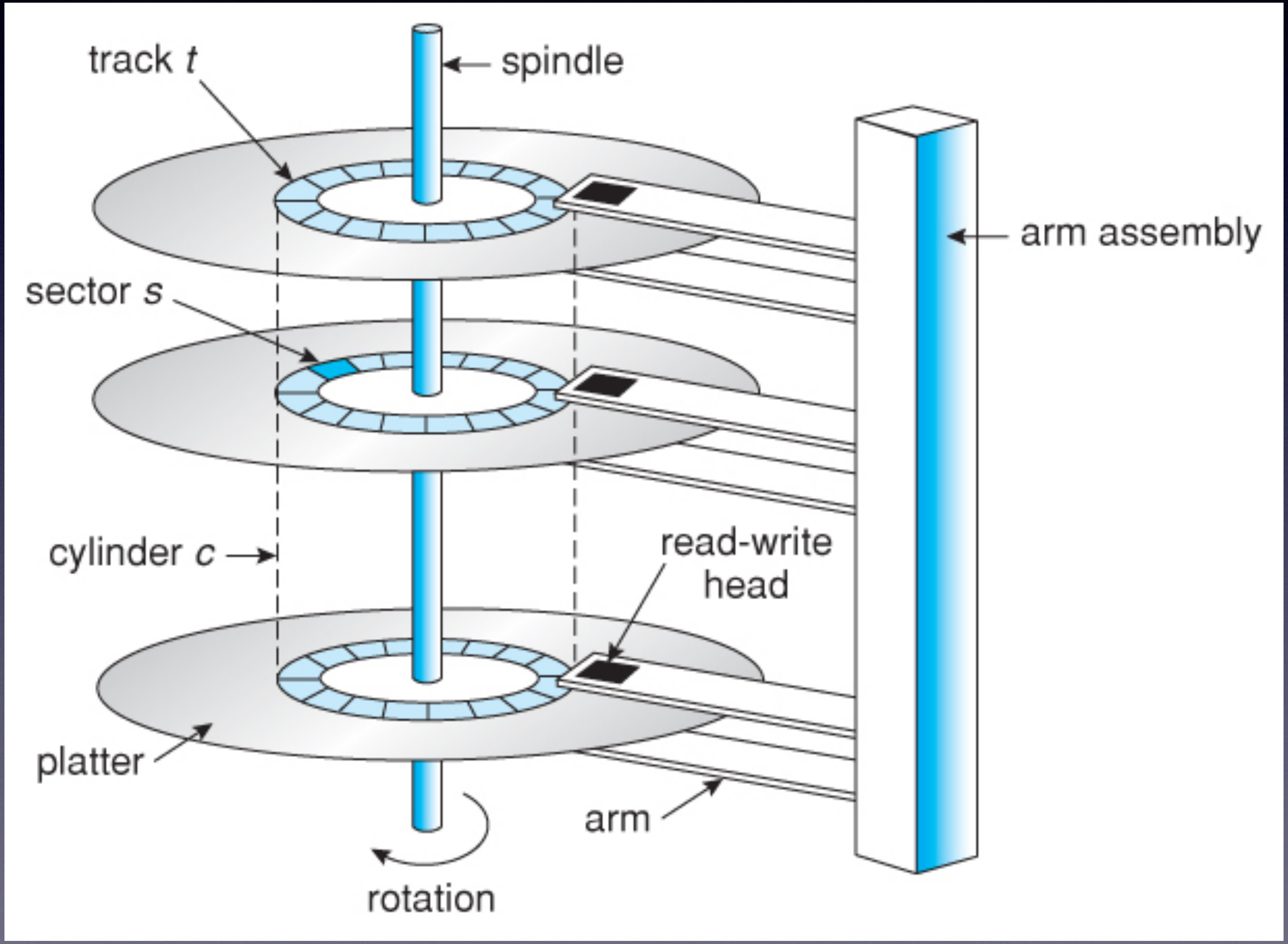
Log-Structured File Systems

Outline

- Unix Fast File Systems
- Log structured file systems

Disk Structure



Background

- Directory: maps name to I-node number
- I-node: structure for per-file metadata
  - contains ownership, perms, timestamps + 10 datablock pointers
  - form an array, indexed by "i-number"
  - array is explicit in Unix File system, implicit for LFS
- Indirect blocks:
  - i-node only holds a small number of datablock ptrs
  - for larger files, i-node points to an indirect block, which in turn points to the data blocks
  - can have multiple levels of indirect blocks

Unix File System

- Original Unix file system was simple and elegant, but slow
    - achieved only about 2% of disk bandwidth
- What can explain such bad performance?
    - Do we still care about file system performance?

Unix File System

- Problems:
  - blocks too small
  - consecutive blocks of files not close together
  - i-nodes far from data
  - i-nodes of directory not close together
  - no read-ahead

Unix Fast File System

- Larger block size (4K to 8K)
  - why not choose even larger blocks?
- Disk divided into cylinder groups
- Each contains super-block, i-nodes, bitmap of free blocks, usage summary information
- I-nodes are now spread across the disk
  - keep i-node near file, i-nodes of a directory together
  - cylinder groups ~ 16 cylinders

Locality

- Key ideas:
  - don’t let disk fill up in any one area
  - paradox: to achieve locality, must spread unrelated things far apart
  - result: achieved about 20% of disk bandwidth

Locality Policy

- Keep directory within a cylinder group, spread out different directories to other groups
- Allocate runs of blocks within a cylinder group; every once in a while, jump to a new cylinder group
- Layout policy: global & local
  - global policy allocates files & directories to cylinder groups
  - local allocation search order:
    - rotationally closest in current cylinder, current cylinder group, hash to another cylinder group

LFS

- Radically different file system design
- Technology motivations:
  - CPUs outpacing disks
  - Big memories
  - Disks becoming more complicated
- What are the implications of these tech trends?
  - Are they still relevant today?

Implications/Problems

- Lots of little writes
  - because reads are taken care of
  - because most files are small
- Synchronous: wait for disk in too many places
  - because of recovery concerns
- 5 seeks to create a file:
  - file i-node (create), file data, directory entry, file i-node (finalize), directory i-node (mod time)

Basic Idea of LFS

- Log all data and meta-data with efficient, large, sequential writes
- Log is the “only and entire” truth, there is nothing else
  - need to keep an index of the log’s contents
  - rely on a large memory to provide fast access through caching
- Turn the disk into a tape!

Two Potential Problems

- No update-in-place; (almost) nothing has a permanent home
  - so how do we find things? (log retrieval)
- Wrap around: what happens when end of disk is reached?
  - no longer any big, empty runs available
  - how to prevent fragmentation?

Log Retrieval

- Keep same basic file structure as Unix (data, inode, indirect blocks)
- Let i-nodes float, so we need to find a file’s inode
  - Solution: an “inode map” that tells position of inode
  - inode map gets written to log like everything else
  - So need “map of inode map” to keep track of inode maps; small enough to be in memory
  - Map of inode map gets written in special checkpoint location on disk; used in crash recovery

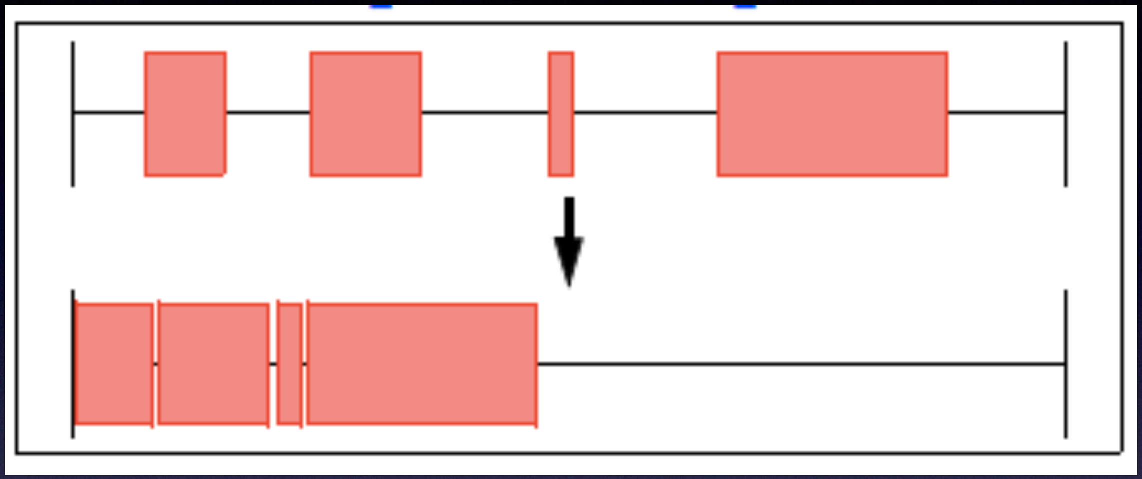
LFS Data Structures

- Read:
  - follow: map of inode map, to inode map, to inode, to block
  - get some locality in inode map; cache a lot of it in memory
- Recover:
  - read checkpoint, get map of map
  - roll forward in log to update map of map

Two Potential Problems

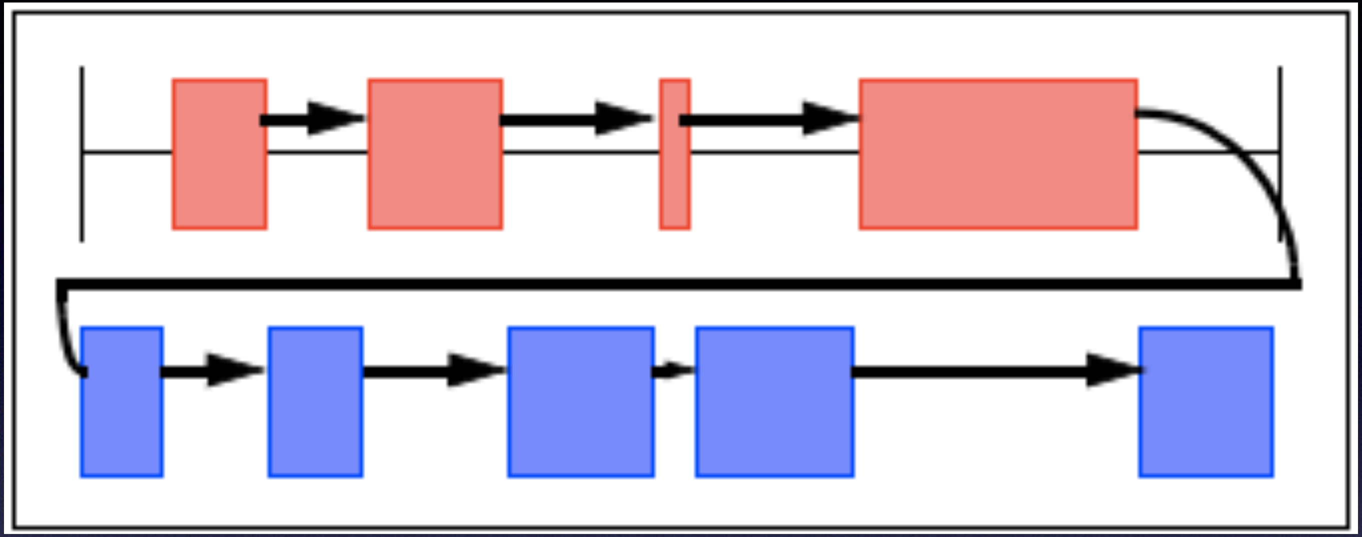
- No update-in-place; (almost) nothing has a permanent home
  - so how do we find things? (log retrieval)
- Wrap around: what happens when end of disk is reached?
  - no longer any big, empty runs available
  - how to prevent fragmentation?

Approach #1: Compaction



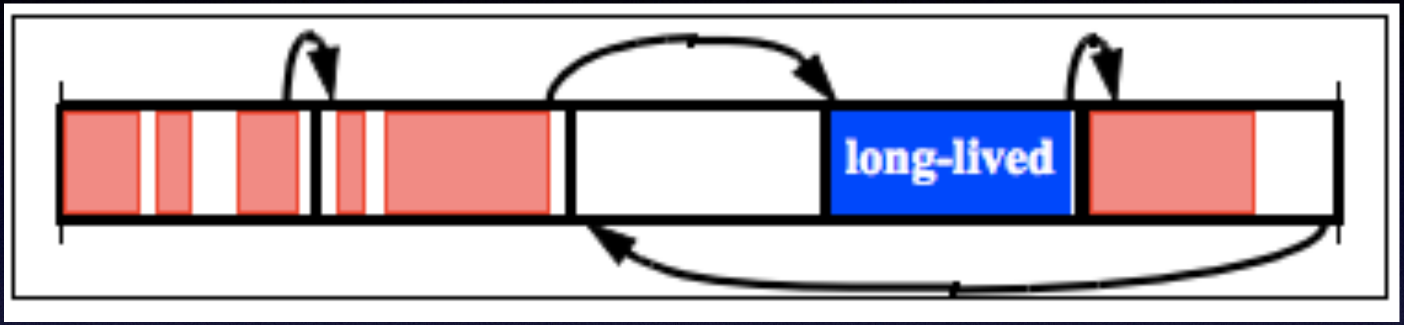
- Works fine if you have a mostly empty disk
- But suppose 90% utilization:
  - write 10%
  - compact 90% (read 90%, write 90%)
  - repeat!

Approach #2: Threading



- Fill in empty spaces
- Start at the beginning of disk once you reach end
- What is the problem with this approach?

Solution: Segmented Log



- Use both compaction & threading
  - compaction: big free space
  - threading: leave long living things in place & don’t copy
- Segmented log:
  - chop disk into a bunch of large segments
  - compaction within segment, threading among segments

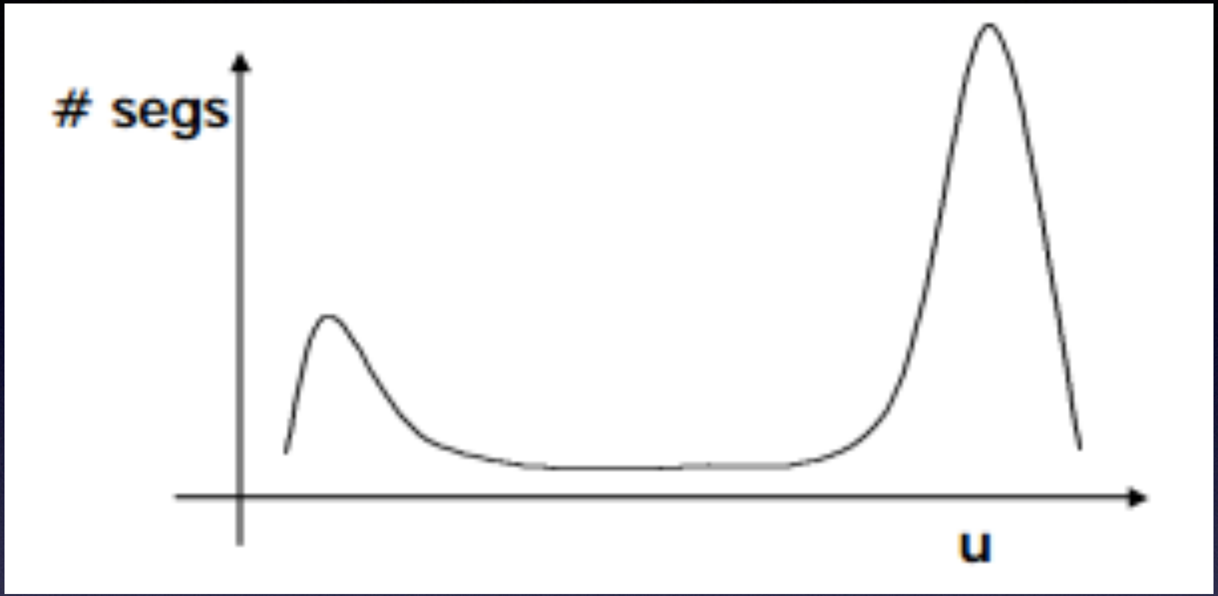
Segmented Log (contd.)

- When writing, use only clean segments (i.e., no live data)
- Occasionally clean segments:
  - read in several, write out live data in compacted form, leaving some segments free
  - try to collect long-lived information into segments that never need to be cleaned

Cleaning Issues

- Which segments to clean?
- What information to keep track per segment? (and how to keep track of them)

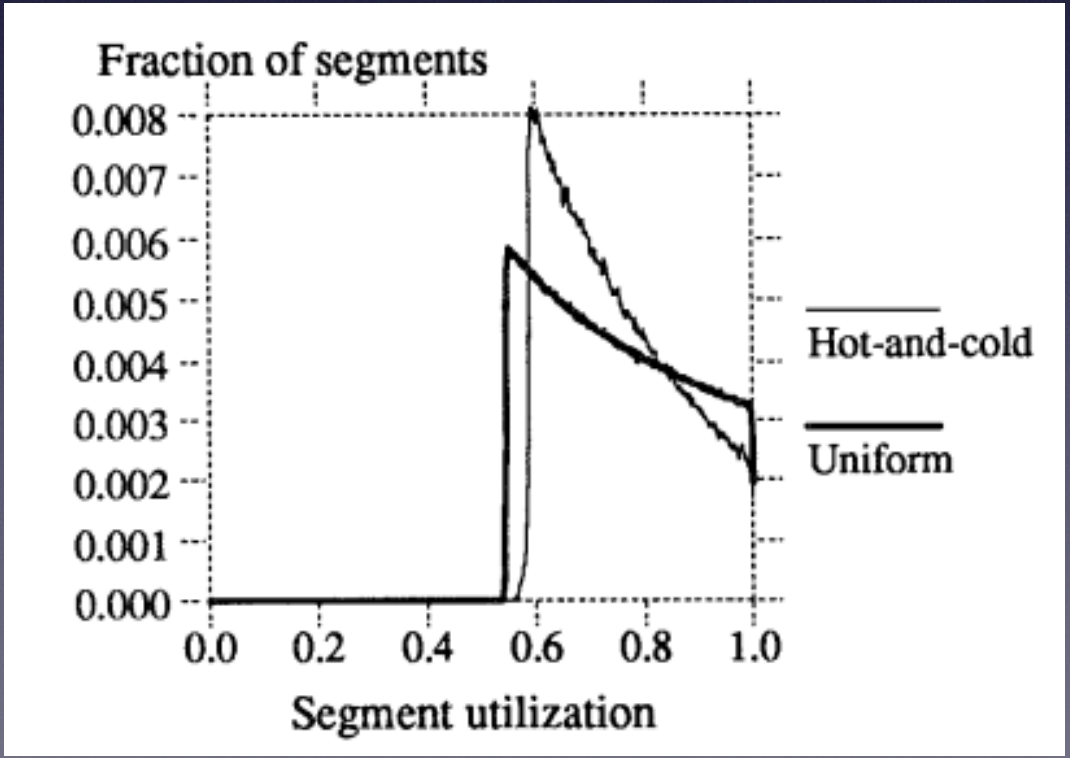
Cleaning Goals



- Want bimodal distribution:
  - small number of low-utilized segments (so cleaner can find easy segments to clean)
  - large number of high-utilized segments (so disk is well utilized)

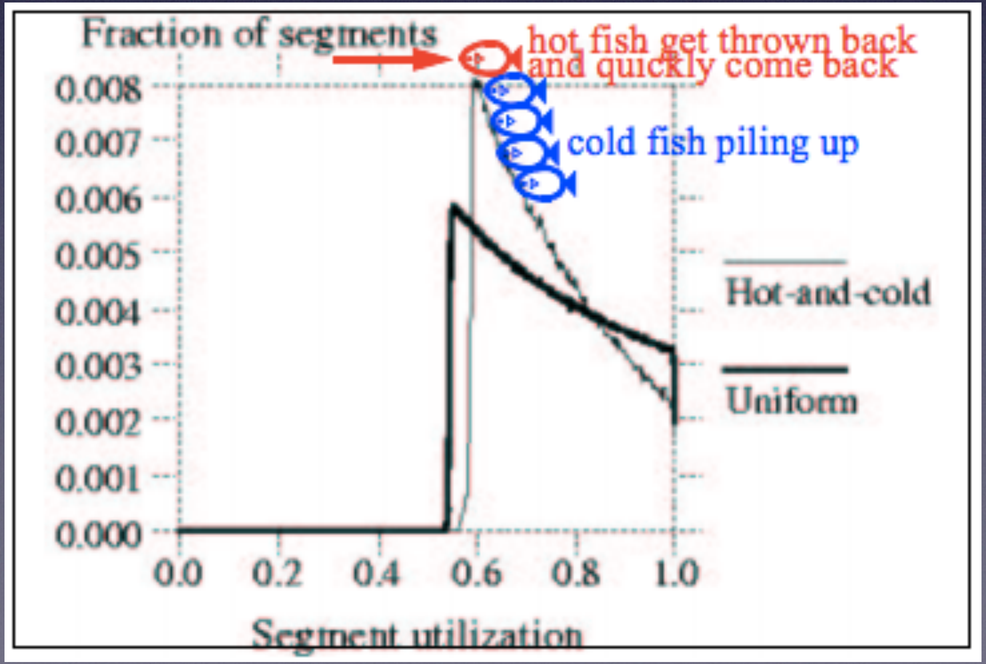
Greedy cleaner

- Pick the lowest util to clean
- Works not so great for random workload
- For “hot-cold” workload: even worse



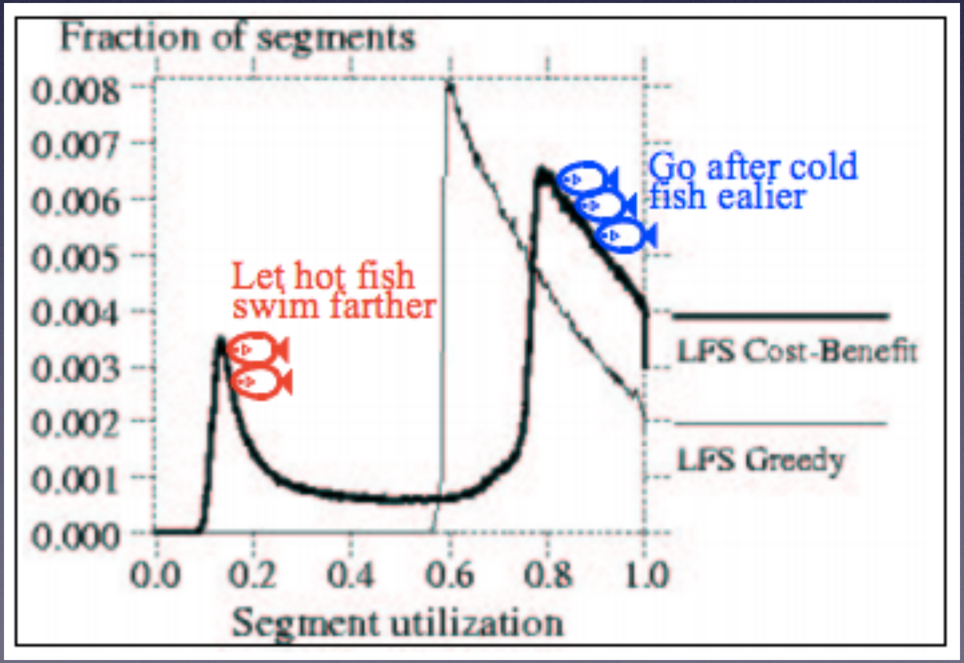
Induce Bi-modal

- Segments are like “fish”: swimming to the left
- Cleaner spends all its time repeatedly slinging a few hot fish back
- Cold fish hide lots of free space, but cleaner can’t get to them fast



Induce Bi-modal

- Cold segment space more valuable: if you clean cold segments, takes them longer to come back
- Hot free space is less valuable: might as well wait a bit longer



Key Feature of the Paper

- Keen awareness of technology trends
- Willing to radically depart from conventional practice
  - Yet keep sufficient compatibility
- Provide insight with simplified math
- Simulation to evaluate and validate ideas
- Rethink what is primary and what is secondary in a design