DAC: A Device-Aware Cache Management Algorithm for Heterogeneous Mobile Storage Systems

Young-Jin Kim,
Department of Computer Science and Engineering,
Sun Moon University,
Korea

Jihong Kim,
School of Computer Science and Engineering,
Seoul National University,
Korea

2008 The Institute of Electronics, Information and Communication Engineers
Outline

• Introduction
• Motivational Example
• Proposed Scheme
• Simulation
• Conclusion
Introduction (1/3)

- Heterogeneous Storage Devices
  - Hard Disk
    - low cost per bit
    - significant power consumptions
    - poor performance for random I/O requests
  - NAND flash memory
    - fast response times
    - low power consumptions
    - cost per bit are much higher
    - limited write/erase cycles

<table>
<thead>
<tr>
<th>Device</th>
<th>Hard disk</th>
<th>NAND flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (512B)</td>
<td>Read</td>
<td>Write</td>
</tr>
<tr>
<td></td>
<td>2.5”</td>
<td>1.8”</td>
</tr>
<tr>
<td></td>
<td>19.1 (ms)</td>
<td>22.1 (ms)</td>
</tr>
<tr>
<td></td>
<td>25 (us)</td>
<td>200 (us)</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>800 (us)</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>1.5 (ms)</td>
</tr>
<tr>
<td></td>
<td>1.5 (ms)</td>
<td></td>
</tr>
<tr>
<td>Power (mW)</td>
<td>Active</td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td>2300</td>
<td>950</td>
</tr>
<tr>
<td></td>
<td>1400</td>
<td>400</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>&gt; 33</td>
<td>&gt; 0.13</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Cost per GB ($)</td>
<td>0.82</td>
<td>1.82</td>
</tr>
<tr>
<td></td>
<td>10.89</td>
<td>3.34</td>
</tr>
</tbody>
</table>
Introduction (2/3)

• work (or load) balance
  – work is evenly spread over the available devices
  – depends on I/O request distribution over devices
    • I/O request distribution depends on buffer cache management directly (including management policy and filtering effect)

• in heterogeneous storage systems, work balance is fragile due to a large gap between access times of different devices
  – device-aware (or cost-aware) buffer caching algorithms need to take accounts of the fluctuates of access times between the different devices as well as the workload patterns (temporal locality)
• work balance is not always achievable in heterogeneous
  – Ex. highly skewed sequential I/O accesses over a slower device can derive workload imbalance
• thus, aim at optimizing the overall I/O performance rather than achieving work balance

• How to achieve “device-awareness”?
  – to deal with blocks with heterogeneous access times within a single partition cache
  – to partition a buffer cache into separate partitions (one per device) and managing them according to the assigned value to each block
    • static partitioning
    • dynamic partitioning
Motivational Example (1/2)

Fig. 1 Total I/O cost in a heterogeneous storage system with a 30 MB cache for a real mobile workload from [25]. As the cache partition size for a disk grows, the total I/O cost decreases largely but has the smallest at a cache partition configuration of 27 MB for a disk and 3 MB for a flash memory, outperforming LRU with a single cache partition by about 63%.
Motivational Example (2/2)

Fig. 2  Comparisons of the optimal static cache partition sizes in terms of the total I/O cost for three different synthetic mobile workloads in a storage system with a disk and a NAND flash memory. We notice that different workload patterns derive different best static cache partition configurations, that is, static cache partitioning policies.
Proposed Scheme (1/8)

- n different devices $D_i$’s ($D_1$ is the slowest device, $D_N$ is the fastest)
- a buffer cache $B$ is divided into $n$ partitions $P_i$’s

\[ B = \bigcup_{i=1}^{n} P_i \text{ and } P_i \cap P_j = \phi \text{ and } P_i \cap P_j = \phi, \text{ for } i \neq j \]

- $R_i = (\text{access time of } D_i) / (\text{access time of } D_n)$
  - $R_1 \geq R_2 \geq \ldots \geq R_N$
- $c(P_i) = R_i \cdot m(P_i)$

Find $P$ that minimizes the sum of the I/O costs

\[ c_{tot}(P) = R_1 \cdot m(P_1) + R_2 \cdot m(P_2) \]

subject to $B = P_1 \bigcup P_2$ and $P_1 \bigcap P_2 = \phi$

, where $D_1$ = a disk and $D_2$ = a NAND flash memory.
Proposed Scheme (2/8)

- **Static** cache partitioning solution loses many chances of obtaining more optimal cache partition configurations in terms of the total I/O cost.
- Variable access patterns of requests across heterogeneous devices produce **dynamically varying functions of cache miss counts** to partition sizes, thus obtain the optimal solution during run time is almost impossible.

![Graph](image)

Find $P$ that minimizes the sum of the I/O costs

$$c_{tot}(P) = R_1 \cdot m(P_1) + R_2 \cdot m(P_2)$$

subject to $B = P_1 \bigcup P_2$ and $P_1 \bigcap P_2 = \phi$

where $D_1 = $ a disk and $D_2 = $ a NAND flash memory.

**Fig. 3** Relation between the weighted cache miss counts and cache partition sizes for typical mobile workloads. Since they have mixed I/O access patterns of loop and random accesses, the cache miss counts are likely to be a mix of convex and non-convex functions of the cache partition sizes.
Problem:
1. may obtain a cache partition solution in proportion to the probability in the worst case, such randomness is not desirable
2. cannot know how much the current total I/O cost is close to the smallest value
Proposed Scheme (3/8)

• under the inspiration of the simulated annealing method

1. Monitors $\Delta m(P_1), \Delta m(P_2), \Delta c_{tot}$, and workload patterns periodically.
2. Adapts a weight for increment or decrement of each partition size according to the observed workload patterns.
3. Adjusts the partition sizes ($s(P_1)$ and $s(P_2)$).
4. If $\Delta c_{tot} < 0$
5. If $\Delta m(P_1) < 0$ and $\Delta m(P_2) > 0$
6. Increase $s(P_2)$ with a fine weight.
7. Else If $\Delta m(P_1) < 0$ and $\Delta m(P_2) < 0$
8. Increase $s(P_1)$ with a fine weight.
9. Else
10. Increase $s(P_1)$ or $s(P_2)$ with a fine weight based on random miss counts of $D_1$ and $D_2$.
11. Else
12. Increase $s(P_1)$ or $s(P_2)$ with a fine weight based on random miss counts of $D_1$ and $D_2$.
13. If $s(P_1) > |B|$, $s(P_1) = |B|
14. Else if $s(P_1) < 0$, $s(P_1) = 0$
15. Else do nothing
16. $s(P_2) = |B| - s(P_1)$

Dynamic partitioning heuristic

• (3~10) fine-grained adjustment
  • to have the total I/O still decrease or not increase
• (11~12) coarse-grained adjustment
  • to have the total I/O cost decrease
• (13~16) bounding the total cache size

to minimize the total I/O cost by adjusting $s(P_1)$ and $s(P_2)$

Fig. 4  Pseudo codes of the proposed dynamic partitioning heuristic.
Proposed Scheme (4/8)

• DAC minimize the total I/O cost by
  
  – **device-aware cache management** (higher-level)
    • adjusting the cache partition using the dynamic partitioning heuristic

  – **workload-aware management** (lower-level)
    • obtaining performance enhancement by having **sequential** blocks for a **disk** and **read** blocks for a **flash memory** to be evicted earlier **within each partition**
VARIABLES AND STRUCTURES
x: referenced block, |B|: total cache size
W: partition-adjusting period
T1, T2: partitions for a disk and a flash memory
cumulated_ref_count: cumulated reference count

MAIN ALGORITHM
1. Initialize required variables and structures
2. Determine the access pattern of each block if it is sequential or random at every reference of blocks
3. Increase cumulated_ref_count
4. If a cache hit occurs in T1 (or T2)
5. Move the block to the MRU in T1 (or T2)
6. Execute H_update
7. Else
8. Execute List_update
9. If cumulated_ref_count = W
10. Execute the dynamic partitioning algorithm in Fig. 4

SUBROUTINE List_update
1. If a miss occurs in T_j ( j = 1 or 2)
2. If |T_j| > T_j - req and |T_j| ≠ 0
3. T_j.L ← min{y.H | y ∈ T_j}
4. Evict y which satisfies y.H = T_j.L
5. Decrease |T_j|
6. Fetch the block to the cache and move it to the MRU in T_j
7. Execute H_update
8. Increase |T_j|
9. If |T_1| + |T_2| ≥ |B| and |T_j| > T_j - req (j = 1 or 2)
10. T_j.L ← min{y.H | y ∈ T_j}
11. Evict y which satisfies y.H = T_j.L
12. Decrease |T_j|
13. Update random miss count and total miss count in T_j

SUBROUTINE H_update
1. If a miss occurs in T1
2. If block x’s pattern is sequential
3. x.H ← T1.L + C_SEQ
4. Else
5. x.H ← T1.L + C_RAND
6. Else
7. If x’s I/O type is read
8. If x’s pattern is sequential
9. x.H ← T2.L + C_SEQ * C_READ
10. Else
11. x.H ← T2.L + C_RAND * C_READ
12. Else
13. If x’s pattern is sequential
14. x.H ← T2.L + C_SEQ * C_WRITE
15. Else
16. x.H ← T2.L + C_RAND * C_WRITE

Fig. 5 Proposed device-aware cache management algorithm (DAC).

Fig. 4 Pseudo codes of the proposed dynamic partitioning heuristic.
Proposed Scheme (6/8)

- determine the access pattern of each block by **sequentiality threshold**
  - cumulated_ref_count++

- if the partition is full, then **Evict** the block with **minimum H**
  - fetch the accessed block to MRU
  - H_update
  - bounding the cache size
  - track its miss count

- use dynamic partitioning algorithm to compute the **target size** of T1 and T2

- Hit
  - Move to MRU
  - H_update

- Miss
  - adjust each partition size toward its target size in the period

- every \( W \) references
Proposed Scheme (7/8)

when eviction, the block with the minimum H value would be selected as a victim

SUBROUTINE H_update

1. If a miss occurs in T1
2. If block x’s pattern is sequential
   3. \( x.H \leftarrow T1.L + C_{SEQ} \)
4. Else
   5. \( x.H \leftarrow T1.L + C_{RAND} \)
6. Else
7. If x’s I/O type is read
   8. If x’s pattern is sequential
      9. \( x.H \leftarrow T2.L + C_{SEQ} \ast C_{READ} \)
   10. Else
      11. \( x.H \leftarrow T2.L + C_{RAND} \ast C_{READ} \)
   12. Else
    13. If x’s pattern is sequential
    14. \( x.H \leftarrow T2.L + C_{SEQ} \ast C_{WRITE} \)
15. Else
16. \( x.H \leftarrow T2.L + C_{RAND} \ast C_{WRITE} \)

• \( C_{SEQ} < C_{RAND} \)
• \( C_{READ} < C_{WRITE} \)
• assume that $|B|=10$, $|T_1|=5$, $|T_2|=6$, $T_2_{\text{req}}=5$

| T2 | 6 |
|-----|
| B6 |
| B5 |
| B4 |
| B2 |
| B1 |
| B3 |
| H  |
| 3+4|

• now access B7, miss
  – Evict B4 ($B4.H$ is the minimum H), $T2.L = 3$

| T2 | 5 |
|-----|
| B6 |
| B5 |
| B2 |
| B1 |
| B3 |
| H  |
| 5  |

• fetch B7 to MRU and update it’s H

| T2 | 6 |
|-----|
| B6 |
| B7 |
| B5 |
| B2 |
| B1 |
| B3 |
| H  |
| 7  |

• check cache size bounding, if $|T_1|+|T_2| \geq |B|$ AND $|T_2|>T_2_{\text{req}}$.
  – if so, select a victim to decrease $|T_2|$, Evict B1

| T2 | 5 |
|-----|
| B6 |
| B7 |
| B5 |
| B2 |
| B3 |
| H  |
| 7  |
• built a trace-based cache simulator, which simulates LRU, Greedy-Dual, and DAC, and concatenated it with a multi-device I/O simulator, which can simulate file allocation and I/O over a heterogeneous system

• process real file I/O traces from an evaluation board similar to a PDA, including a 400Mhz PXA 255 processor, an 10 Mbps ethernet card, and a 64MB SDRAM
Simulation (2/13)

- real trace
  - PDA & PMP

- synthetic trace

<table>
<thead>
<tr>
<th></th>
<th>average interval time</th>
<th>maximum file size</th>
<th>total file size</th>
<th>write ratio</th>
<th>I/O access pattern</th>
<th>trace time</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace 1</td>
<td>70 (ms)</td>
<td>5 (MB)</td>
<td>350 (MB)</td>
<td>0.5</td>
<td>COMPOUND</td>
<td>80 (min)</td>
</tr>
<tr>
<td>trace 2</td>
<td>20 (ms)</td>
<td>1 (MB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PDA</th>
<th>PMP</th>
<th>trace 1</th>
<th>trace 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>working set size (MB)</td>
<td>44</td>
<td>51</td>
<td>23</td>
<td>57</td>
</tr>
<tr>
<td>trace file size (MB)</td>
<td>31</td>
<td>38</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

- The overheads of re-partitioning and handling blocks per partition in DAC over LRU are able to ignore.
  - LRU needs $O(n)$ steps to check hit or not, DAC needs more $O(\log(n))$ steps when it searches eviction.

- set $W=200$, $R1=35$, $R2=1$ for DAC
Simulation (3/13)

• DAC improves performance by up to 64% and 14% over LRU and GD
• PDA trace has a large number of loop access and a considerable number of random ones, and accessed uniformly across a disk and a flash
• thus, GD has a good performance

![chart](chart.png)

**Fig. 6** For the PDA trace, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.
Simulation (4/13)

- DAS achieves I/O cost improvement by up to **77.8%** over LRU
- Because PMP trace has a larger number of and longer loop accesses than the PDA trace, GD shows almost similar performance enhancement to DAC.

![Graphs](image)

**Fig. 7** For the PMP trace, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.
Simulation (5/13)

- trace1 has a lot of random I/O requests and doesn’t include loop patterns
- LRU shows comparable total I/O costs with GD and DAC

Fig. 8  For the trace1, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.
Simulation (6/13)

- trace2 has more block accesses with temporal localities than the trace1 although the degree of randomness may be similar.
- DAC improves the total I/O cost by up to **21%** and **11%** over LRU and GD, respectively.

**Fig. 9** For the trace2, (a) total I/O costs of LRU, GD, and DAC (b) total I/O costs of GD and DAC normalized over LRU.
Simulation (7/13)

- plot(a), DAC lowers the number of flash writes by up to 30% and 23% over LRU and GD, respectively.
- plot(b), DAC lowers the number of flash writes by up to 57% and 28% over LRU and GD, respectively.

Fig. 10 (a) For the PDA trace, flash write counts of GD and DAC normalized over LRU (b) For the PMP trace, flash write counts of GD and DAC normalized over LRU.
Simulation (8/13)

- the reduction of flash write count is shown less effective
- the authors guess that random accesses of these two traces and another randomness used to assign write I/O types to each access may disturb DAC’s attempt to evict write block later

Fig. 11  (a) For the trace 1, flash write counts of GD and DAC normalized over LRU  (b) For the trace 2, flash write counts of GD and DAC normalized over LRU.
Simulation (9/13)

- plot(a), DAC saves energy by up to 44% and 11% over LRU and GD, respectively
- plot(b), DAC saves 58% and 11% over LRU and GD, respectively
- good energy savings of DAC can be attributed to considerably reduced I/O accesses onto the hard disk

**Fig. 12**  (a) For the PDA trace, total energy consumptions of GD and DAC normalized over LRU (b) For the PMP trace, total energy consumptions of GD and DAC normalized over LRU.
Simulation (10/13)

- plot(a), trace1 has a small working set and its I/O accesses have temporal localities, the total standby time is shown similar regardless of the buffer cache algorithm.
- plot(b), DAC saves energy by up to 17% over LRU.

Fig. 13 (a) For the trace1, total energy consumptions of GD and DAC normalized over LRU (b) For the trace2, total energy consumptions of GD and DAC normalized over LRU.
Simulation (11/13)

- the total I/O costs with these best static partition configurations are larger than DAC
- DAC can adapt different workload patterns dynamically and achieves a more close-to-optimal performance than static cache partitioning techniques

![Graphs showing I/O costs comparison](image)

**Fig. 14** Total I/O costs of DAC and static partitioning, which uses a fixed static partitioning policy and the intra-partition management of DAC (a) for the PDA trace, (b) for the PMP trace.
Simulation (12/13)

- DAC still can be said to find close-to-optimal performance by dynamically adjusting the required partition size per each device.

Fig. 15 Total I/O costs of DAC and static partitioning, which uses a fixed static partitioning policy and the intra-partition management of DAC (a) for the trace1, (b) for the trace2.
Simulation (13/13)

- compares the total I/O costs of four techniques: (1) static partitioning + intra-partition management algorithm of DAC (2) static partitioning + LRU (3) DAC (4) dynamic partitioning + LRU

- plot(a), PDA trace has a large number of loop access and a smaller number of random ones and access pattern are varying continuously

- plot(b), trace1 has a large number of random accesses and less workload pattern fluctuations

Fig. 16 Total I/O costs of four techniques: the static partitioning policy + the intra-partition management algorithm of DAC, DAC, and the dynamic cache partitioning of DAC + LRU (a) for the PDA trace, (b) for the trace1.
Conclusion

• proposed a novel buffer cache management algorithm with a heterogeneous storage to **minimize the total I/O cost** under varying workload patterns

• trace-based simulations show that the proposed algorithm improves the total I/O cost

• future work
  – MEMS-based storage and a hard disk
  – data migration techniques
  – consideration of prefetching
  – implementation and verification in the real file systems