
# Fixing XFS Filesystems Faster

Dave Chinner <dgc@sgi.com>
Barry Naujok <bnaujok@sgi.com>

# Overview

- The “Repair Problem”
- The “First Attempt”
- An “Alternate Solution”
- Analysis of Failure and Success
- The “Final Design”
- Results
- Futures

# The “Repair Problem”

- Filesystem capacity grows faster than disk capabilities
- Number of objects indexed grows faster than the rate we can read them
- Repair reads every object in the filesystem
- Therefore, if repair doesn't get smarter, it will take longer as capacity grows
- 4 years ago a customer was very unhappy with `xfs_repair` taking 8 days to complete.

# What Does `xfs_repair` Do?

- Phase 1 – finds and validates primary metadata
- Phase 2 – reads in free space and inode locations
- Phase 3 – inode discovery and checking
- Phase 4 – extent discovery and checking
- Phase 5 – rebuild free space and inode indexes
- Phase 6 – check directory structure
- Phase 7 – check link counts

# The “First Attempt”

- Was aimed at improving `xfs_repair` on Irix
- No kernel block device caching in Irix
- Lots of relatively slow CPUs but with high I/O throughput
- Phases 3 and 4 scan each Allocation Group (AG) sequentially, but each AG is mostly self contained

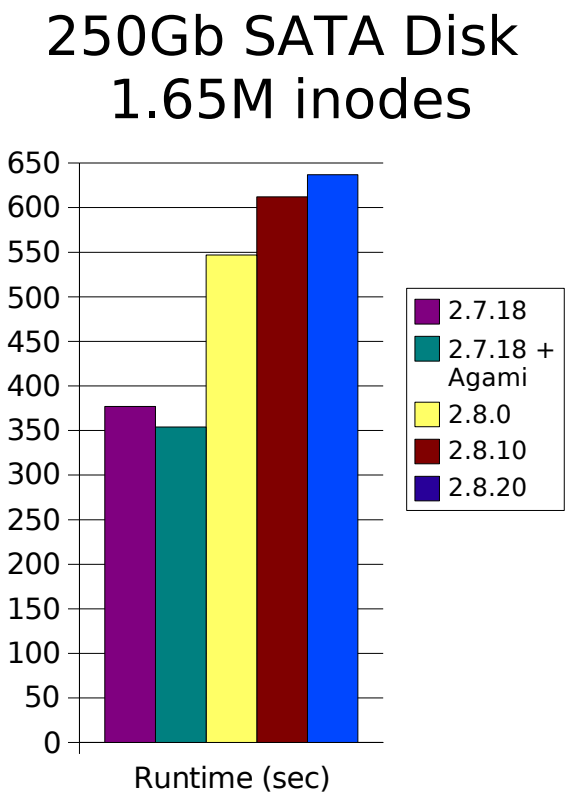
# The “First Attempt”, Part 2

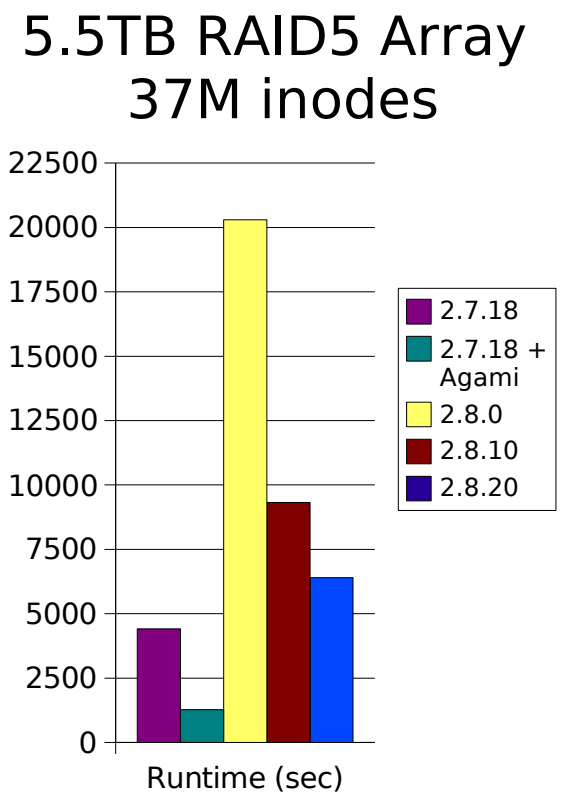
- Add hash-based block caching to xfs_repair
- Use a thread per AG and process multiple AGs at once
- Little I/O optimisation
  - mainly relying on multiple CPUs being able to issue I/O faster than a single process
  - some optimisation by batching synchronous readahead I/O
- Block based caching was released for Linux in version 2.8.0
- Multithreading was released in version 2.8.11

# An “Alternate Solution”

- Patch to 2.7.18 created by Agami Systems
- Used intelligent object based prefetch to prime the kernel buffer cache
- Processed inodes passed off to prefetch threads to read in associated metadata
- Processes only a single AG at a time
- Faster on a single disk than 2.7.18 until it ran out of memory
- Much faster than 2.7.18 on multi-disk arrays

# Success and Failure



# Analysis of Success and Failure

- We started comparison of 2.7.18 + Agami's patch against 2.7.18 and 2.8.20
- Surprise! In almost all cases, 2.8.x was **much** slower than 2.7.18.
- Block caching in `xfs_repair` was not working at all well on Linux
- Threading across AGs making it even worse.

# Analysis of Failure

- The optimisations for Irix focussed on CPU level parallelism
  - CPU bound not I/O bound
- Linux analysis was done on CPUs 2-3x faster and a smaller I/O subsystem
  - I/O bound, not CPU bound
- Adding more seeks into an already I/O bound setup makes it slower, not faster

# Analysis of Success

- The Agami patch used 10 threads to prefetch objects from a queue of 100, and adds 10 objects at a time to the prefetch queue
- Prefetch threads do no processing, only prime the kernel block device cache
- Processing thread feeds the prefetch queue as it processes objects it has read
- Speed up due to removing I/O latency in the processing thread.

# Rejecting Success!

- The Agami patch was superior to existing threading but we rejected it
- Not a cross-platform solution
  - needs to run on Irix and FreeBSD as well, which lack raw block device caching in the kernel
- Other technical reasons:
  - non-trivial porting effort to 2.8.x
  - Can not control cache usage or low memory readahead thrashing
  - Does not optimise I/O patterns at all

# Are We Crazy? (YES!)

- But we'd seen the light!
- Object based prefetch reduces I/O latency within an AG to speed up per-AG processing
- Per-AG parallelism allows saturation of larger, more complex storage configurations
- We could combine the two methods and go even faster!

# Further Analysis

- Further analysis on a single threaded repair:
  - Tracing exact order of  I/O from repair process
  - Identifying common patterns of metadata
    - often contiguous
    - lots of single blocks separated by small number of data blocks
  - identifying sub-optimal I/O patterns
    - backwards seeks
    - seeks across a large portion of the disk
- Looking for ways to sequentialise and reduce the number of I/Os the repair process issued.

# The “Final Solution”

- All patches included in `xfs_repair` version 2.9.4
- Added a pair of per-AG prefetch queues
  - one for blocks ahead of the current location
  - one for blocks behind current location
  - Second pass for “behind blocks” removing backwards seeks.
- Prefetch threads process the queue
  - identify contiguous blocks and metadata dense sparse ranges
  - issues single large I/O and throws away non-metadata blocks
  - uses bandwidth instead of seeks to read metadata blocks close together

# The “Final Solution”, Part 2

- Processing thread could stall on blocks in “behind queue”
  - prefetch threads switch queues if the primary block queue starts to run low
- Block cache needed work:
  - needed locking to be thread-safe
  - Different phases read metadata in different block sizes
    - Used to purge cache between phases and reread blocks
    - Made all I/O sizes the same -> no re-read between phases

# The “Final Solution”, Part 3

- Phase 6 – directory scanning was improved
  - now uses same inode scanning as Phase 3+4
  - visits each directory and inode counting links in a more I/O efficient manner
- Phase 7 – link count verification
  - needed another inode scan to record link counts in inodes
  - now recorded in Phase 3 and compared to calculated counts from Phase 6
  - only does I/O if they differ

# The “Final Solution”, Part 4

- Per-AG parallelism enhanced with “ag_stride”
  - avoids parallel processing of AGs on same disks
  - If phase 3 does not overflow the cache, phase 4 is fully parallelised without needing I/O
- Low memory behaviour optimised
  - cached blocks given priority based on:
    - how likely they are to be used again
    - how expensive they were to read in initially
  - low priority blocks purged first when cache overflows
  - reuse of free blocks to prevent heap fragmentation

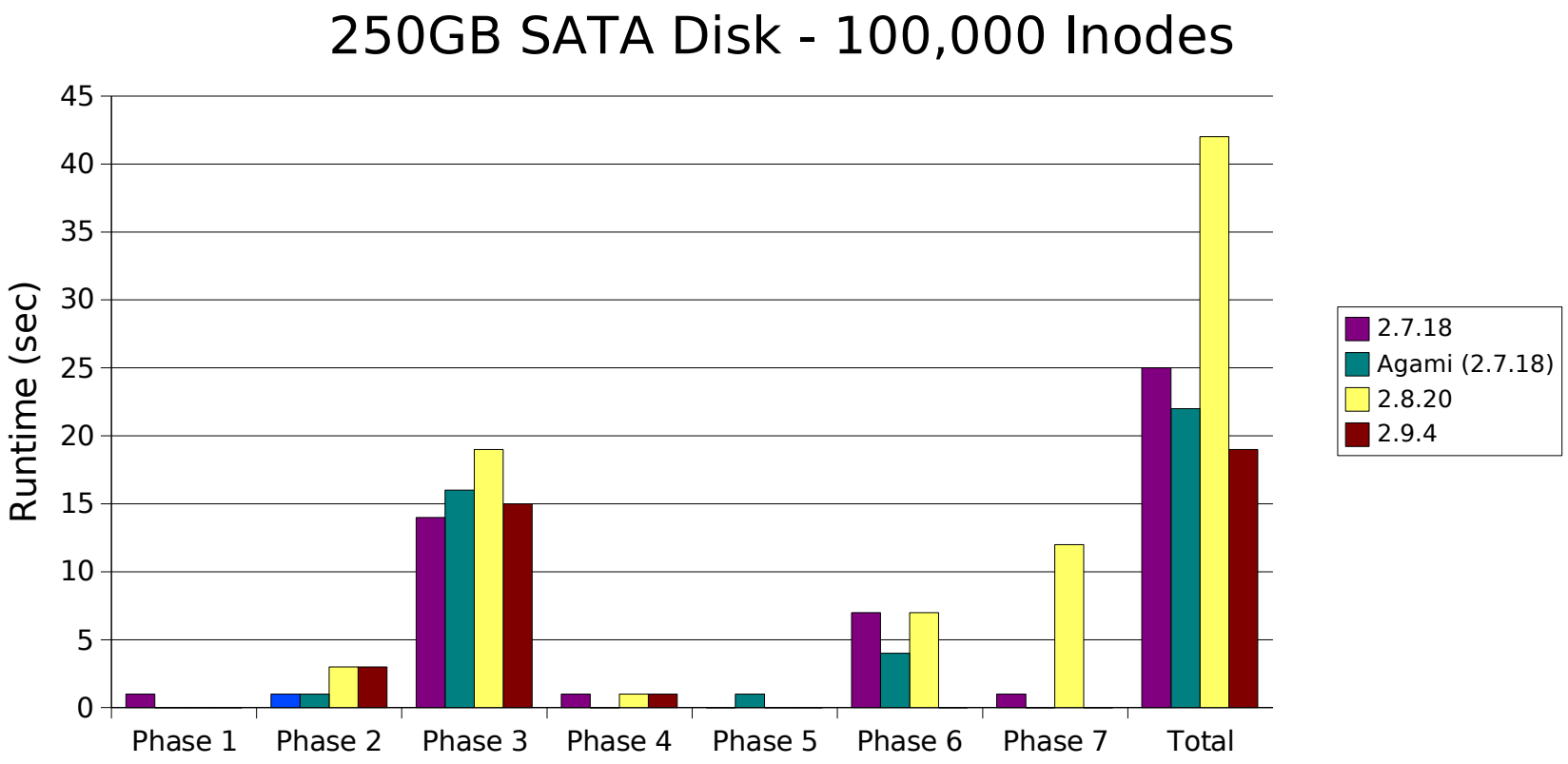
# Generating Test Filesystems

- Need to simulate aged filesystems
- Script runs at least 10 processes in parallel
- Each process
  - creates variable sized files at a varying directory depth
  - uses small direct I/Os to cause non-optimal allocation patterns
  - 10% probability of deleting a file instead of creating.
- Results in:
  - large and fragmented directory structures
  - physically separate inode chunks
  - Generates fragmented files and hence randomly varying inode extent lists

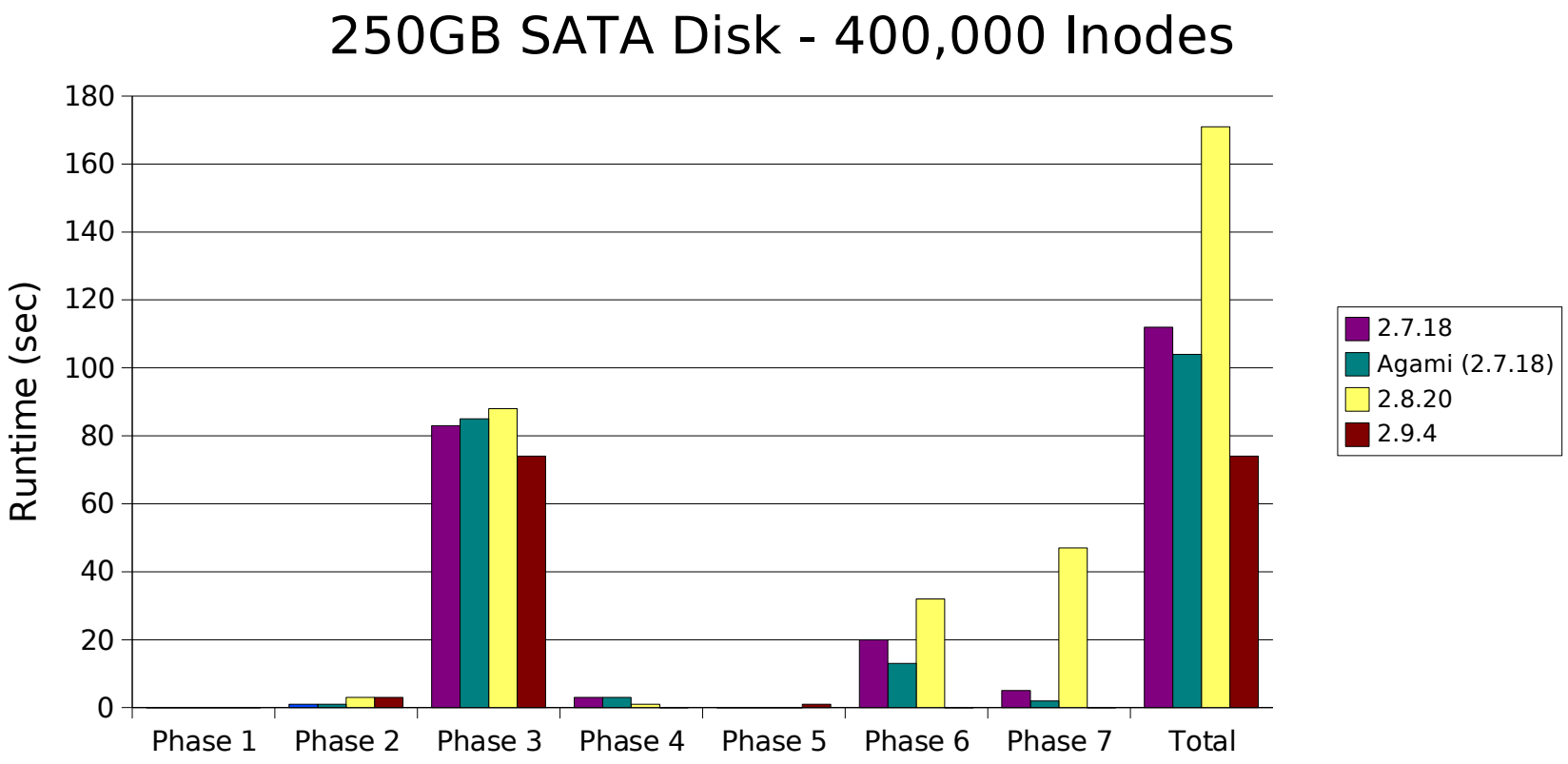
# The Results

- Test system #1 – Desktop/Workstation
  - dual processor x86_64, 2GB RAM, single 250GB SATA disk
    - 100,000 inodes, 7% full
    - 400,000 inodes, 100% full
    - 815,000 inodes, 100% full
    - 1.65M inodes, 100% full
    - 5.7M inodes, 100% full
    - 11M inodes, 37% full
    - 17M inodes, 100% full

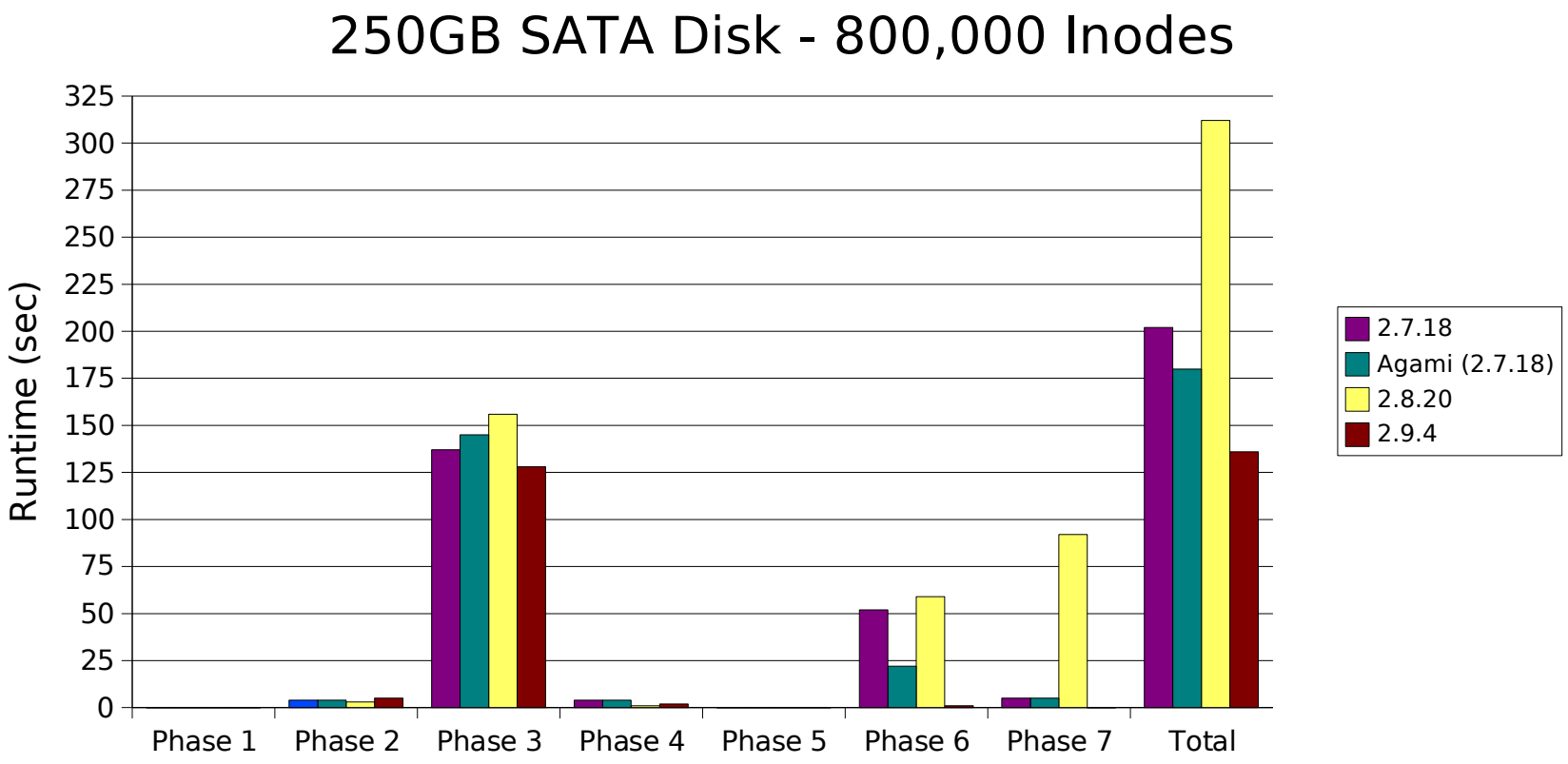
# 250GB SATA Disk - 100,000 Inodes



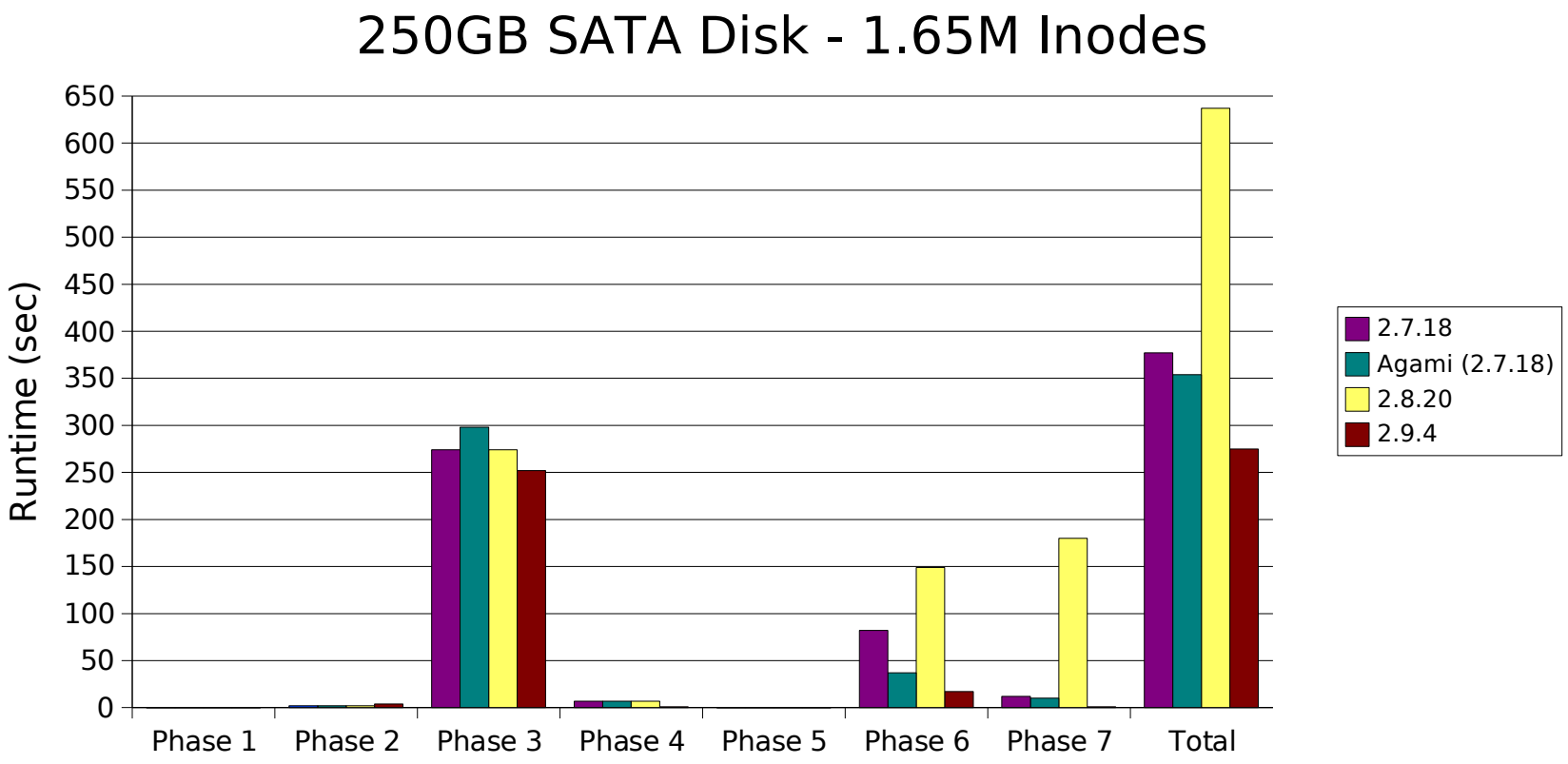
# 250GB SATA Disk - 400,000 Inodes



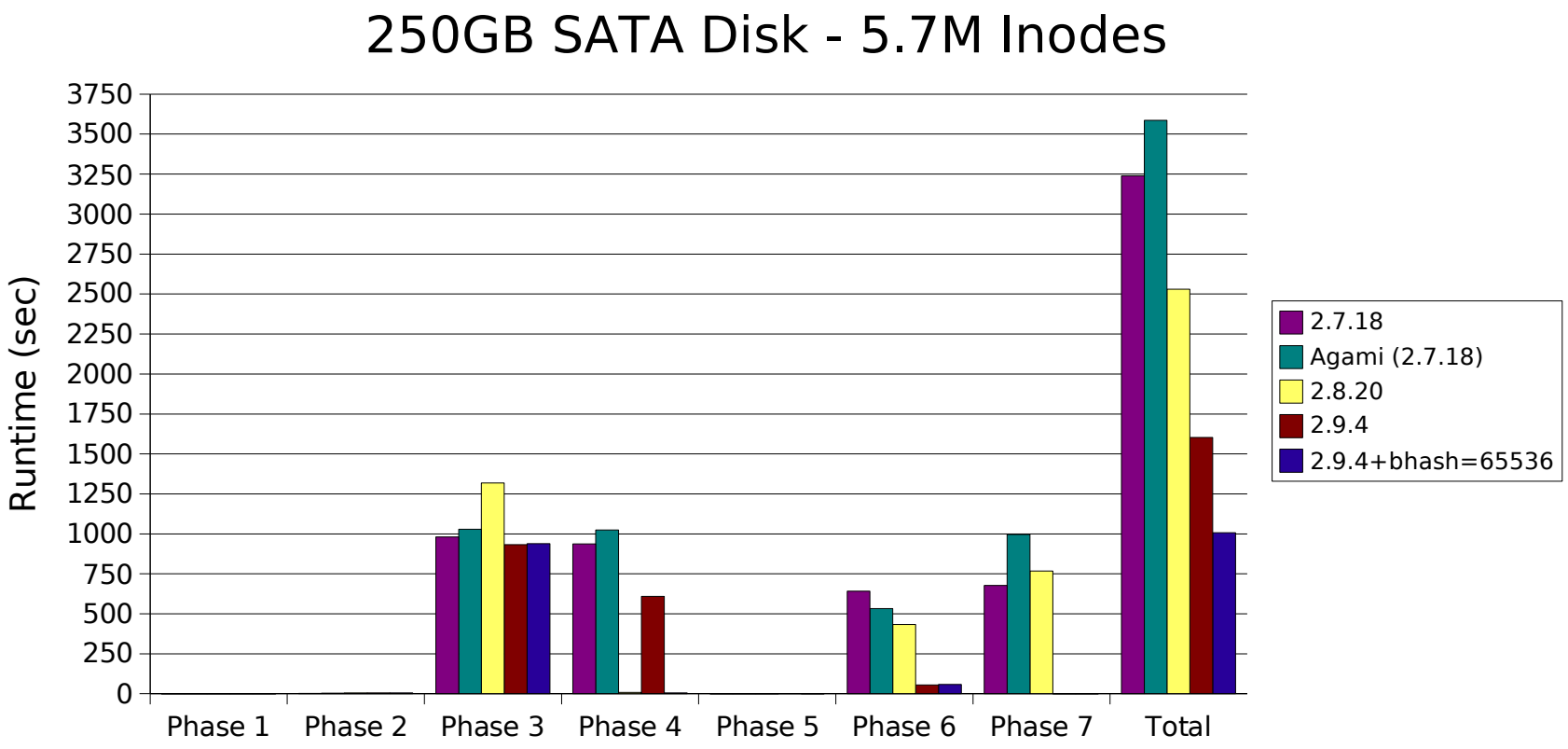
# 250GB SATA Disk - 800,000 Inodes



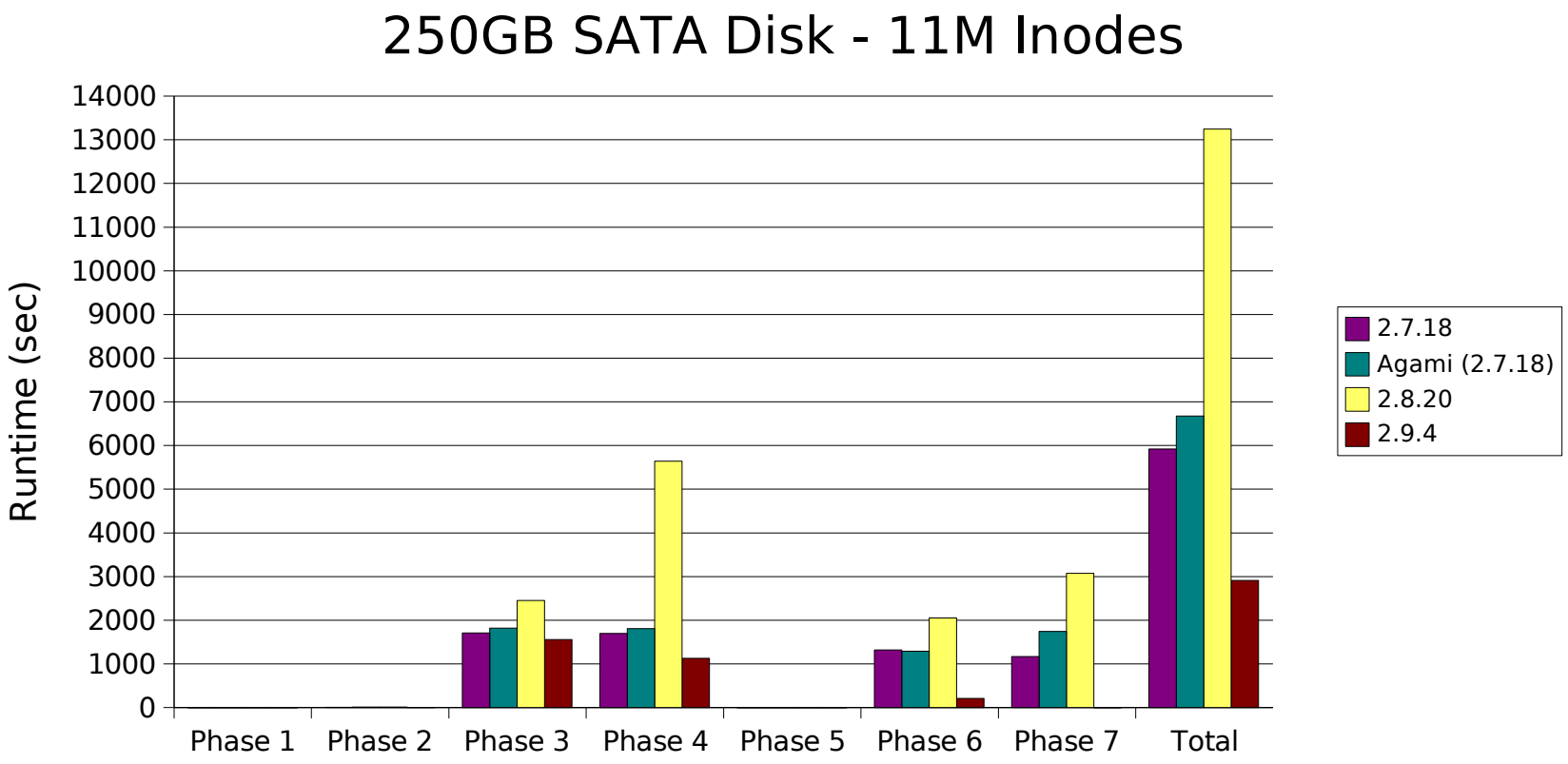
# 250GB SATA Disk – 1.65M Inodes



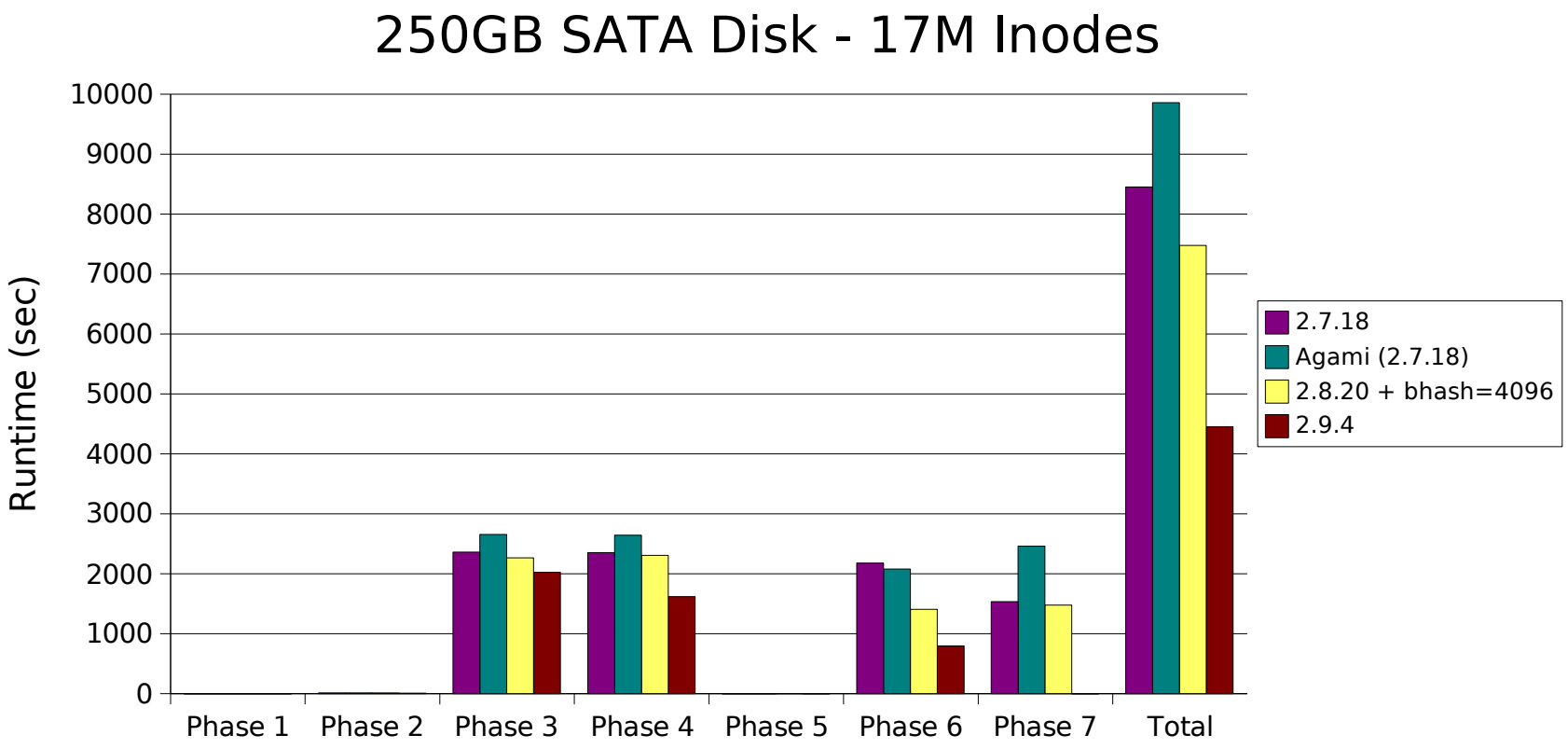
# 250GB SATA Disk - 5.7M Inodes



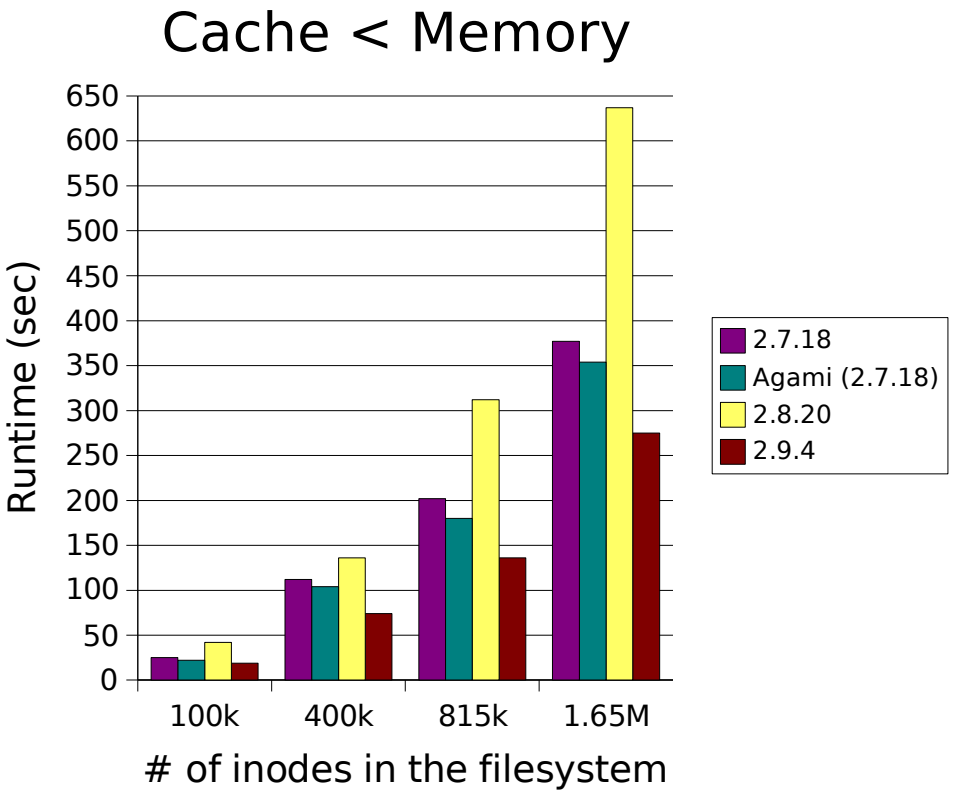
# 250GB SATA Disk - 11M Inodes

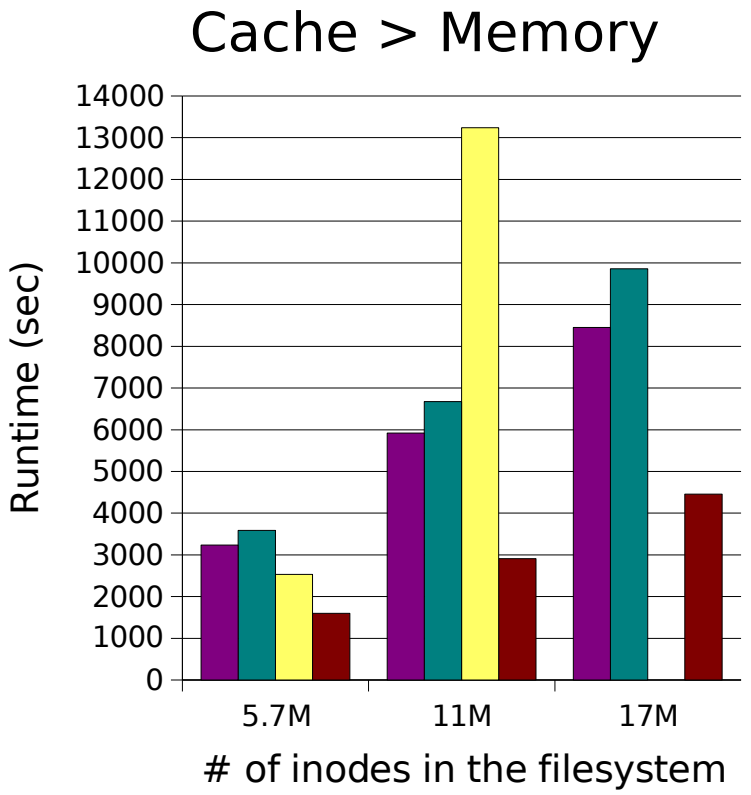


# 250GB SATA Disk - 17M Inodes

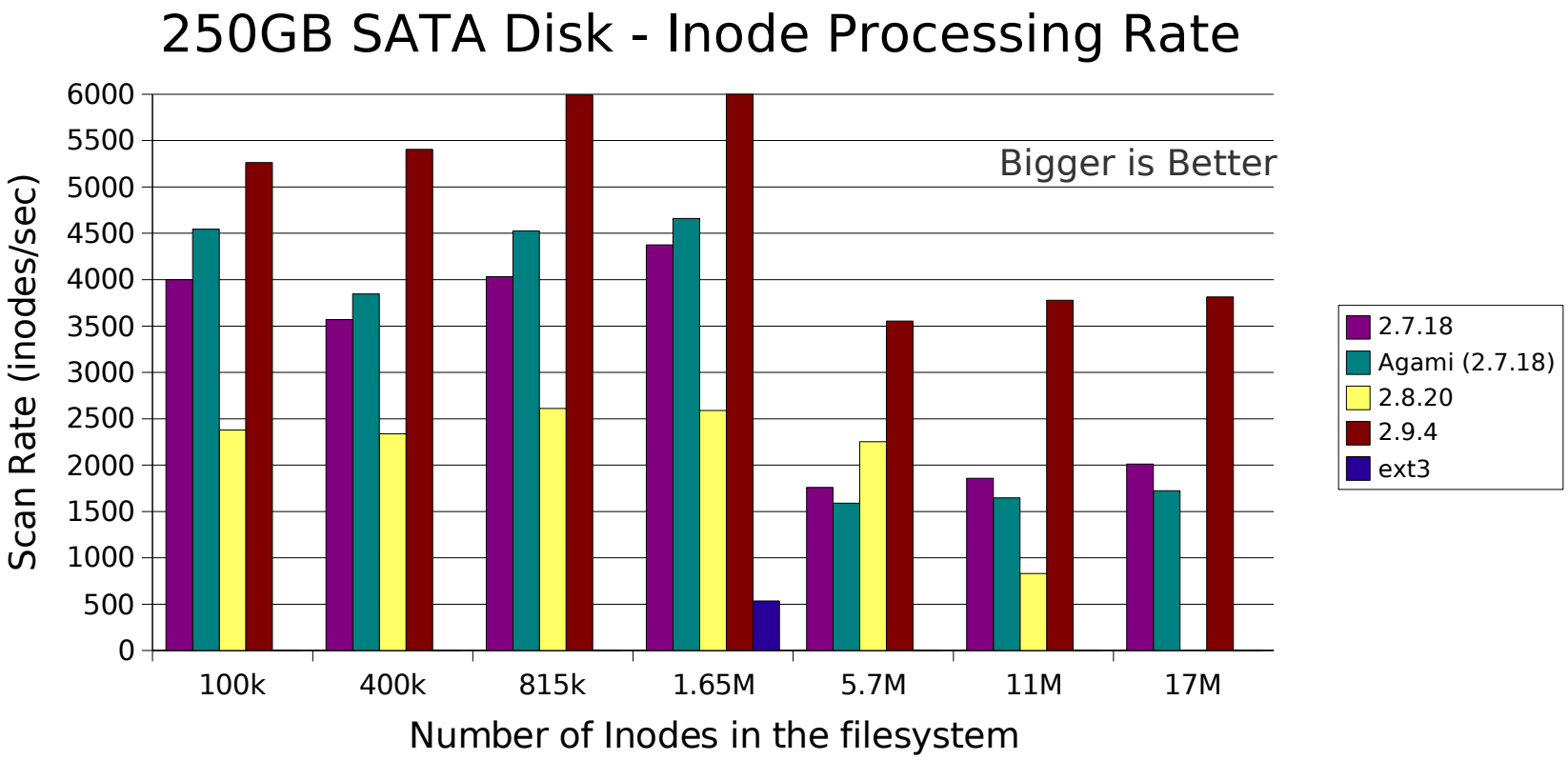


# 250GB SATA Disk – Runtime Scaling



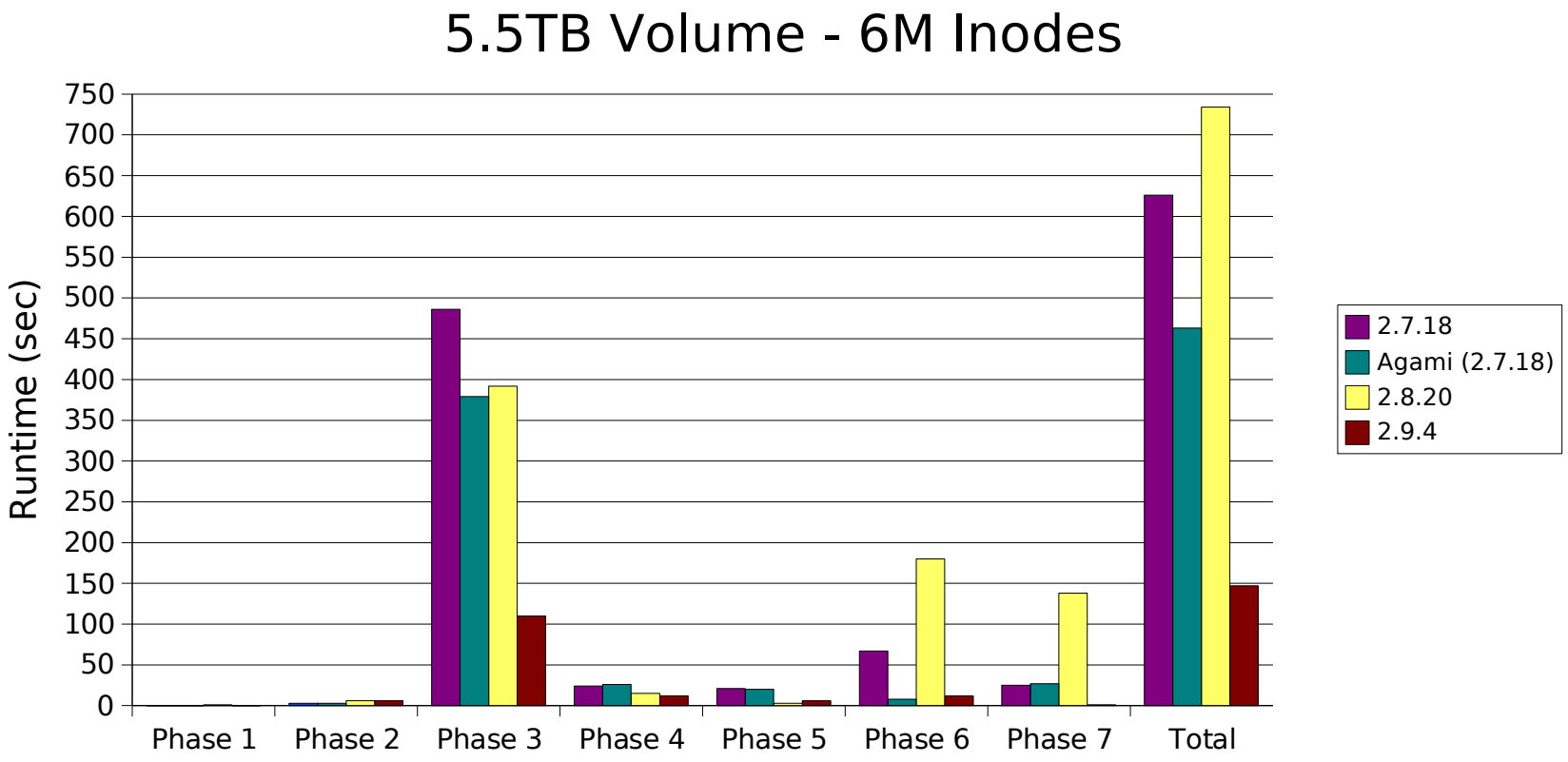
# 250GB SATA Disk – Inode Processing Rate



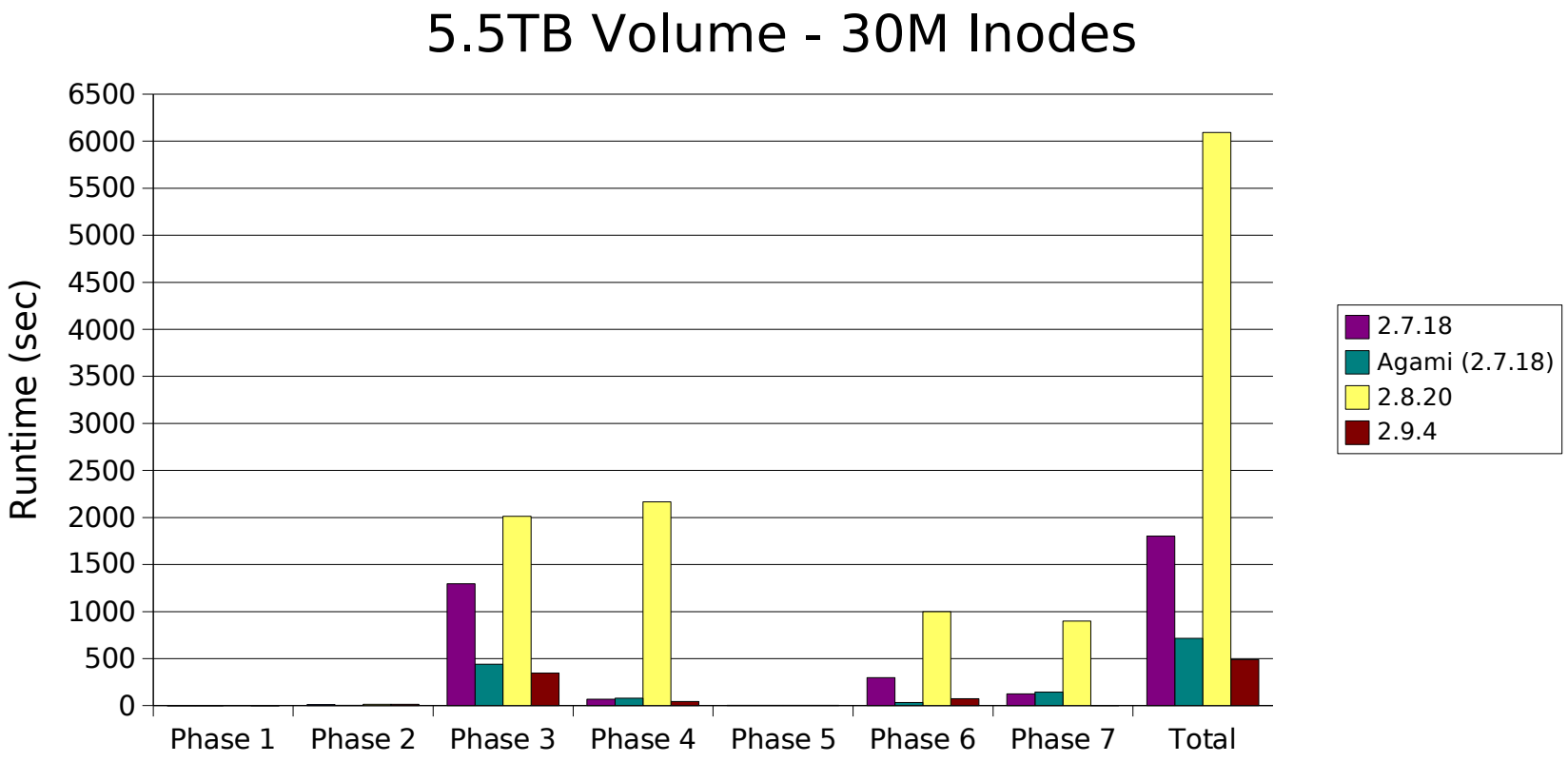
# More Results

- Test system #2 – large server
  - 4p ia64, 48GB RAM:
    - 5-way RAID0 stripe of 4+1 hardware RAID5 luns, 5.5TB capacity
      - 6M inodes, 80% full
      - 30M inodes, 100% full
      - 300M inodes, 60% full

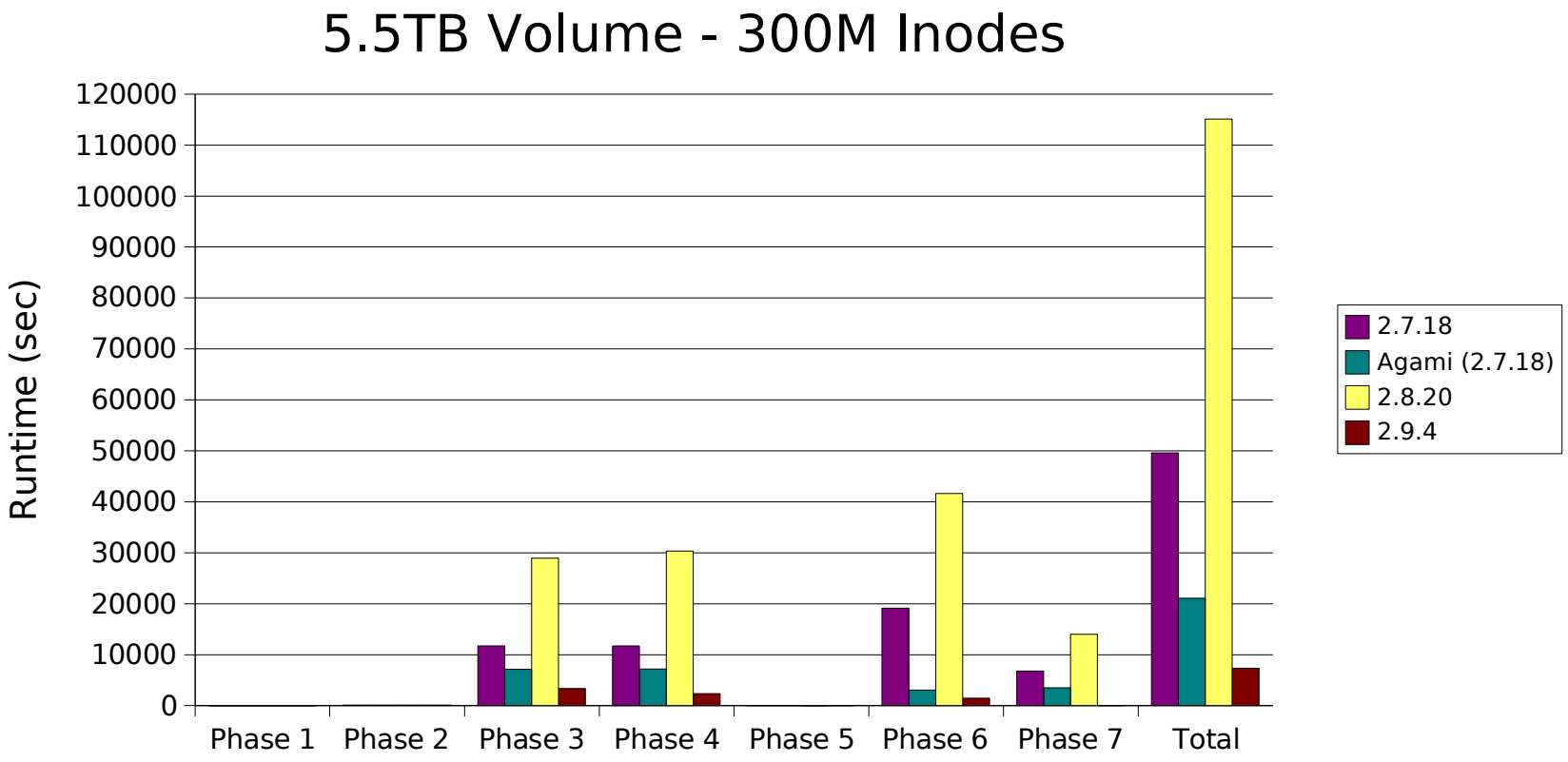
# 5.5TB Volume - 6M Inodes



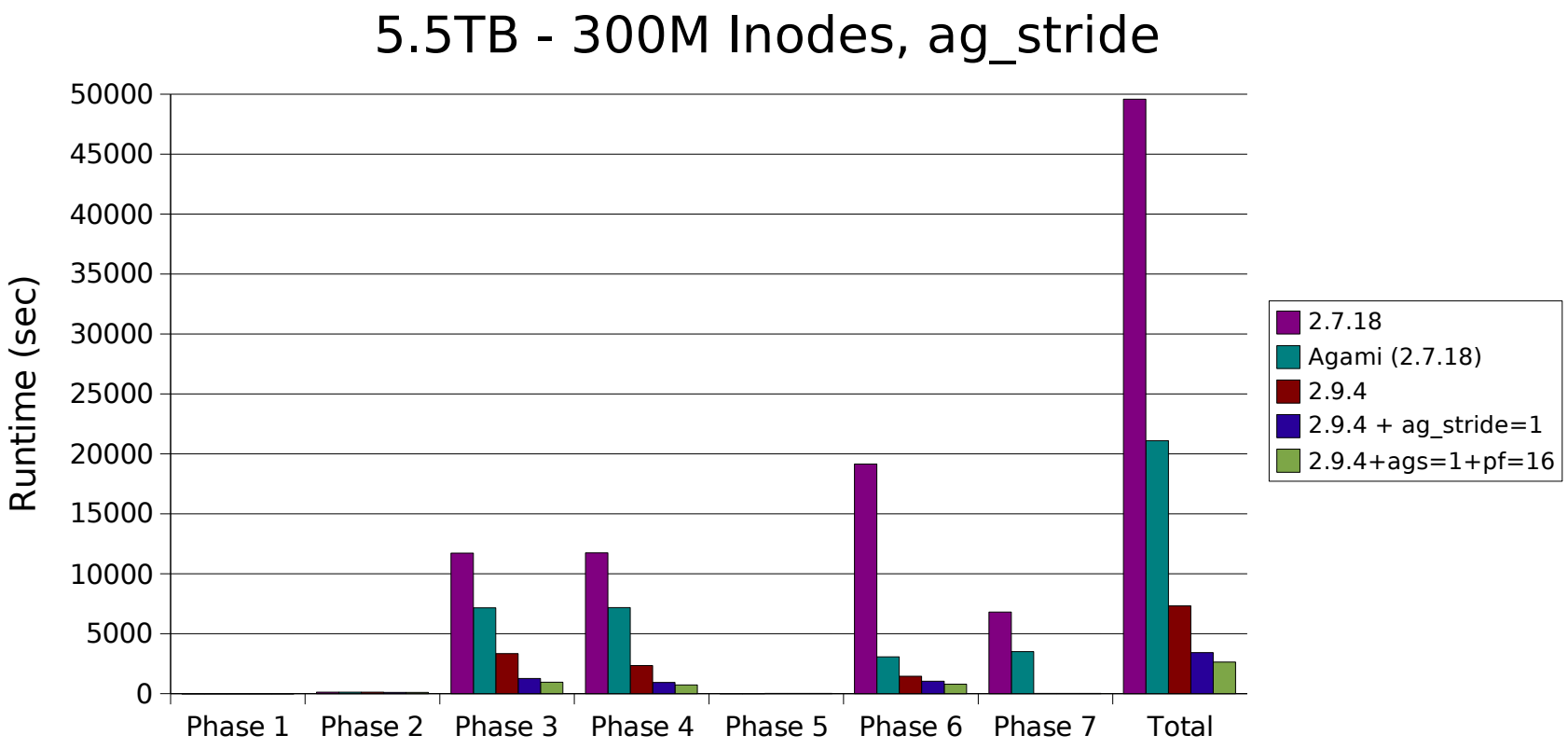
# 5.5TB Volume - 30M Inodes



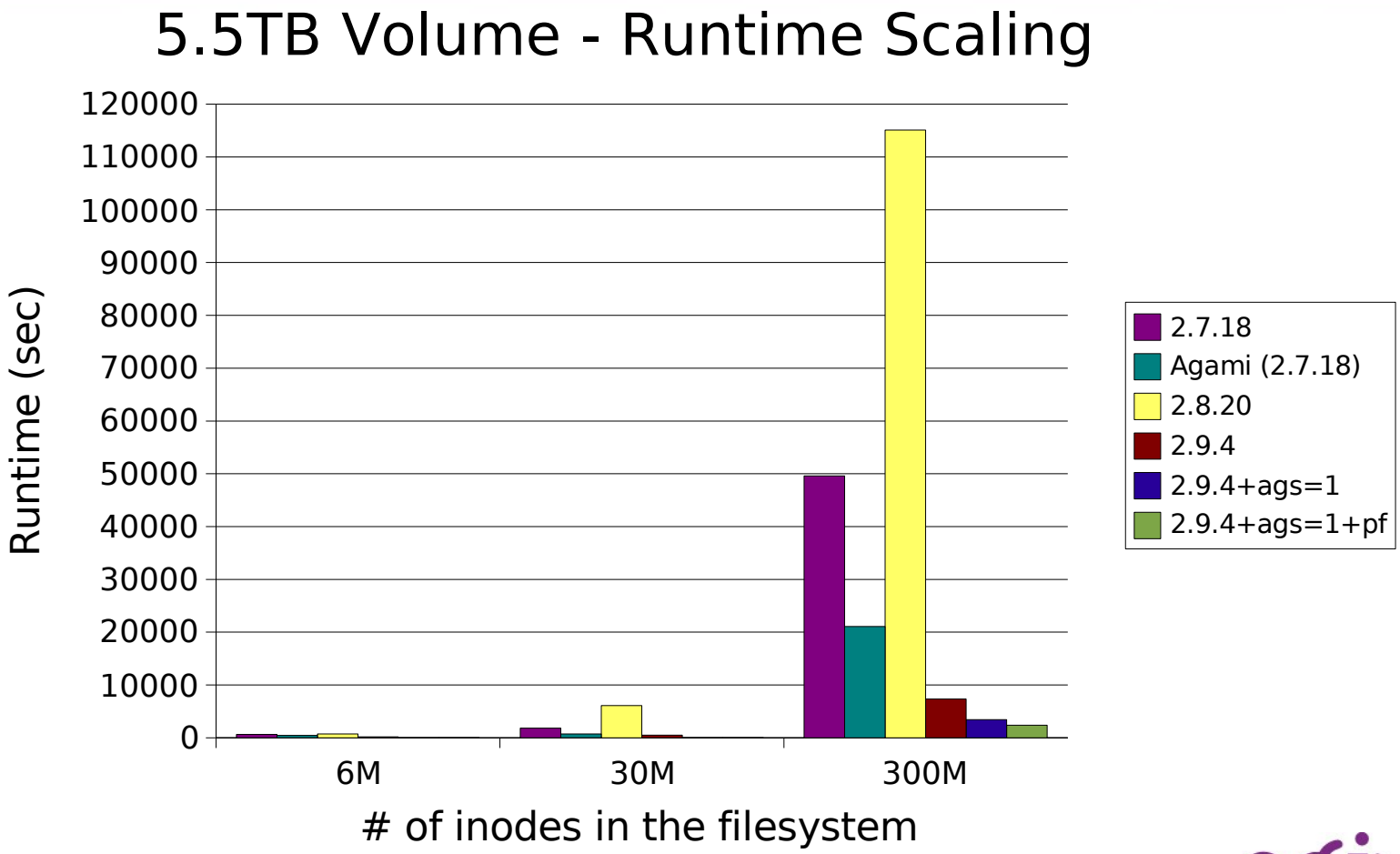
# 5.5TB Volume - 300M Inodes



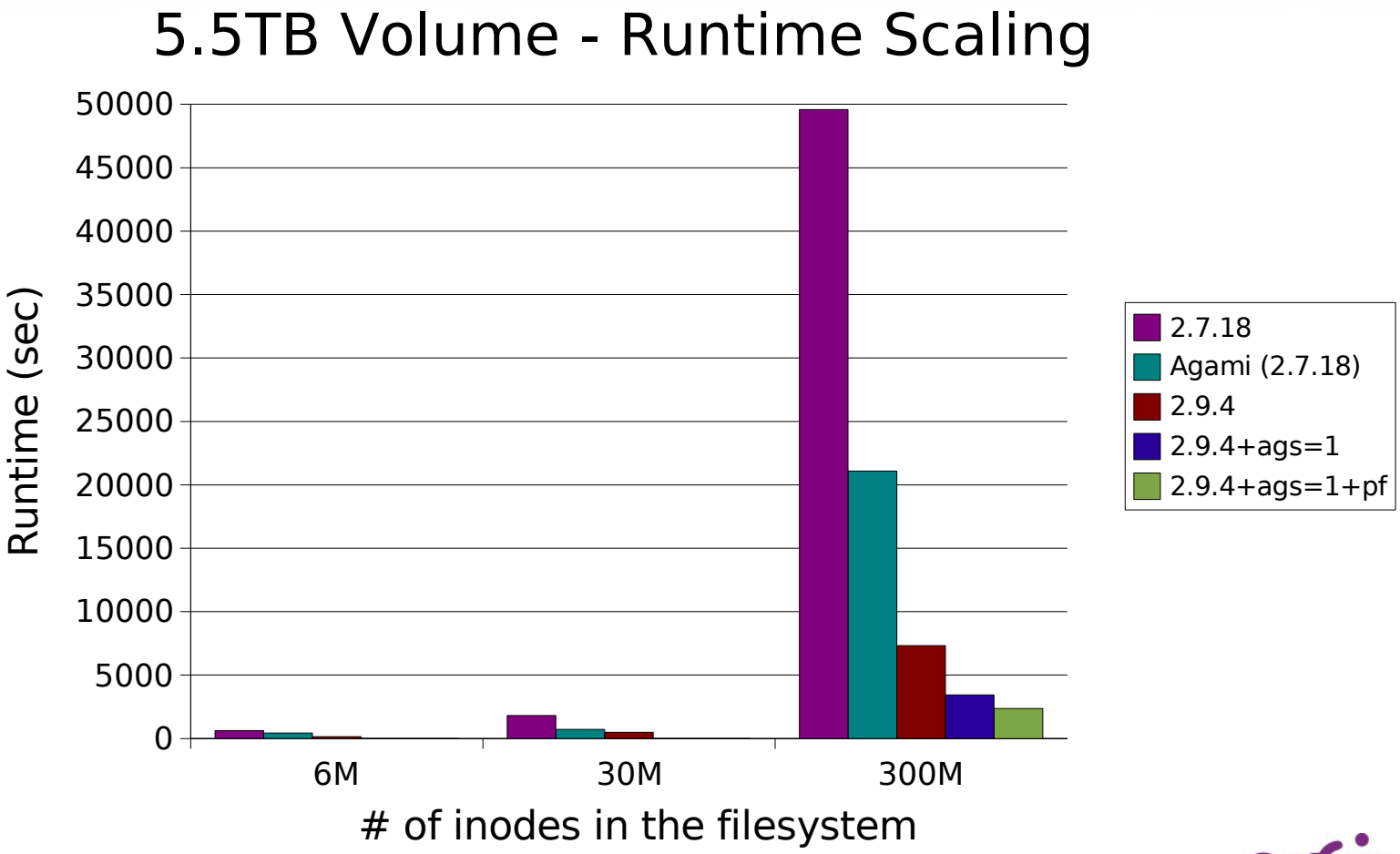
# 5.5TB - 300M Inodes, Part 2



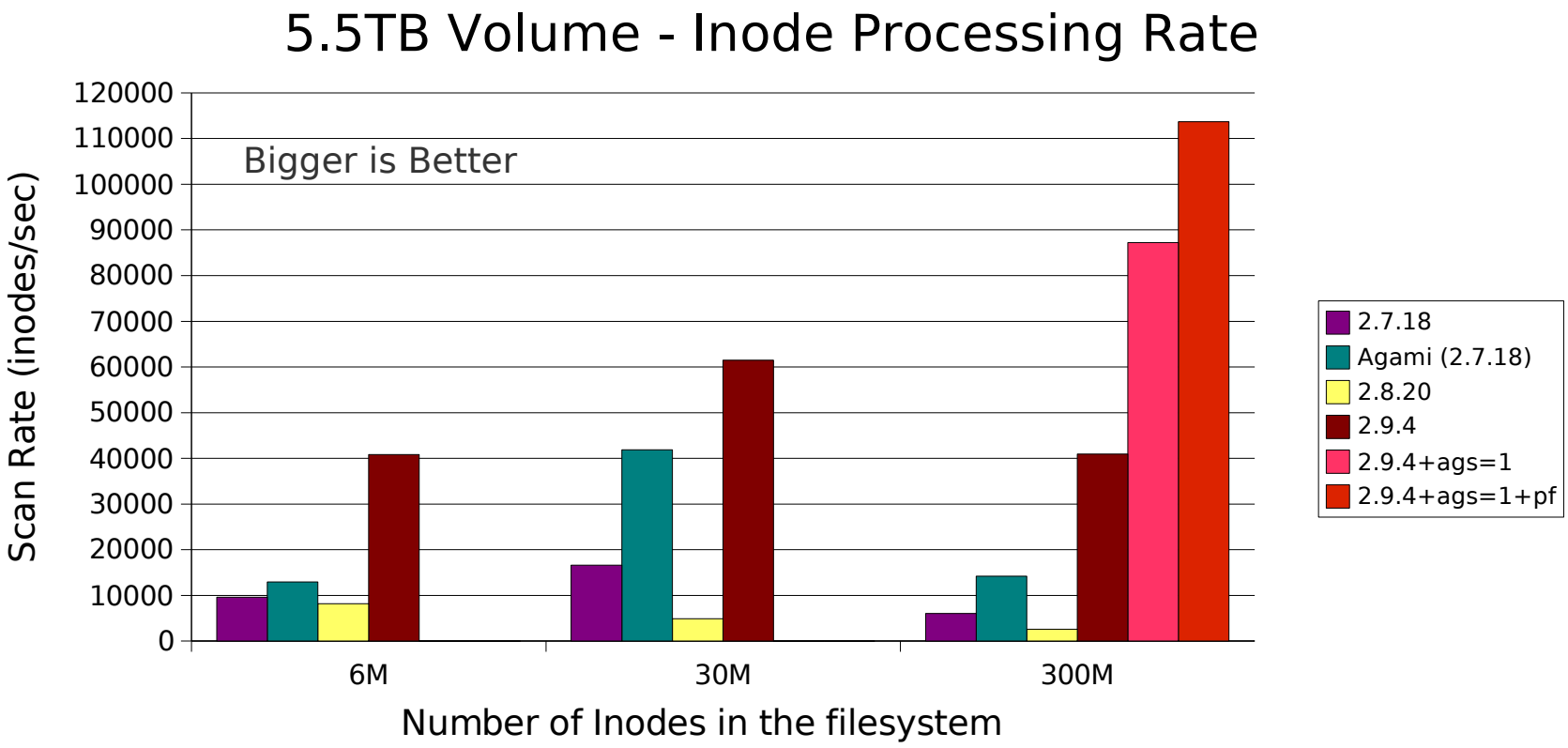
# 5.5TB Volume – Runtime Scaling



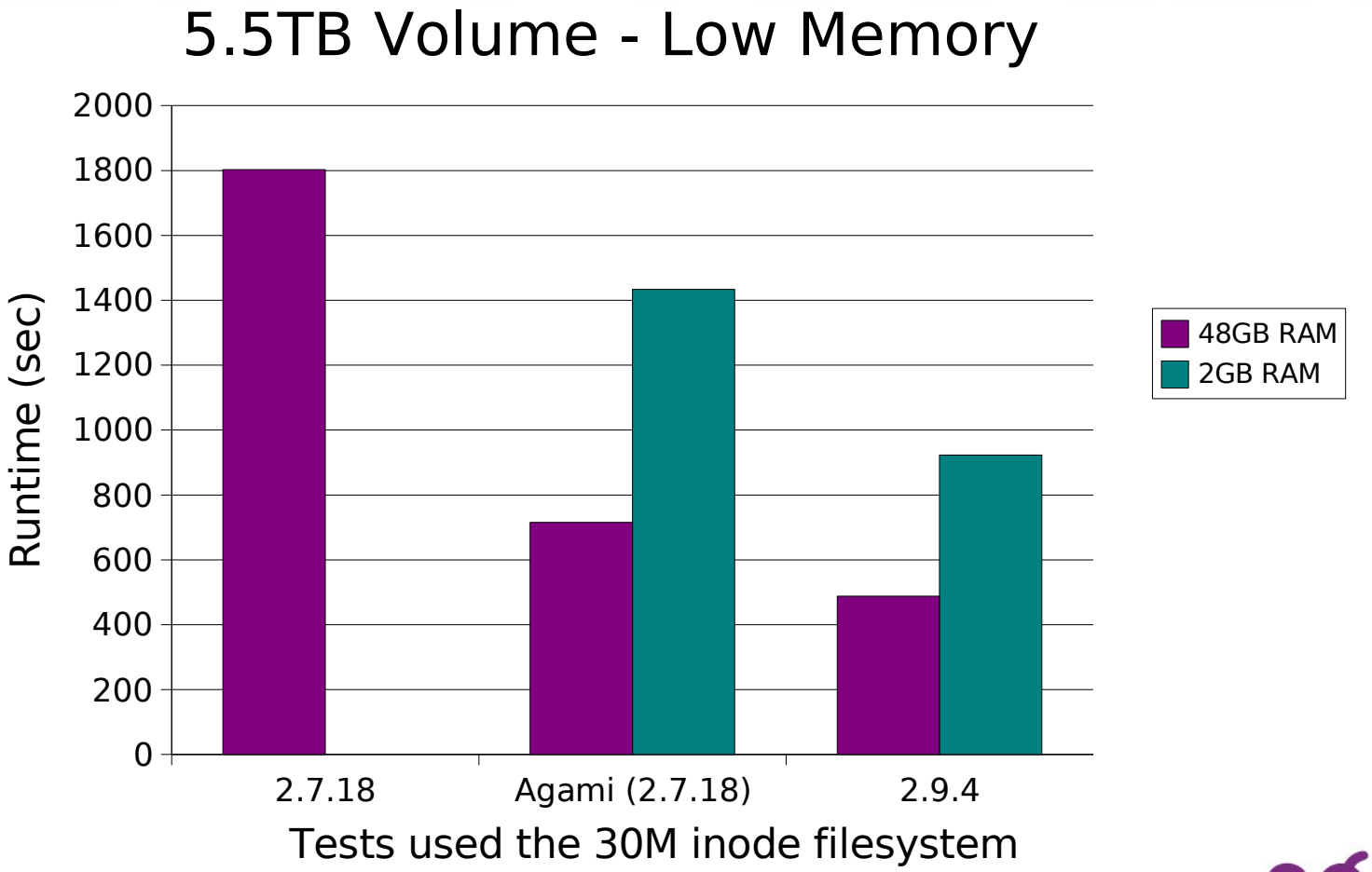
# 5.5TB Volume – Runtime Scaling



# 5.5TB Volume – Inode Processing Rate



# 5.5TB Volume – Low Memory



# Futures

- Memory usage reductions
  - allow larger filesystems to be checked in small RAM configs
  - Introduce more efficient indexing structures
  - Use extents for indexing free space
- Performance
  - Multithreading of Phase 6
  - Directory name hash checking scalability
  - Trade memory usage savings for larger caches
- Robustness
  - Phase 1 on badly broken filesystems
  - Preservation of broken directories

# Questions?