍甚至上百倍，进一步减少了请求磁盘的次数. FusionCache 相对原生 IET 和只使用 DRAM 的缓存带来的延时减少分别为 60% 和 33% 左右，比 Flashcache 减少 27%左右.



(a)吞吐量



(b)延时

图 6　IOMeter 读取性能测试

Fig.6　IOMeter read performance test

此外，以 4 K，8 K，16 K 请求大小对存储系统连续写入 1 小时（顺序），并统计对 SSD 的写入次数，以验证对 SSD 寿命的影响. 结果见图 7，front cache 减少了 30% 左右的 SSD 写入次数，因此可以提高 SSD 使用寿命，能够降低成本.

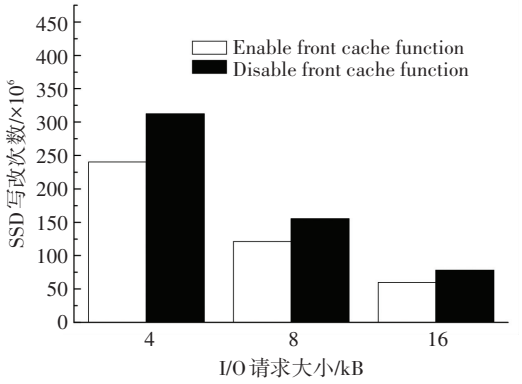


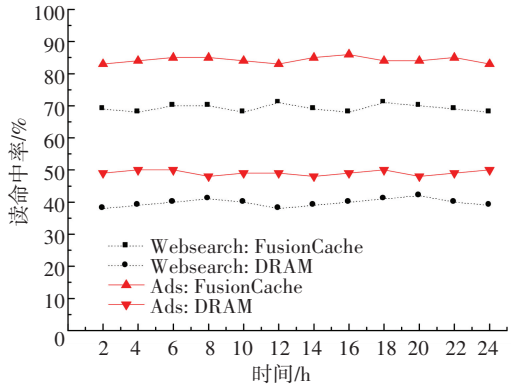
图 7　SSD 写入次数

Fig.7　SSD write times

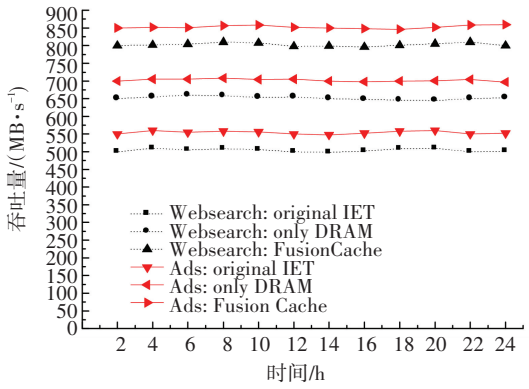
为了测试稳定状态下 Websearch 和 Ads 两种 traces 负载下的缓存命中率和 I/O 吞吐，先使两个 traces 各自运行一小时再记录数据，以跳过缓存的

热身(warm up)阶段,结果见图 8. 图 8(a)显示,在两种 trace 下,FusionCache 的读缓存命中率比只使用 DRAM 的分别提高 30%和 35%左右.

图 8(b)显示,FusionCache 的 I/O 吞吐在 Websearch trace 下,比原生 IET 和只用 DRAM 的方法分别提高 60% 和 23% 左右;在 Ads trace 下,FusionCache 比另外两种方法分别提高 54%和 21%左右.



(a)缓存命中率



(b)I/O 吞吐

图 8 两种 trace 性能测试

Fig.8 Two trace performance tests

4 结 论

本文针对 IET 控制器软件缺少独立缓存模块的问题,提出一种采用闪存的融合缓存机制 FusionCache. FusionCache 利用闪存和内存(DRAM)组成统一的融合缓存架构,闪存充当 DRAM 的扩展空间. 基于基数树(radix tree)管理缓存块元数据,用于加速缓存块的查找;基于回归拟合统计并预测缓存块访问热度,只允许热点数据进入闪存. 闪存采用改进的 LRU 算法对缓存块进行替换,并且在写回过程中考虑 iSCSI 会话状态. 实验结果表明,与其他方法相比,无论是采用 IOMeter 还是实际应用场景负载测试,其性能都要更好.

参考文献

[1] SATRAN J, METH K, SAPUNTZAKIS C, et al. Internet Small Computer Systems Interface (iSCSI) [EB/OL]. (2004-04-27). https://www.ietf.org/rfc/rfc3720.txt.

[2] HENSBERGEN E V, ZHAO Ming. Dynamic policy disk caching for storage networking: IBM Research Report RC24123 [R]. USA: IBM, 2006.

[3] 尹洋, 刘振军, 许鲁. 一种基于磁盘介质的网络存储系统缓存[J]. 软件学报, 2009, 20(10): 2752-2765.
YIN Yang, LIU Zhenjun, XU Lu. Cache system based on disk media for network storage[J]. Chinese Journal of Software, 2009, 20(10): 2752-2765.

[4] BYAN S, LENTINI J, MADAN A, et al. Mercury: Host-side flash caching for the data center [C]// IEEE Symposium on Mass Storage Systems and Technologies. Pacific Grove, CA: IEEE Publisher, 2012: 1-12.

[5] KOLLER R, MARMOL L, RANGASWAMI R, et al. Write policies for host-side flash caches [C]// USENIX Conference on File and Storage Technologies. San Jose, CA: USENIX Publisher, 2013: 45-58.

[6] ARTEAGA D, ZHAO Ming. Client-side flash caching for cloud systems [C]// Proceedings of International Conference on Systems and Storage. Haifa, Israel: ACM Publisher, 2014: 1-11.

[7] KOLLER R, MASHTIZADEH A J, RANGASWAMI R. Centaur: Host-side SSD caching for storage performance control[C]// IEEE International Conference on Autonomic Computing. Grenoble: IEEE Publisher, 2015: 51-60.

[8] WANG Jun, YAO Xiaoyu, MITCHELL C, et al. A new hierarchical data cache architecture for iSCSI storage server[J]. IEEE Transactions on Computers, 2009, 58(4): 433-447.

[9] REN Y, LI T, YU D, et al. Design, implementation, and evaluation of a NUMA-aware cache for iSCSI storage servers[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(2): 413-422.

[10] SUEI P L, YEH M Y, KUO T W. Endurance-aware flash-cache management for storage servers[J]. IEEE Transactions on Computers, 2014, 63(10): 2416-2430.

[11] HUO Zhisheng, XIAO Limin, ZHONG Qiaoling, et al. A metadata cooperative caching architecture based on SSD and DRAM for file systems[C]//International Conference on Algorithms and Architectures for Parallel Processing. Zhangjiajie: Springer Publisher, 2015: 31-51.

[12] LIU Yi, GE Xiongzi, DU D H, et al. SSD as a cloud cache? carefully design about it[J]. Taiwan Journal of Computers, 2016, 27(1): 26-37.

[13] SHANG Qiuli, ZHANG Wu, GUO Xiuyan, et al. An energy-saving scheduling scheme for streaming media storage systems[J]. High Technology Letters, 2015, 21(3): 347-357.

[14] MCNUTT B, BATES K. Umass trace repository: search engine I/O [EB/OL]. (2007-06-01). http://traces.cs.umass.edu/index.php/Storage/Storage.

[15] KAVALANEKAR S, WORTHINGTON B, ZHANG Qi, et al. Characterization of storage workload traces from production Windows servers[C]//IEEE International Symposium on Workload Characterization. Seattle: IEEE Publisher, 2008: 119-128.

(编辑 苗秀芝)