Speculator
==========

What is close-to-open consistency?
What are single-copy semantics?

Some alternative implementation techniques:
  Callbacks (used by AFS)
  Leases

What is premise of the paper?
  1. Can correctly predict many operations
  2. Checkpoints are cheaper than network round trips
  3. Have unused CPU time

How does NFS3 write separate write and commit?
  Not all writes must go through to disk
    WRITE reply has stability + "verf" that changes on server reboots
    Client keep buffering data if server says not stable
  Eventually (on fsync, close) client calls COMMIT
    Server writes data through to disk and replies with same verf
    If verf changed (server rebooted), client resends buffered writes

What NFS operations can correctly be predicted?
  Use GETATTR or ACCESS on every file open to check access rights and cache
    Unless file modified on server or other client, cache will be okay
  Use GETATTR on dir to validate name cache entry on file name lookup
  Use GETATTR on stat syscall during, e.g., ls -l
  Wait for write to complete before commit (Figure 1)
    Would Figure 1 be better off with NFS version 2 (synchronous writes)?
    NFS3 WRITE RPC has synchronous option.  Why not use that?
      Might not know client is going to call close--could overwrite block
      But note client buffers data until complete data block or close
    Why not do COMMIT in background after file close (not fsync)?
      Server reboots: brief window violates close-to-open consistency
      Server reboots slowly, meantime client crashes: violates 30-sec rule
    Note either is acceptable for an NFS implementation

What does speculator look like in the kernel (Fig 2)?
* create_speculation(OUT spec_id, OUT dependencies)
  Often takes checkpoint of process
     Leverage copy-on-write logic in fork to copy memory
       But *don't* place fake child on run queue (i.e., make runnable)
     Also safe file descriptor state, pending signals
   Allocates speculation structure
     Tracks kernel objects modified by speculation
     Adds speculation to process undo log
   Henceforth adds undo log entry added to all modified kernel objects
     These objects *depends on* the speculation
     If any speculation in an object's log fails, revert state
     If all speculations commit, delete log
   When do we *not* need to create a checkpoint?
     New speculation follows read-only speculation less than 500ms ago
     If second speculation fails, just rewind the first one as well
     How do we know last speculation read-only?
       Can tell from log on process
     Why always checkpoint after 500ms?
       Limit amount of work you need to re-execute
       Limit delay in speculate-printf-speculate scenario (Sec 5.3 end)
* commit_speculation(spec_id)
    Reclaim all the resources associated with fake child process
* fail_speculation(spec_id)
    Original parent marked for termination--exits next time scheduled
      OS guarantees none of its side effects were externally visible
    Fake child assumes identity of failed parent

PID, files, pending signals, etc.
Fake child re-executes failed system call non-speculatively

What two invariants prevent speculative state from leaking (Sec 5.3)?
1. Never show speculative state to user or any external device
2. A process can't see speculative state unless dependent on the speculation
   Gives two options when a process tried to view speculative state
   - Block until speculation resolved
   - Become dependent on speculation and subject to rollback

Note: amenable to incremental implementation strategy
First block all system calls while speculating
Then allow simple read-only ones (e.g., getpid)
Then allow speculative operations on speculative file systems
But some file systems only support read speculation
Also note Unix-domain sockets, fifos special (destructive reads)
Change external writes to net/screen to be buffered, not blocking

What happens when processes communicate through tmpfs?
Fig 3: tmpfs inode acquires dependency on first process's speculations
Sec 6.3: rmdir affects directory and inode
Any process that reads directory will depend on speculation
If speculation fails, reinsert directory entry

What about an on-disk ext3 file system?
Must obviously block fsync/fdatasync until no speculations
"no steal" policy: Never write speculative state to disk
Term from DB--can't steal dirty physical page from uncommited transaction
Means you can always fail speculation by blowing away cache
Reread from disk will return prior state
What complications arise with superblock/bitmaps:
Accessed so much would propagate way too many dependencies
Would constantly be speculative and could never be written back
Superblock/bitmap solution?
Observation: Okay for speculation to affect inode/block allocation
E.g., applications don't care what inode number they get
So all processes can see speculative metadata without acquiring dependency
For writeback, keep 2 copies: actual (non-speculative) and shadow
For each speculation, keep both undo and redo operations
On commit, apply redo to actual state
On abort, apply undo to shadow state
Always write back actual state. Depends on commutativity of operations

What happens with pipes/fifos?
Save read data so it can be put back on failure
Keep count of written bytes so they can be "unwritten"
Track create/delete; don't destroy pipe while speculations pending
Unix sockets?
Straight forward, but always propagates speculations bidirectionally
What is complication with signals?
Much easier to have a process to checkpoint itself
1. Add signal to speculative sigqueue
   Propagate dependencies from receiver to sender (who learns kill ok)
2. Linux already checks for signals on return from kernel mode
   If speculative ones, take checkpoint and dependencies other way
   Checkpoint flagged so as not to re-execute syscall (normal behavior)
   Must add signals to undo log

What about multithreaded applications? Not supported (6.10)

What happens with "ls /dfs/foo > /dfs/bar" (Sec. 7)?
File server always knows correct up-to-date state of file system
Server never returns speculative state
So dependencies never propagate across clients
So modify server to check speculation hypotheses & only mutate if true
Hypotheses are dependencies returned by create_speculation

    So include these in any RPC so server can check them
    Requires modifying protocols to include hypotheses
  Requirements to make this work:
    Server must keep per-client list of failed speculations
    Server must process messages in order (use sequence numbers)
  Chose representation for dependencies that compresses well
    <pid,sped_id> where spec_id is counter
    Concisely encode ranges of spec_ids

What about "ls /dfs1/foo > /dfs2/bar"?  Doesn't work (end of Sec 7.1)

What is group commit and why do we care?
  Don't do one mutation per back-to-back RPC
    Instead, save up many writes and do them all at once
    Use speculation to absorb the added latency
  This is really good with a write-ahead log.  What's this?
    Write changes to a log before writing them to permanent location
    Sequential writes to a log can achieve full disk bandwidth
    If you crash before updating in place, just replay log
  Does NFS have a write-ahead log?  No.  Then why does group commit help?
    Get better disk arm scheduling if many concurrent writes
    Can coalesce writes to the same block
  What happens if server crashes during NFS group commit
    Uh oh, no write-ahead log so arbitrary subset of ops may be written back
    Okay-ish because client will keep retransmitting until committed
      Means after crash, other clients may see things happen in weird order
      Could even conceivably see operation undone then redone
      Earlier we said some NFS3 implementations do this anyway
      Just slightly weird to compare performance to NFS that doesn't

What is complication with create and mkdir?
  These return file handles--how to speculate on file without its handle?
  Solution: create alias handles that server maps to real handles
    Replace alias with real handle at client as soon as you learn it
    How to you avoid picking an alias that collides with real handle?
      Paper unclear on how handle alias mappings maintained
      Maybe encode by file handle length (NFS3 allows variable-length FH)
      Create, mkdir would need extra argument field to specify the alias

How is eval?

Open problems before you would use this in practice?
  Multiple file servers
  Dealing with legacy clients/servers
  Multi-threaded programs
  mmap of remote file system?