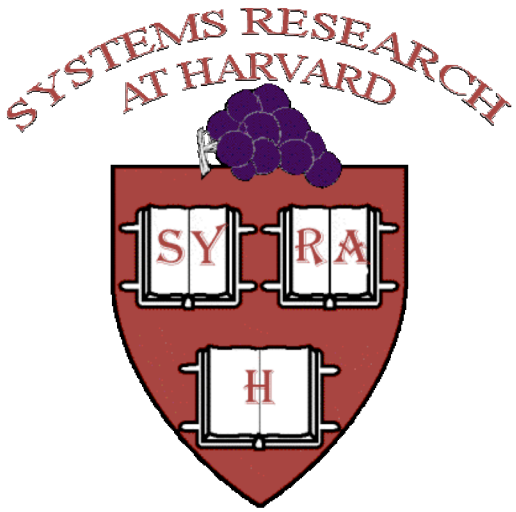
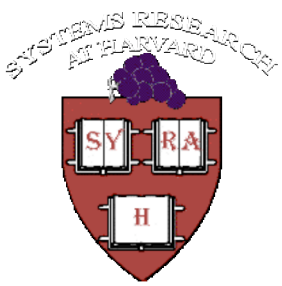
NFS Tricks and Benchmarking Traps



Daniel Ellard and Margo Seltzer

FREENIX 2003 - June 12, 2003



Outline

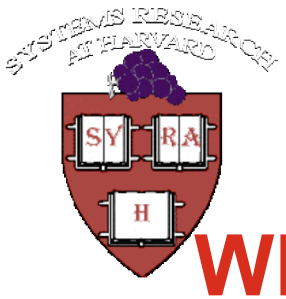
- Motivation
  - Research questions
  - Benchmarking traps
- New NFS Read-Ahead Heuristics
  - Optimize sequential reads
  - Improve non-sequential reads
- Results
- Conclusions

Goal - Improve NFS Read Throughput

- We are interested in improving the throughput of data accessed from disk via NFS.
  - Example: email workload
- Our approach: improve the heuristics that control the amount of read-ahead done by the server.

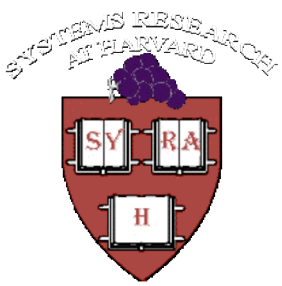
Why Improve Read-Ahead Heuristics?

- With busy NFS clients, 5-10% of NFS requests arrive at the server out-of-order.
- nfsiods are the primary source of reordering.
  - nfsiod is a client daemon that marshals and schedules NFS requests.
  - Many implementations use multiple nfsiods.
  - Contention for resources and process scheduling effects can cause reordering.



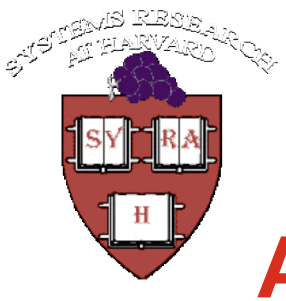
Why Improve Read-Ahead Heuristics?

- Sequential access patterns may appear non-sequential if requests are reordered.
- Servers do less (or no) read-ahead for non-sequential access patterns.
- Read-ahead is necessary for good performance.



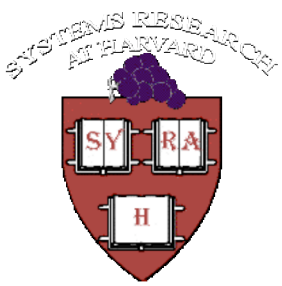
Research Questions

- Can we improve performance for sequential reads by improving the way the NFS sequentiality-detection heuristic handles “slightly” out-of-order requests?
- Can we detect non-sequential access patterns that have sequential components and therefore can benefit from read-ahead?



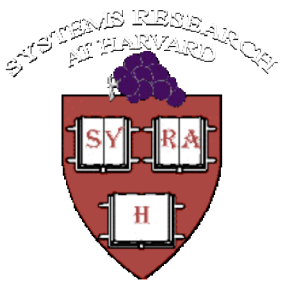
A Micro-Benchmark for NFS Reads

- Long sequential reads
- Many concurrent readers
- Inspired by observed email workloads
- All tests begin with a cold cache on client and server.
  - All data is brought from disk during the benchmark.



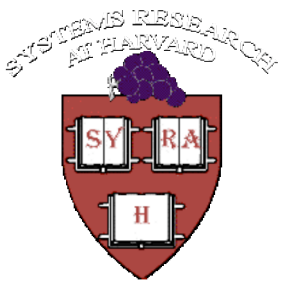
The Testbed

- FreeBSD 4.6.2
- Commodity PCs
  - Note: PCI bus transfer speed of 54 MB/s
- Intel PRO/1000 TX gigabit Ethernet
  - em device driver
  - MTU=1500
  - Raw TCP transfer rate of 49 MB/s
- IDE and SCSI drives
  - Paper discusses SCSI, this talk focuses on IDE



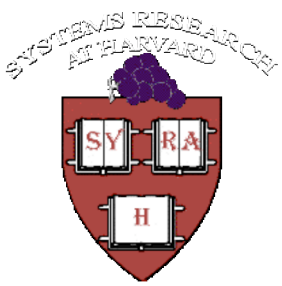
Preliminary Results

- Before measuring the effect of our changes to the NFS server, we must understand the default system.
- Results of our benchmarks were frustrating:
  - Large variance
  - Strange effects
- We decided to investigate these effects before proceeding.



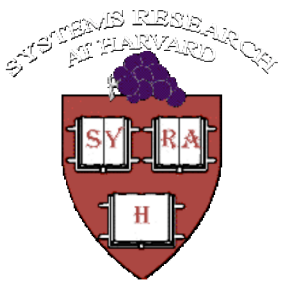
Benchmarking Traps

- Properties of disks and their drivers:
  - ZCAV/disk geometry effects
  - Disk scheduling algorithms
  - Tagged command queues
- Arbitrary limits in the NFS implementation
- Network issues
  - TCP vs UDP for RPC

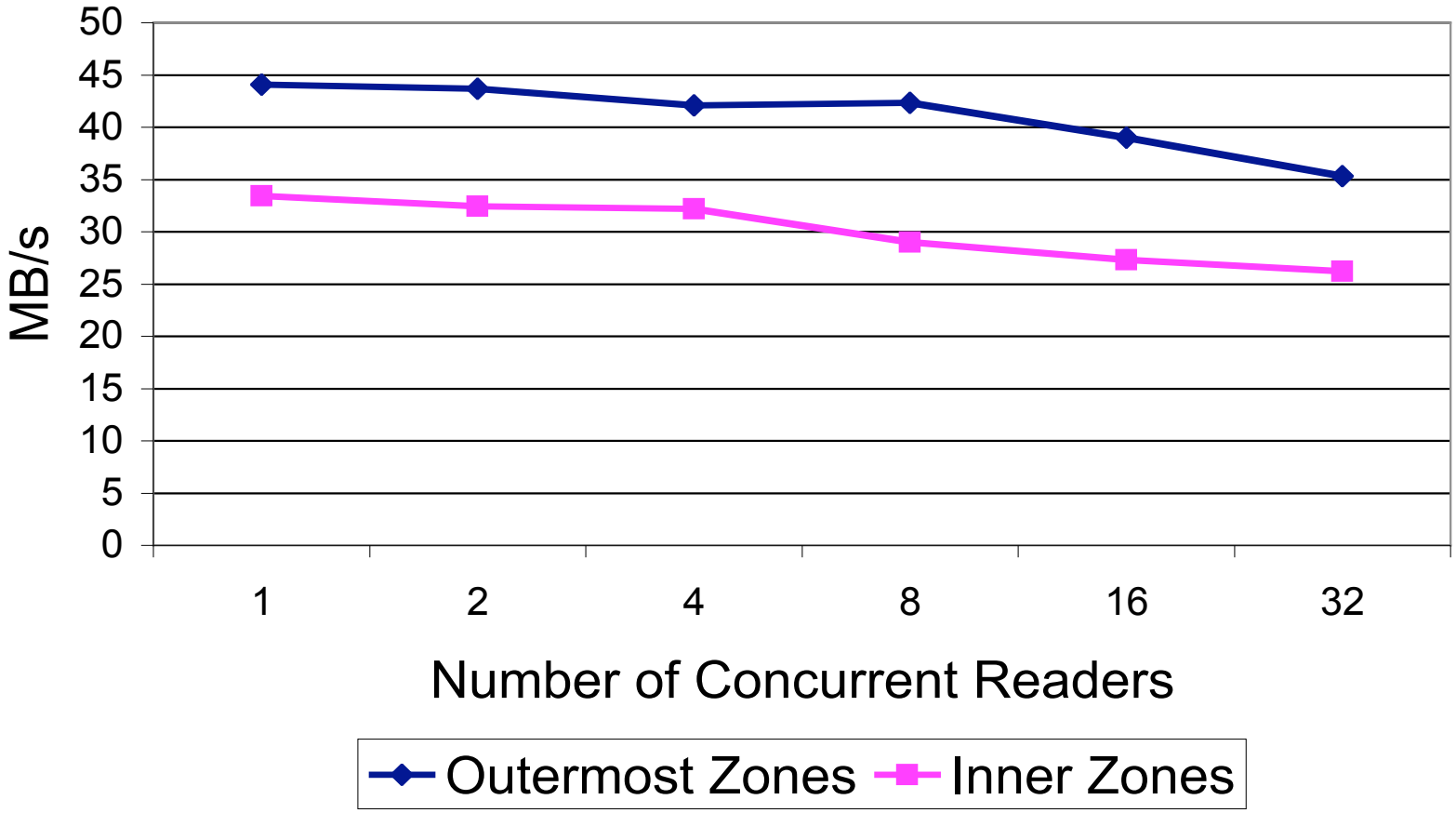


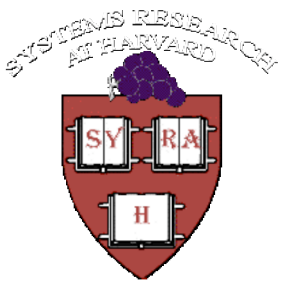
ZCAV Effects

- ZCAV - “Zoned Constant Angular Velocity”
  - Disk tracks are grouped into zones.
  - Within each zone, each track has the same number of sectors.
  - The number of sectors is roughly proportional to the length of the track.
- Tracks in the outer zones hold 1.2 - 2 times more data
  - Outer zone has a higher transfer rate
  - Outer zone requires fewer seeks



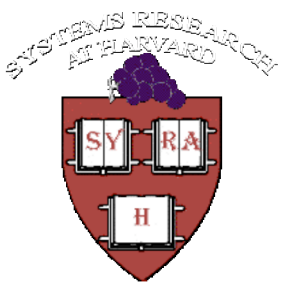
The ZCAV Effect - Local IDE Disk





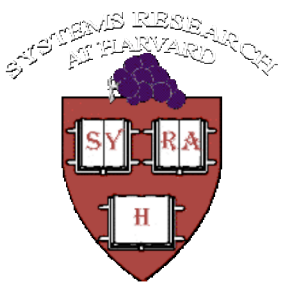
Controlling for ZCAV Effects

- To minimize the ZCAV effect, minimize the difference between the innermost and outermost zones you use.
  - Use a large disk.
  - Run your benchmark in a small partition.
- To measure the effect, create several partitions and repeat your benchmark in each.



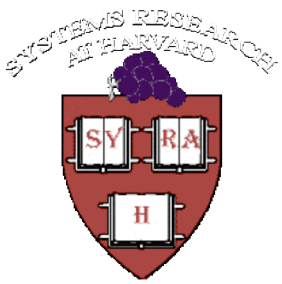
Disk Scheduler Issues

- BSD systems use the CSCAN scheduler.
- CSCAN trades fairness for disk utilitization.
  - Some requests are serviced much sooner than others.
  - It is not hard to create request streams that starve other requests for the disk.
  - Overall throughput is very good.
- Many scheduling algorithms are unfair.



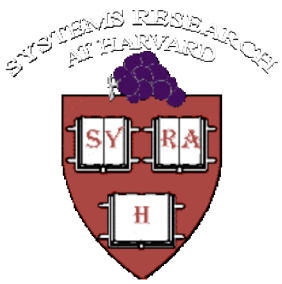
Controlling for Scheduler Effects

- Application specific!
- For our purposes:
  - Total throughput for concurrent readers
  - Measure the total time it takes for all the concurrent readers to finish their tasks, instead of the time of each individual reader.
- There is large variation in the time each reader takes, but the time required by the slowest reader is reasonably consistent.



Tagged Command Queues

- SCSI drives have tagged command queues.
  - Disk requests are sent to the drive as soon as they reach the front of the scheduler queue.
  - The drive schedules the requests according to its own scheduling algorithm.
- For our benchmarks and hardware:
  - Tagged command queues increase fairness.
  - Unfortunately, throughput is reduced (almost 50% in the worst case).

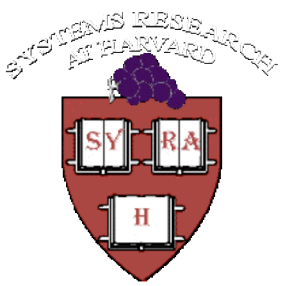


Back to the Experiments…

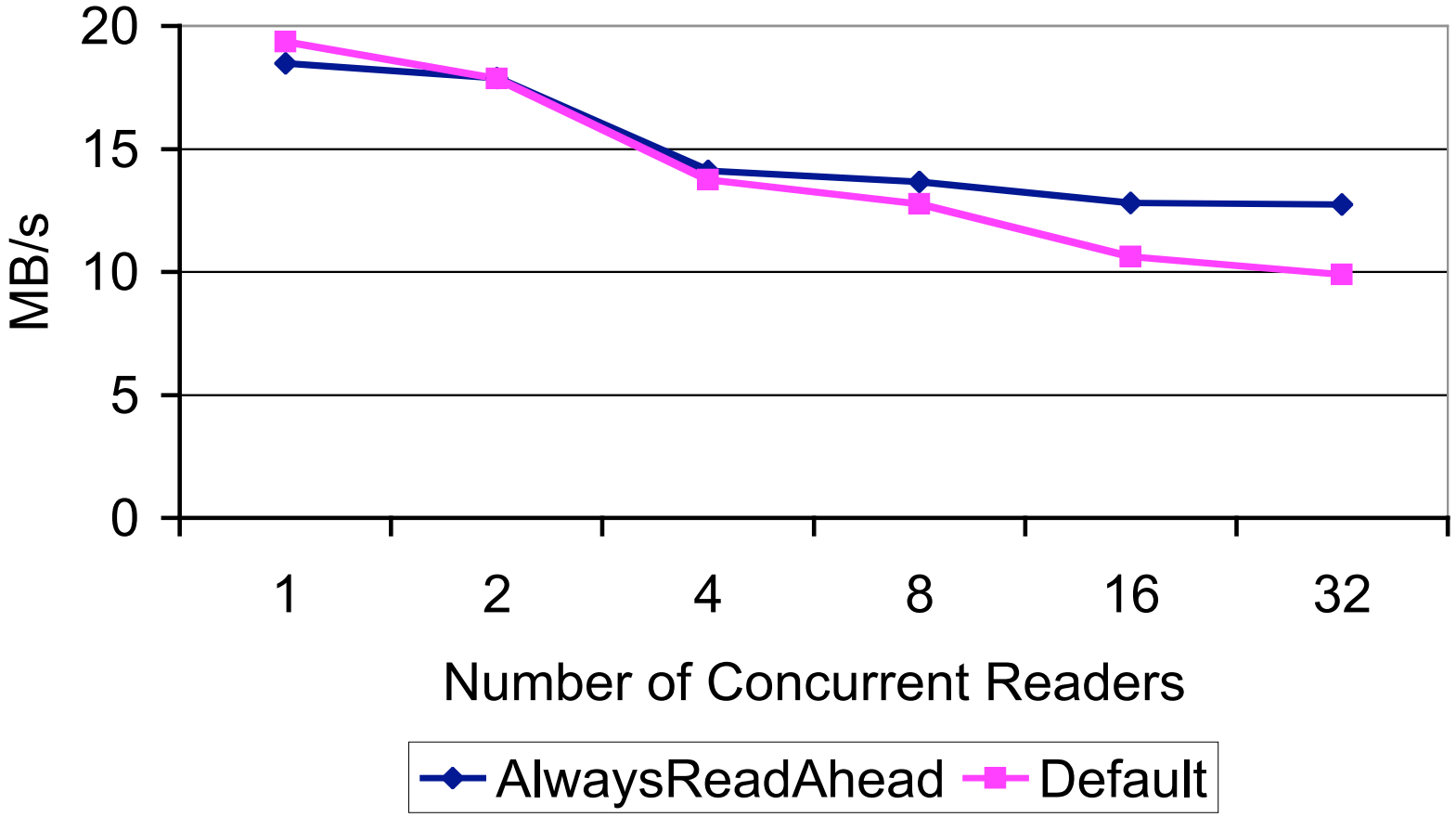
Q: What is the potential for improvement in the read-ahead algorithm?

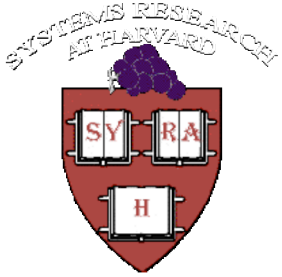
- Compare the default system to AlwaysReadAhead, a system that aggressively always does as much read-ahead as it can.

A: There is benefit when the degree of concurrency is high and requests arrive out-of-order.



NFS Read Throughput (Busy Clients)





The SlowDown Heuristic

Default Heuristic

If the access is sequential relative to the previous access:

seqCount++

else

seqCount = small const

SlowDown Heuristic

If the access is sequential relative to the previous access:

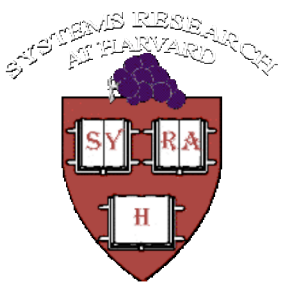
seqCount++

else if the access is “close” to the previous access:

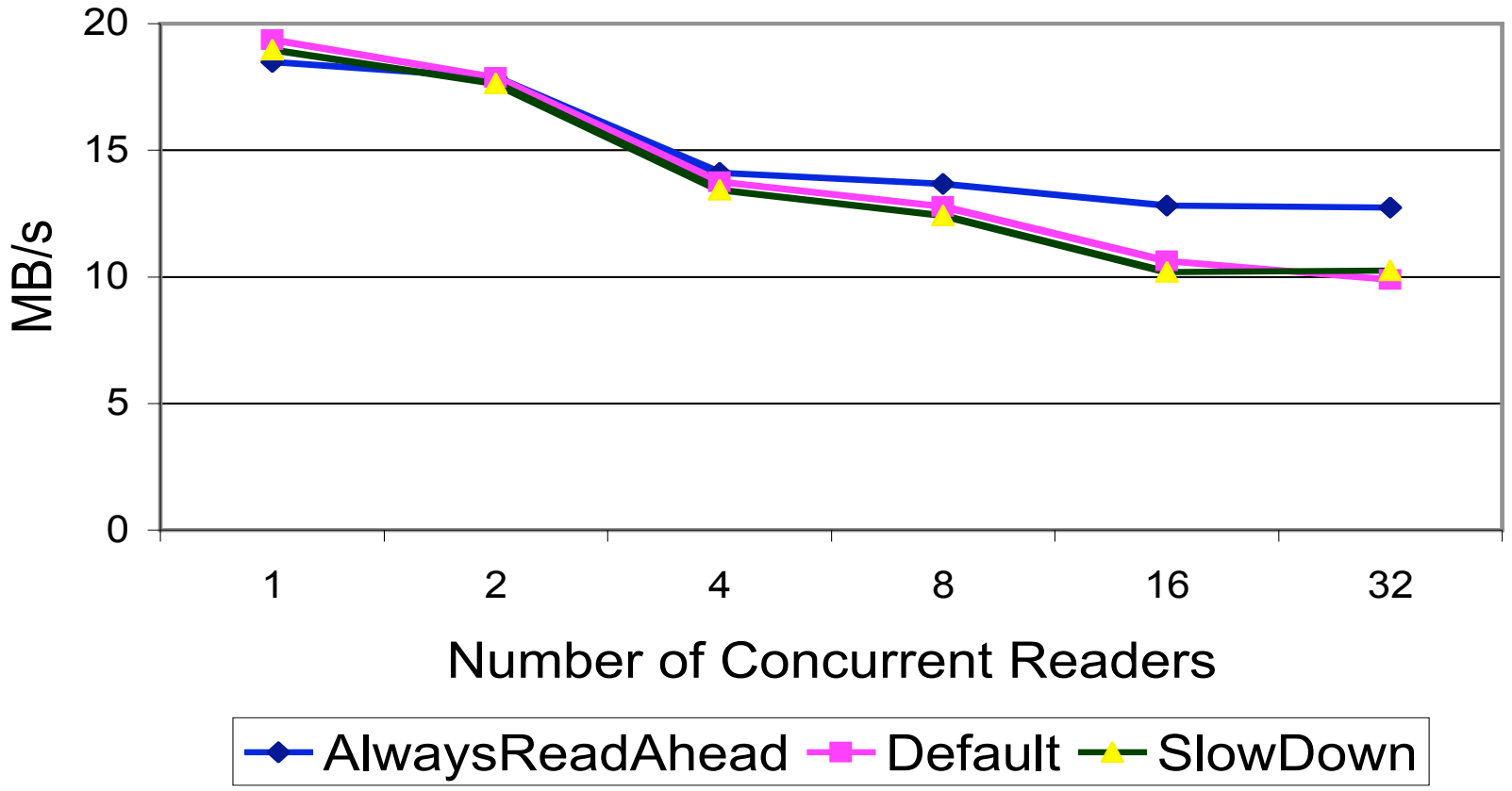
seqCount is unchanged

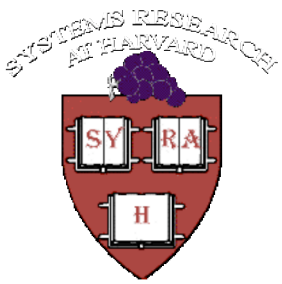
else

seqCount = seqCount / 2



The Effect of SlowDown

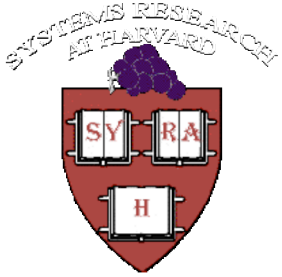




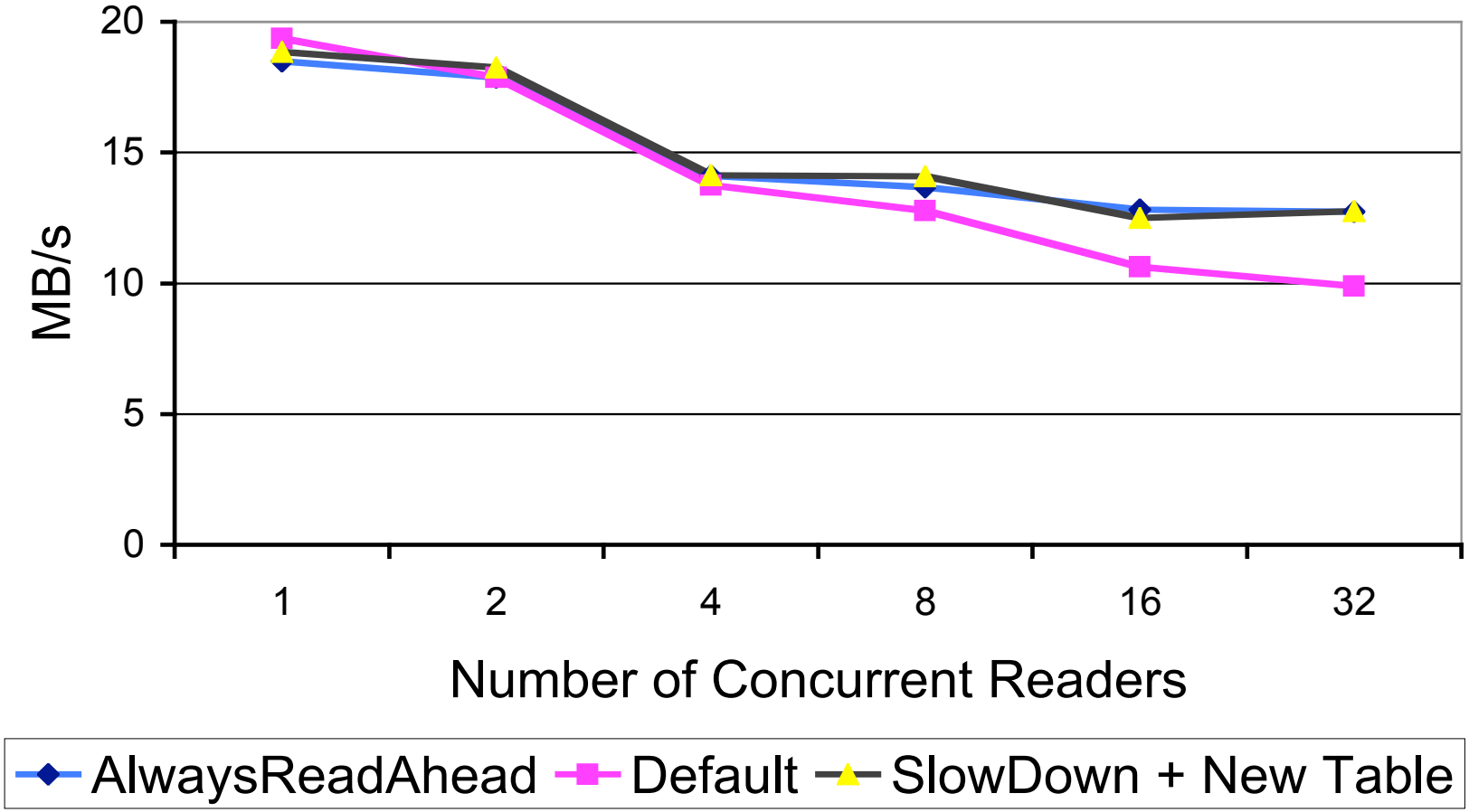
Why Doesn't SlowDown Help?

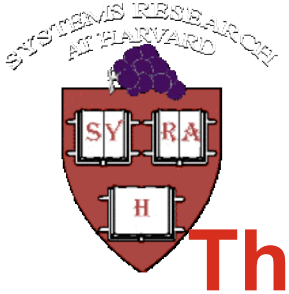
The problem is not SlowDown.

- In FreeBSD, the sequentiality scores are stored in a fixed-size hash table.
- When the table is full, adding a new entry forces the ejection of another.
- The hash table is too small to support more than a few readers.



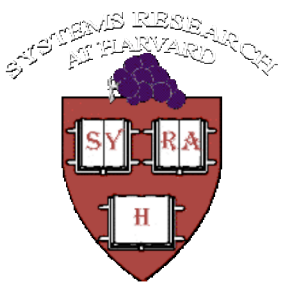
SlowDown with the Larger Table





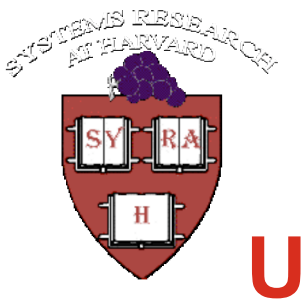
The Effect of Increasing the Table Size

- Increasing the hash table size makes SlowDown as fast as AlwaysReadAhead.
- Fixing the table also makes the ***default*** algorithm as fast as AlwaysReadAhead.
  - For our current testbed, it is enough simply to have a reasonable value for seqCount.
  - Perhaps in the future having a more accurate value will become important.



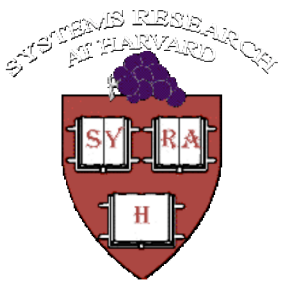
Improving Non-Sequential Reads

- Some read patterns are non-sequential, but do contain sequential components.
- One example is two threads reading sequentially from the same file:
  - Thread 1 reads blocks 0, 1, 2, 3, 4 …
  - Thread 2 reads blocks 1000, 1001, 1002, 1003 …
  - Server sees 0, 1000, 1, 1001, 2, 1002, 3, 1003 …
- This pattern is not sequential according to the default or SlowDown read-ahead heuristics.

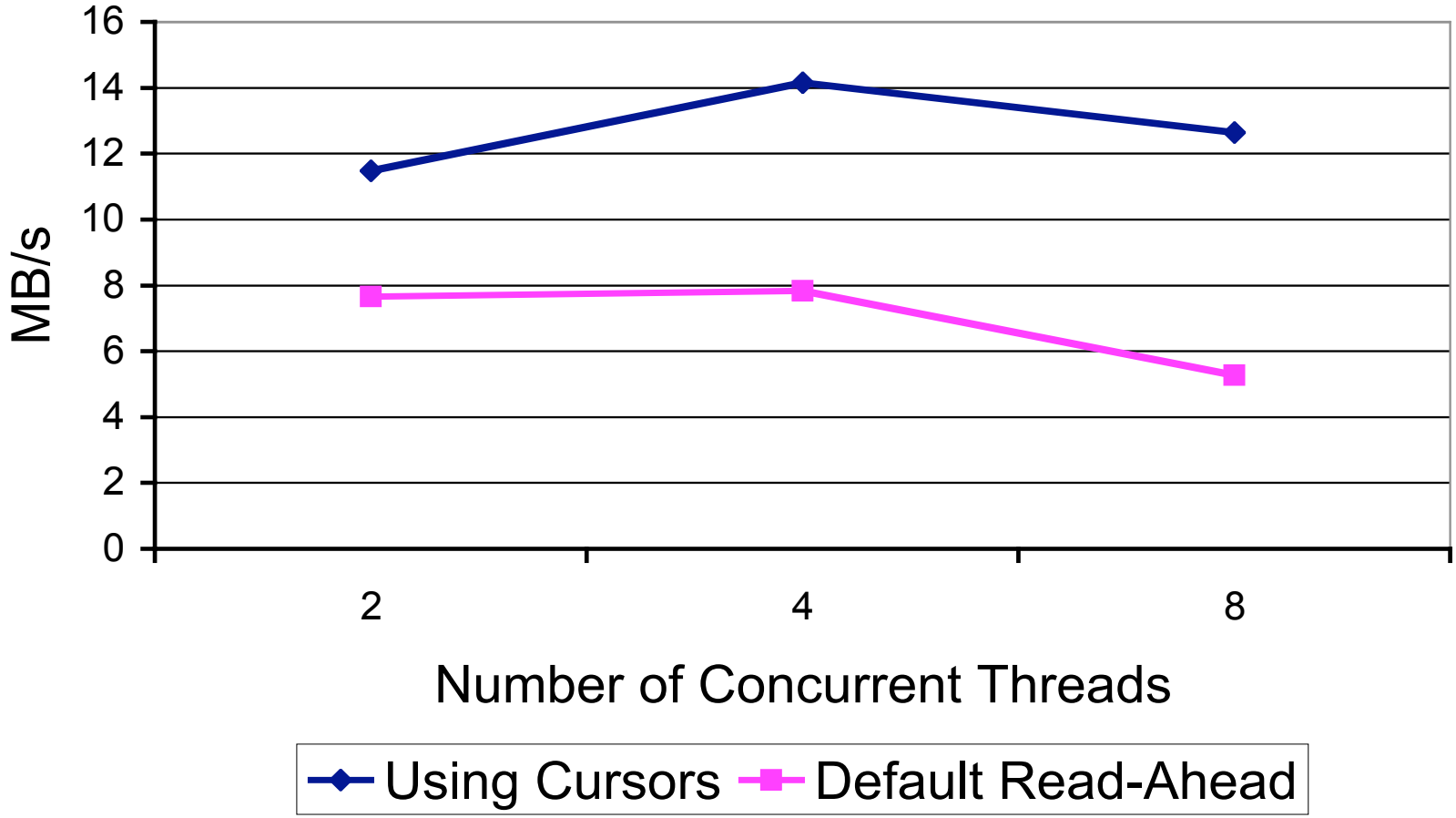


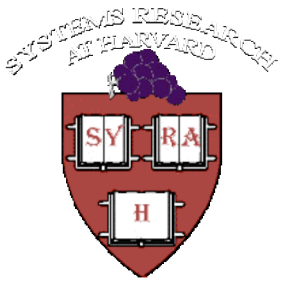
Using Cursors to Find Components

- For each active file, maintain a set of cursors.
  - Each cursor is a position and sequentiality score.
- For each read access to the file, choose the cursor with the closest position:
  - If there is no “close” cursor, create one.
  - If there are already too many cursors for this file, eject the least recently used.
  - Update the sequentiality score for the cursor.



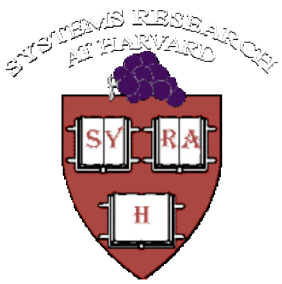
The Effect of Cursors





Conclusions

- The SlowDown heuristic does not help much, at least not for our system.
  - Fixing the hash table does help
- Cursors work well for access patterns that are the composition of sequential access patterns.
- Benchmarking is hard, even for simple changes.



Obtaining Our Code

Daniel Ellard

ellard@eecs.harvard.edu

http://www.eecs.harvard.edu/~ellard/NFS