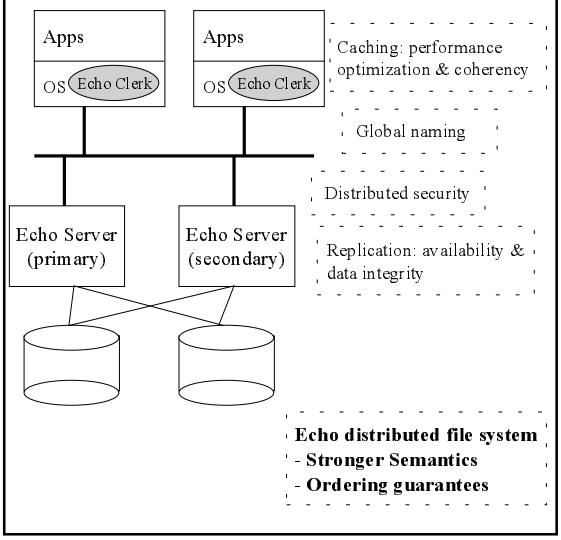
A Coherent Distributed File Cache with Directory Write-Behind

15-712

presented by
Akihiko Miyoshi
miyos@cs.cmu.edu

Echo: The Big Picture



Initial Goals

- Use of cache for performance
- Stronger semantics
  - single system view to apps
  - cache coherency
  - ordering guarantees

Caching in Echo

- Performance
  - aggressive use of cache (16MB of main memory dedicated to cache)
  - cache both file & directory
  - write-behind
- Comparison of 3 configuration
  - A) write-behind for both files & dir
  - B) write-behind for file, write-through for dir modification
  - C) write-through for dir, write-back on close for file

Caching in Echo(2)

- Cost (mean running time)
  - A < B < C
  - A) gives 4% to 42% benefits
- Observation
  - Directory write-behind gives improvement
    - depends strongly on the amount of work done between directory writes
    - probably more effective on slower/loaded servers (faster response time for clients)
  - Penalty for file write-back on close

Semantics

- single-copy equivalent/single system view
  - at any moment, the fs data is in a single, well-defined state
  - cache with strict coherence
  - replication with strict consistency
  - eg) when faults occur, updates cached (written behind) may be discarded (undone)
- clients see one consistent fs (not slightly inconsistent ones)
  - hence, require no change in existing apps

Ordering Constraints

- write-behinds can be discarded….
  - ordering becomes important
- specifies partial order on writes
- guarantees that actual write to disk will be consistent with this partial order
- give application stronger control

Fault-tolerance

- mask faults when possible & fail cleanly otherwise
  - replication of servers/disks
  - bookkeeping data replicated in main memory of two servers
  - clean failure semantics
    - guarantees ordering of write-behind or else returns error to apps

Echo's Ordering Guarantees (Informal Version)

- (A) If a write is requested by one client & the result is observed by another, the write is stable
- (B) Writes to a given object become stable in the same order they are logically performed. (overwrites may be reordered)
- (C) Writes are stable when they are forced to disk by "fsync"

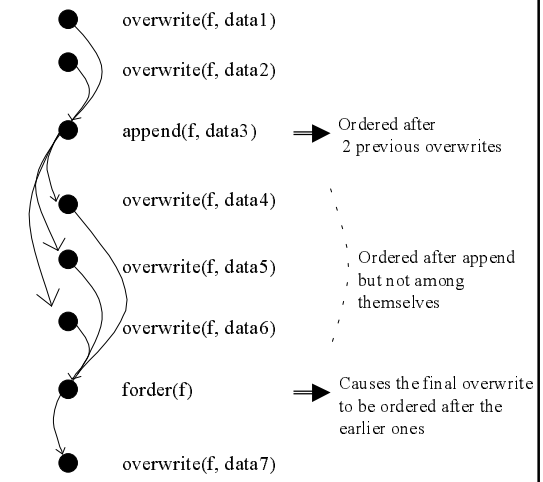
Echo's Ordering Guarantees (Formal Version)

- -> : data dependency (first operation could have affected the result of second)
- => : partial order in which writes are guaranteed to reach disk.

Example (1)

=> relation for writes to a single file

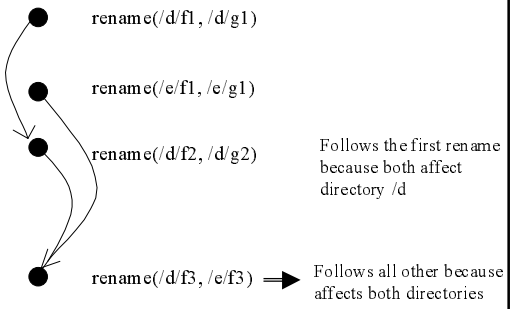
* arrows represent => relations



Example (2)

=> relation for renames

* arrows represent => relations

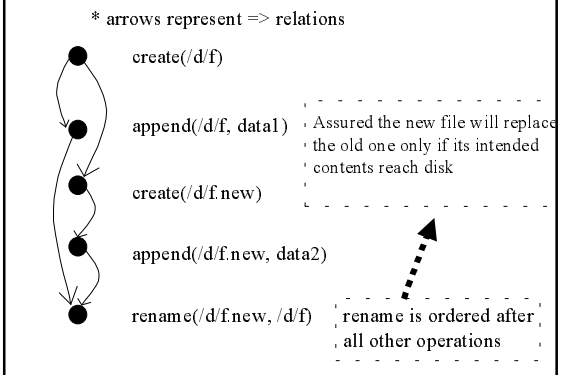


Using Echo's caching & ordering semantics

- Writing apps are easier
  - in NFS, "parallel make" may not find an object already compiled
    - does not have coherence on directory
  - in Echo, "make" can be guaranteed to find the objects
- Applications maybe more robust against failure
  - preventing corrupt files in crashes

Using Ordering Semantics (1)

Replacing a file with a new version



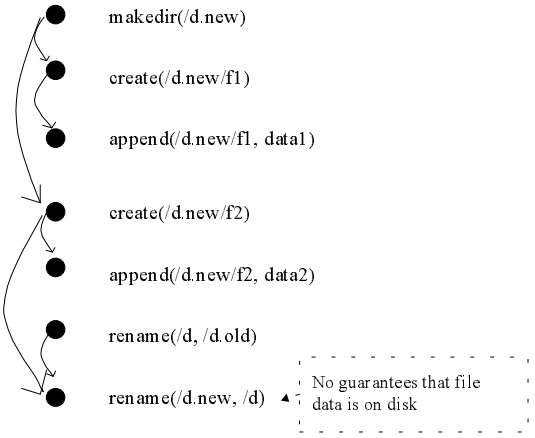
In case of UNIX (dir operations = write through, file data = write-behind), /d/f.new will be renamed before its contents reach disk.

If crashes before data reaches disk, /d/f will point to garbage!

Example (2)

Replacing a directory with a new version

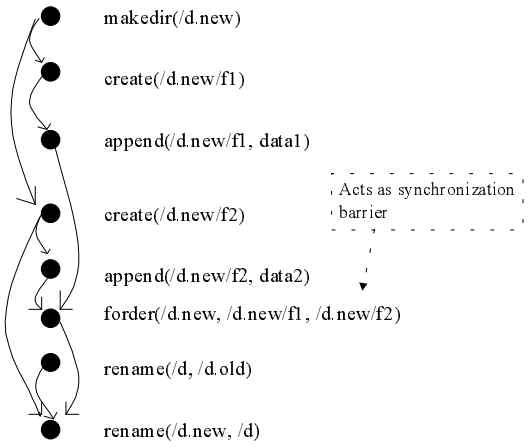
* arrows represent => relations



Example (2) fix

Replacing a directory with a new version

* arrows represent => relations



Implementing Cache coherency

- tokens
  - before an Echo clerk can hold data in cache, it must obtain token from server
    - read token
    - write token
  - token covers "a whole file or directory"
    - in some cases covering byte ranges of files may be better design
  - server will keep track of tokens
    - if any clerk holds write tokens, then no other clerk is allowed to hold read/write token on the same object
    - no write tokens given out
      - all cache in server/clerk are identical
    - write token given out
      - clerk with the token is the only one accessing the data

Token management

- Request for a token on a data
  - no conflict
    - server grants the token
  - conflict
    - server asks clerk to give it back
    - read token: clerk holding the token gives it back immediately
    - write token: clerk writes back immediately the data covered by token & then gives back token
    - in both cases, clerks discards cached copy of data

Token management (2)

- Leases
  - agreement between server & clerk that tokens remains valid for a given period of time
    - multiple tokens are associated with one lease
  - allows servers to reclaim tokens
    - clerk crashes
    - communication failure
  - ensures caches remain single-copy equivalent
    - if lease is in force when operation requested,
      - cached data is good
    - if lease has expired
      - if server has not leased the token to other clerks, reestablish the lease. (cache valid)
      - else, discard cached data & write-behind (lost write-behind!)

Lost Write-behinds

- Write-behinds may be lost
  - in crashes
  - communication fault (timeout)
- Reporting lost write-behinds
  - on access request by a process, clerk reports lost write-behind, if token has been revoked by the server
  - conservative
    - there may not have been any write-behind lost
    - even if write-behind was lost, it may not affect the client
- Handling lost write-behinds
  - no way of notifying applications that has exited already
  - notify processes that did not care about the loss
  - can not notify processes that depend indirectly
  - even when lost write-behind is reported successfully, not quite sure what the apps should do….

Sprite file system

[Nelson et al. 1988]

- uses tokens to maintain coherence
  - Sprite: must hold token to read/write
  - Echo: only need token when caching data
- clerks cache only files (not directories)
- if 2 clerks hold the same file open in conflicting mode, caching is turned off
  - good design for 1 writer, n reader
  - not so good for clerks accessing different parts of random-access file (would be better to have tokens covering part of file)
  - little difference in performance
- does not use "leases"
  - server will invalidate tokens immediately if it cannot contact clerk
    - advantage in response time (RPC timeout)
  - violates single-copy equivalence
    - only for files that were open, when token was revoked
    - practical tradeoff

Andrew File System

[Howard et al. 1988; Kazar 1988]

- directory updates are write-through
- caches both files & directories
  - caches files entirely (not block by block)
  - file writes are buffered until closed (meaning no single-copy equivalence)
- uses "callbacks" (similar to "tokens")
  - servers can revoke token when
    - leases time out
    - cannot contact clerk
      - » violates single copy equivalence until the token times out

Conclusion

- Succeeded
  - Better ordering constraints
    - succeeded in keeping the data in the file system consistent on failure
- Need more work
  - Had to sacrifice some changes in interface/semantics of single-machine file system
    - explicit use of forder
  - Effectiveness of the features?
    - Conservative reporting of lost write-behind
    - Handling lost write-behind by the application