一、背景

传统方法：

1、缓存：

优先从性能层取数据，如果没找到，再去容量层取数据，在容量层找到数据后，更新该数据到性能层，同时访问该数据。策略：LRU、LFU等

2、分层：

性能层放热点数据、容量层放冷数据，按照业务访问的冷热轨迹去决定存放策略。因为性能层与容量层进行大体量的交换迁移需要很长时间，所以并不实时交换数据。

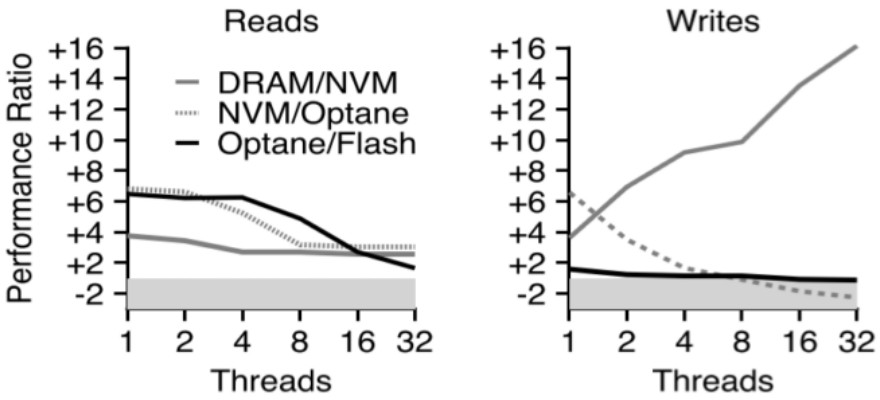
本文方法：

3、非分层缓存—NHC：

因为SSD与flash在高并发下的性能表现情况，所以考虑把性能层的多余负载下放到容量层，使得在进程数很多的情况下，性能层仍然能保持高性能。

出发点：

Example	Latency	Read (GB/s)	Write (GB/s)	Cost ($/GB)
DRAM	80ns	15	15	~7
NVDIMM	300ns	6.8	2.3	~5
Low-latency SSD	10us	2.5	2.3	1
NVMe Flash SSD	80us	~3.0	~2.0	0.3
SATA Flash SSD	180us	0.5	0.5	0.15



各种存储设备延迟有差异但带宽差异不明显，图2为在不同并发级别下对于4KB读取和写入的性能测试结果

DRAM/NVM 16GDDR4与128G傲腾DCPM性能比

NVM/Optane 傲腾DCPM与905P固态硬盘的性能比

Optane/Flash 905P固态硬盘与三星970Pro固态硬盘性能比

在并发数较大的情况下， 各种存储设备性能表现并不存在层次结构。而是与工作类型（读取还是写入）、并发级别高度相关。

二、缓存性能评价模型

分别对并发级别低和高两个极端情况进行建模。

1、对于一个请求（低并发）

平均时间：

$$T_{cache,1} = H \cdot T_{hit} + (1 - H) \cdot T_{miss}$$

T_{hit} 从性能设备读取的时间消耗

H 命中率

T_{miss} 容量层转移到性能层，并且从性能层读取的时间消耗

T_{hit} is simply the inverse of the rate of the fast device, i.e., $T_{hit} = \frac{1}{R_{hi}}$; T_{miss} is the cost of fetching the data from the slow device and also installing it in the faster device, i.e., $T_{miss} = \frac{1}{R_{hi}} + \frac{1}{R_{lo}}$, or $\frac{R_{hi}+R_{lo}}{R_{hi} \cdot R_{lo}}$.

带宽 B 为 T 的倒数

$$B_{cache,1} = \frac{R_{hi} \cdot R_{lo}}{H \cdot R_{lo} + (1 - H) \cdot (R_{hi} + R_{lo})}$$

2、对于N个请求（高并发）

在慢速/高速设备上的开销：（只有miss在慢速设备有开销，所有请求都在高速设备有开销）

$$T_{slow}(N) = N \cdot (1 - H) \cdot \frac{1}{R_{lo}} \quad (3)$$

$$T_{fast}(N) = N \cdot (1 - H) \cdot \frac{1}{R_{hi}} + N \cdot H \cdot \frac{1}{R_{hi}} = N \cdot \frac{1}{R_{hi}} \quad (4)$$

这样总时间由慢速设备和高速设备的最大开销决定（此处计算单一请求平均带宽）

$$T_{cache,many}(N) = max(T_{slow}(N), T_{fast}(N)) \quad (5)$$

$$= max(N \cdot \tfrac{1-H}{R_{lo}}, N \cdot \tfrac{1}{R_{hi}}) \quad (6)$$

Dividing by N (not shown) yields the average time per request. Finally, the bandwidth can be computed, as it is the inverse of the average time per request:

$$B_{cache,many} = \frac{1}{max(\frac{1-H}{R_{lo}}, \frac{1}{R_{hi}})} \quad (7)$$

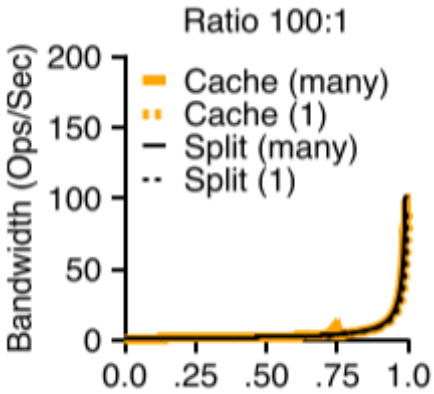
3、分级缓存机制下的带宽

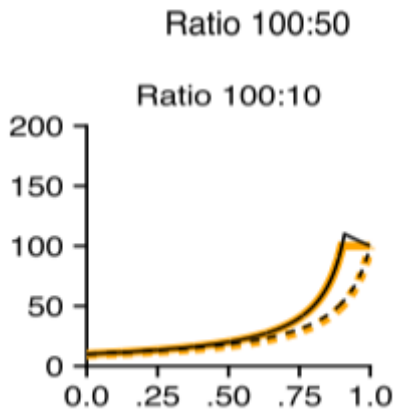
s为冷热数据分离比，s为存储在热性能层的比例

$$B_{split,1} = \frac{1}{\frac{1-S}{R_{lo}} + \frac{S}{R_{hi}}}$$

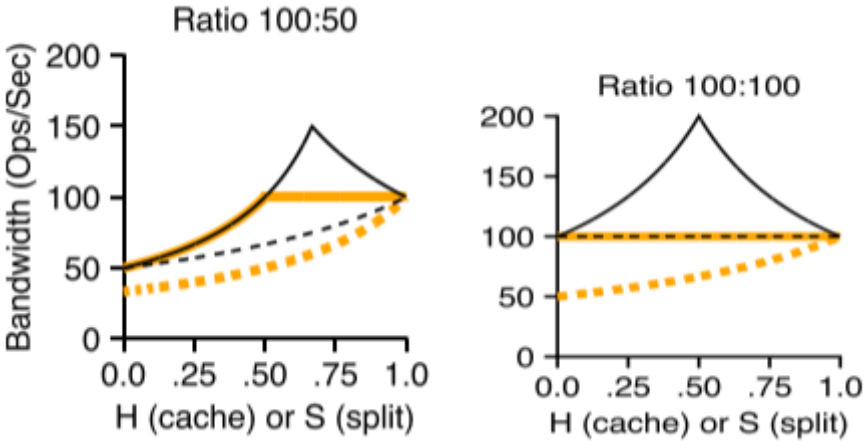
$$B_{split,many} = \frac{1}{\max\left(\frac{1-S}{R_{lo}}, \frac{S}{R_{hi}}\right)}$$

4、模型效果





在传统的缓存和分离的两种缓存策略下，只有几乎全部在性能层命中时才能表现良好，80%时都仍然很慢（高速设备与低速设备差距很大，差距缩小后，命中率较低时也能体现出提升了）。



在现代层次结构中，设备之间性能差距没有那么大，单独提升性能层的请求比例时，并不总会产生最好性能。

三、实验准备

1、HFIO

用于实现缓存和拆分，缓存使用LRU策略，可以实现各种参数的控制工作，可以调整缓存的大小，通过访问数据的地址偏移去调整缓存的命中率，每个cache block大小为32KB，实验在随机访问的情况下进行。

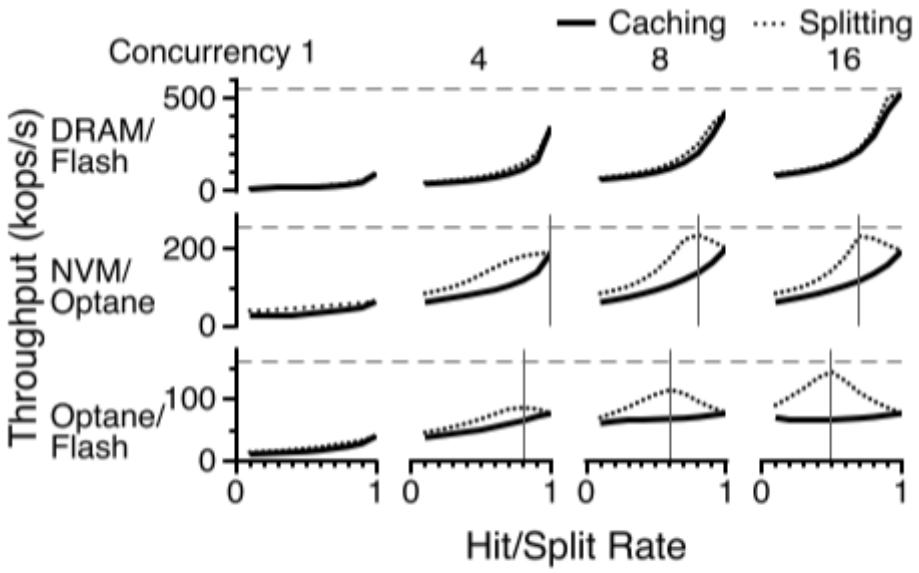


Figure 4: **Performance of Caching and Splitting.** *This figure shows the throughput of read-only workloads. Horizontal dotted lines represent the combined bandwidth of both devices (the maximum possible throughput).*

DRAM/Flash 适用于过去的传统存储设备结构，随着命中率的提高，吞吐量得到了大量的提升。

NVM/Optane 和 Optane/Flash 符合现代设备的存储结构，可以看到采用分层处理的情况下，在并发数提高时候，并没有在最高命中率时得到最佳性能，符合我们的预期。

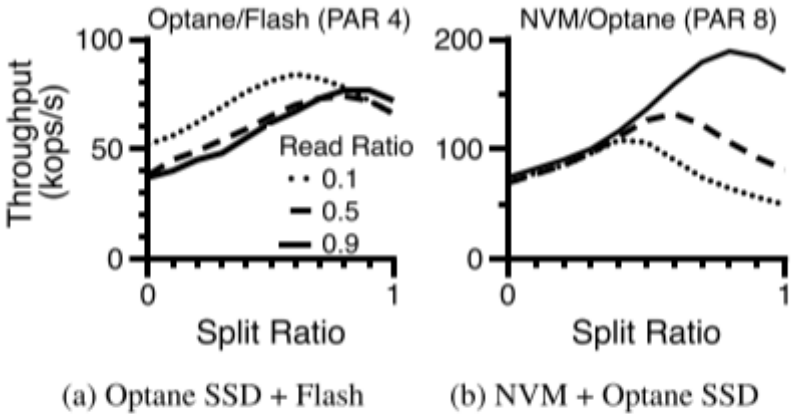


Figure 5: **Mixed Reads and Writes Workloads.** *The figure shows the performance of splitting with read-write workloads; PAR: workload parallelism/concurrency.*

在不同读写操作比例下，最佳性能在不同分离比例时刻出现。

2、启发

在现代存储结构当中，性能层与容量层的速度差距并没有那么大。但性能层往往承受了很大的带宽压力，这时候可以考虑将一部分热数据转移到容量层，从而利用容量层的性能。

四、NHC缓存框架

1、目标

1）性能与经典缓存一样或更好。在性能层没有超出负载时候，将数据都存放在性能层，而当超出负载时，将多余负载放到容量层处理。

2）不要求事先了解设备工作内容的详细信息或性能特征（不事先知道访存轨迹），能够适应任何层次结构。

3）NHC缓存框架可以在负载动态变化中做出适应性的决策（在过载时候下放任务）。

在启动时候，NHC与传统的缓存框架相同。在命中率稳定之后，NHC框架减少进入性能层的数据量，这个时候在从容量层提取数据时，并不将其更新到性能层。如果命中率一直不稳定，那么NHC将一直以传统缓存框架的方式运行。NHC使用与传统缓存框架相同的写分配策略（写回/回写）。

2、NHC结构

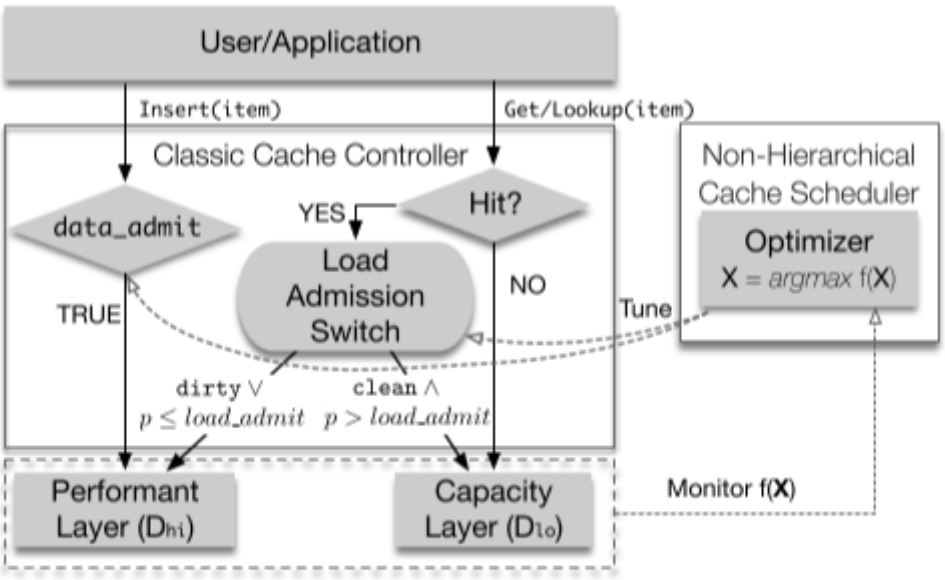


Figure 6: **Non-Hierarchical Caching Architecture.** *This figure shows the architecture of NHC. NHC adds decision points and a scheduler to classic caching. As before, NHC is transparent to users. Any classic caching implementation can be upgraded to be a NHC one. Note that decision points only tune cache read hits/misses.*

经典缓存，使用LRU等策略控制性能层的数据内容。

NHC框架中，设立一个BOOL量 data_admit(记为da) 和 一个变量 load_admit (记为la)。

da控制读取数据时，当**读取miss**时，是否将数据复制送入性能层当中。如果da == true，那么送入性能层，否则不送入性能层。经典的缓存框架，da始终为true。

The la variable controls how **read hits** are handled and designates the percentage of read hits that should be sent to D_{hi}; when la is 0, all read hits are sent to D_{lo}. Specifically, for each read hit, a random number $R \in [0, 1.0]$ is generated; if $R <= \texttt{la}$, the request is sent to D_{hi}; else, it is sent to D_{lo}. In classic caching, la is always 1.

la控制读取数据时，当读取hit时。我们使用一个随机数R去控制，我们是从性能层中读取数据，还是从容量层中读取数据。（所以我们在状态2中要不断调整la，同样也可以通过la去控制性能层的负载）经典缓存框架，la始终为1。

da和la不控制写hit/写miss，对于写命中和写miss的处理与传统方法一样。

3、缓存调度算法

NHC调度控制器有两种状态，

状态1：命中率未稳定，性能层还没有超过负载，这个时候发送read miss时候，将数据复制到性能层。

状态2：命中率稳定，保持缓存（性能层）数据不变，同时调整发送到每个设备的请求。

Algorithm 1: Non-hierarchical caching scheduler

cache: classic cache controller
step: the adjustment step size for *load_admit*
f(x): function that measures target performance metric when *load_admit* = *x*, the value is measured by setting *load_admit* = *x* for a time interval

```
1  while true do
       # State 1: Improve hit rate
2      data_admit = true,  load_admit = 1.0
3      while cache.hit_rate is not stable do
4          sleep_a_while()
5      data_admit = false,  start_hit_rate = cache.hit_rate
       # State 2: Adjust load_admit
6      while true do
7          ratio = load_admit
           # Measure f(ratio-step) and f(ratio+step)
8          max_f = Max(f(ratio-step), f(ratio), f(ratio+step))
           # Modify load_admit based on the slope
9          if f(ratio-step) == max_f then
10             load_admit = ratio - step
11         else if f(ratio+step) == max_f then
12             load_admit = ratio + step
13         else if f(ratio) == max_f then
14             load_admit = ratio
15             if load_admit == 1.0 then
16                 goto line 2 # Quit tuning if w < w0
           # Check whether workload locality changes
17         if cache.hit_rate < (1-α)start_hit_rate then
18             goto line 2
```

状态1：

先设置 da = true，采用经典缓存策略，la设置为1。

当cache的命中率没有稳定的时候，一直不进入状态2。

当命中率稳定的时候，设置da = false，初始 la = 1.0

状态2：

每次以固定步长去调整la，使la往更好性能的方向调整。当la已经是最好性能时候，需判断此时la是否为1.0，若为1.0那么需要调整重新进入状态1。

同样，若调整la之后，使得命中率不再稳定，相比之前发生了下降，也要重新进入状态1。

调整步长之后的性能，是在一段时间内进行测量获得。

五、具体实现

https://github.com/josehu07/nhc-demo

1、修改Open CAS

修改了英特尔构建的缓存软件。Open CAS支持回写直写等写分配策略，使用LRU策略进行性能层更换。修改后的OrthusCAS支持Open CAS的所有策略。

https://open-cas.github.io/

https://github.com/Open-CAS/open-cas.github.io

2、在Wisckey(修改自leveldb)的持久缓存中实现NHC

leveldb

https://github.com/google/leveldb

https://github.com/coyorkdow/wisckey

https://zhuanlan.zhihu.com/p/80684560

https://github.com/messixukejia/leveldb

https://www.cnblogs.com/chenhao-zsh/p/11616838.html

在100毫秒内，命中率变化在0.1%之内认定为命中率稳定。

3、目标性能指标

使用Linux block layer 统计吞吐量，间隔小于5ms，统计不准确，实验中，论文使用2%的负载率调整步长，得到了较好的结果。

六、Orthus-KV

基于wisckey中实现，wisckey修改自levelDB。

在Orthus-KV中，使用傲腾SSD和Flash去实验。性能层大小为33GB，总体数据集为100GB。页面缓存限制为1GB。性能测试工具使用YCSB（https://blog.csdn.net/dc_726/article/details/43991871）。测量了在32个线程下的吞吐量。