Dynamic iSCSI at Scale: Remote paging at Google

Nick Black <nlb@google.com>

Linux Plumbers Conference, August 2015, Seattle

Goals of this presentation

- Discuss remote paging of binaries at scale, and its motivation
  - Experimenting with paging binaries and their support data from remote, fast storage
  - This requires a robust implementation of highly dynamic iSCSI
- Share our experience with iSCSI on Linux
  - What's working well? What could be improved?
  - How is our use case different from typical ones?
  - In what ways have we needed to modify the kernel?
- Learn what we could do to improve our use of iSCSI / kernel implementation
  - We'd like to become more involved in Linux's iSCSI and block device projects

Performance problems with local cheap disks

- ❑ Lowest throughput of the local memory hierarchy
- ❑ Highest latency of the local memory hierarchy
- ❑ Unpredictable behavior, especially under load
- ❑ Fetch + page-in times can dominate a task's runtime
- ❑ Slow power control transitions
- ❑ Slowest task in a highly parallelized pipeline can slow down entire job

The cluster enlarges our memory hierarchy

- Thousands of machines, each with some number of
  - Multicore processors with multilevel SRAM/EDRAM caches
  - DDR3/DDR4 DRAM DIMMs (possibly NUMA)
  - Flash storage and/or magnetic storage (IOCH and/or PCIe)
  - Gigabit Ethernet or 10GigE NICs (PCIe, possibly channel-bonded)
- Cluster (common power sources, flat intracluster network bandwidth)
  - Tens of Gbps to each machine from single Tbps switch
  - Single Tbps to each switch in tens-of-Tbps superblocks
  - Tens of Tbps to each superblock in Pbps cluster fabric
  - Tens of thousands of machines in a cluster

Memory hierarchy of generic warehouse computers

- DRAM provides hundreds of Gbps, low hundreds of ns latency, fed by either...
  - PCIe 3.0 x8: 63Gbps, µs latency +
  - 10GigE NIC: 10Gbps, several µs latency (plus wildly variable remote serving latency)

...or...

  - Local SATA3: 4.8Gbps, µs latency +
  - Local SSD: low Gbps, µs latency or
  - Local HDD: low hundreds of Mbps, tens of ms latency, terrible tail latency

Better performance through network paging pt 1

- The SATA3 bus provides 4.8Gbps of usable throughput, but...
  - A low-cost drive might average ~800Mbps on realistic read patterns
  - ...and average several tens of milliseconds of seek time for each chunk
- The network can provide 10Gbps of usable throughput
  - PCIe bus and QPI can handle it
  - Dozens of times more bandwidth than the SATA3 bus
  - Latencies in microseconds
- Disk server can saturate the network
  - Caching effects among machines leaves common data in disk server DRAM
  - Disk servers can be outfitted with expensive high-throughput store (PCIe SSD etc.)
  - Write case can't take advantage of intermachine caching, but the network won't introduce delay compared to local disk write (it can take advantage of quality remote store)

Better performance through network paging pt 2

- Take advantage of demand paging
  - No longer sucking down the full binary + data set to disk
  - Grab, on demand, only the pages we need from remote
  - Fewer total bytes transferred
  - No useless bytes going through local/remote page caches
- Take full advantage of improving technologies
  - CPU, memory, and disk size are all getting better
  - Spinning disk seek times, throughput seem at a wall
  - Spinning disk performance / size ratio is getting steadily worse
    (efficient utilization of magnetic storage results in steadily worsening performance)

Binaries and support files: read-only iSCSI

- Packages built as ext4 images+metadata
    - Kept in global distributed storage (POSIX interface, smart redundancy, etc.)
- Pushed on demand to disk servers implementing custom iSCSI target
    - Lowest-level distributed filesystem nodes: no redundancy at this level
    - Distribution infrastructure maintains a ratio of reachable copies per task
    - Pushes new target lists to initiator to allow dynamic target instances
- Custom iSCSI initiator drives modified Linux kernel iSCSI-over-TCP transport
    - Sets up a dm-verity device atop a dm-multipath (MPIO, not MC/S)
    - Connects to multiple independent remote iSCSI targets
    - Hands off connections to the kernel, one to an iSCSI session
        - Makes new connections on connection failure or if instructed

Load balancing through dm-multipath

- Round-robin: Fill up the IOP queue, then move to the next one
  - We have purposely set target queue depths set fairly low; would result in rapid cycling
  - Doesn't allow backing off from a single loaded target
- Queue length: Select path based off the shortest queue
  - Bytes per IOP are dynamic, but prop delay is likely less than round-trip time
- Service time: Dynamic recalculation based on throughput

Locally-fetched package distribution at scale pt 1

- Alyssa P. Hacker changes her LISP experiment, perhaps a massive neural net to determine whether ants can be trained to sort tiny screws in space.
  - Assume 20,000 tasks, immediately schedulable
  - Each task instance needs 3 packages, totalling .5GB (4Gb)
  - Expected CPU time of each task, assuming an ideal preloaded page cache, is 120s
- 20K tasks * 4Gb compulsory load == 80Tb mandatory distribution
  - Assuming 10Gbps bandwidth, ideal page cache, and ideal disk...
  - Serialized fetches: Average task delayed by 4,000s, 97% of total task time, 33x slowdown
    Worst case task (8,000s) paces job: 2hr+ to job completion, **6677% slowdown**
  - Fully parallel fetches: Ideal exponential distribution (requiring compute node p2p) requires lg2 (20000) = 15 generations of .4s each, worst case 6s, job requires 126s, **5% slowdown**
  - We can approach .4s total by initiating p2p send before complete reception, **.03% slowdown**

Locally-fetched package distribution at scale pt 2

- Introduce a single oversubscribed compute node fetching to contended disk
  - The process must evict 128MB of (possibly not yet written-through) data
  - Another process acquires and releases 128MB, possibly requiring a load from disk
  - The process pages back in some or all of its 128MB
- If each phase takes 2s, 6s is added to the task runtime.
  - Worst task cases are now 6s, **5% slowdown**
  - In reality, many such delays accumulate for at least one task, all due to paging to/from disk
- A damaged sector might result in a 30s delay, **25% slowdown**

Remotely-paged packages at scale

- No compute node peer-to-peer (p2p retained in target distribution level)
  - Assume *n* compute nodes per disk nodes
  - We can distribute in time approaching one copy with exponential p2p (.4s)
  - *n* compute nodes then grab *p* pages of *P* total, worst case approaches .4s * (np/P + 1)
- Only demanded pages traverse page caches or networks
  - Fewer compulsory delays offsets lack of last-level p2p
  - Compulsory delays are smoother over the life of most tasks
  - Task container can be allocated less memory
- Eliminate the annoyances of local spinning disk
  - Tail latencies are much better controlled -- very few slow / contended reads
  - Redundancy -- dm-multipath allows us to fail over quickly
  - Permit radical new physical setups

Coping with an unreliable userspace iscsid

- Kernel expects an userspace iSCSI control daemon to always be around
  - Alas, this expectation cannot always be met (OOMs, crashes, load, etc.)
  - Restart/schedulability might take time, races result in lost kevents
- Becomes particularly problematic in the face of connection errors
  - We want immediate failover to a standby session via dm-multipath
  - iSCSI wants to do connection recovery via external agent
  - No one seems to know whether kernel MC/S works (OpenISCSI initiator doesn't use it)
- We disable the connection recovery timeout, immediately hitting error path
  - Session dies, error bubbles up to dm-multipath, immediate failover
  - Userspace initiator gets to it eventually and creates a new session for multipath device

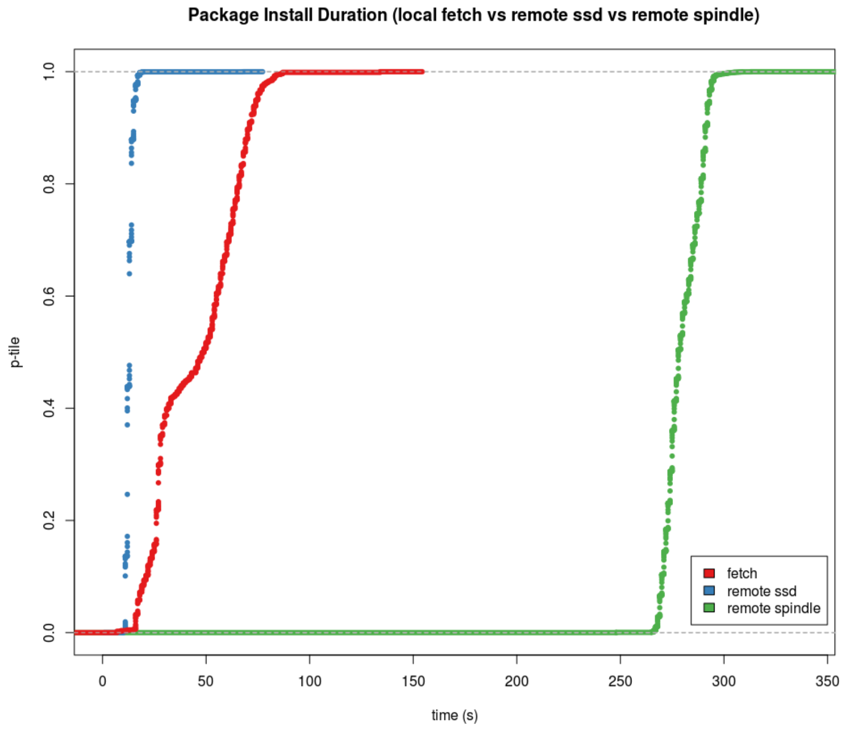
User-initiated stop can race with kernel

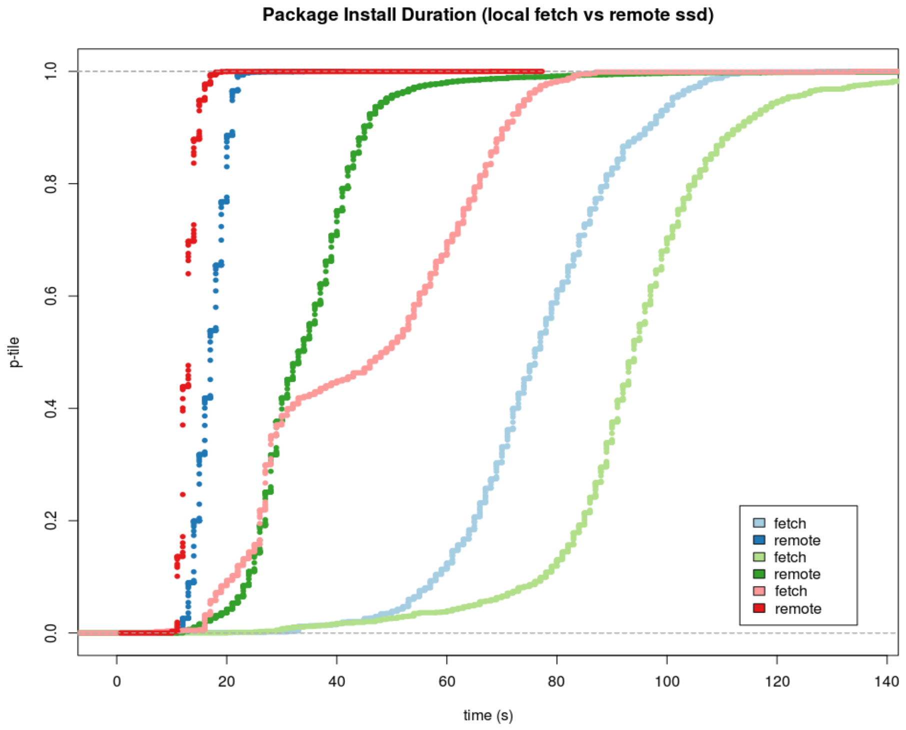
- We still want to deliver the connection stop message, but we don't want to delay connection teardown waiting for userspace.
- Can't just disable userspace-initiated connection stop, as it's necessary for changing up targets and standard client-side termination.
- Added locking to `iscsi_sw_tcp_release_conn`
- Messy interaction between `sk->sk_callback_lock` and `tcp_sw_conn->lock`
- Upstream indicated lack of interest in this solution, but it seems difficult to do reliable, fast fail recovery with MPIO without it, and upstream doesn't want MC/S on the initiator side

Why no MC/S (Multiple Connections per Session)?

- LIO in-kernel target does support MC/S
- Competitor initiator+targets support MC/S
- There's at least some support in the kernel dataplane initiator
  - What is the state of this code? Userspace initiator doesn't use it
- MC/S only supports one target within the session
  - No good for multitarget load balancing
- Mailing list has pushed for MPIO (dm-multipath) to be used exclusively
  - Requires reliable termination of sessions with failed connections (previous slides)
  - Ignorance of command numbering complicates load balancing
  - Difficult to rapidly recover from temporarily-unavailable targets

Winning: lower job start times





Winning: faster tasks

- These graphs reflect a 3.11.10-based kernel
- Missing scsi-mq and other 2014/2015 improvements
- Nowhere near theoretical ideal, but already a big win
- 4.x.x rebase ought improve things for free