The Server Push Architecture for High End Computing

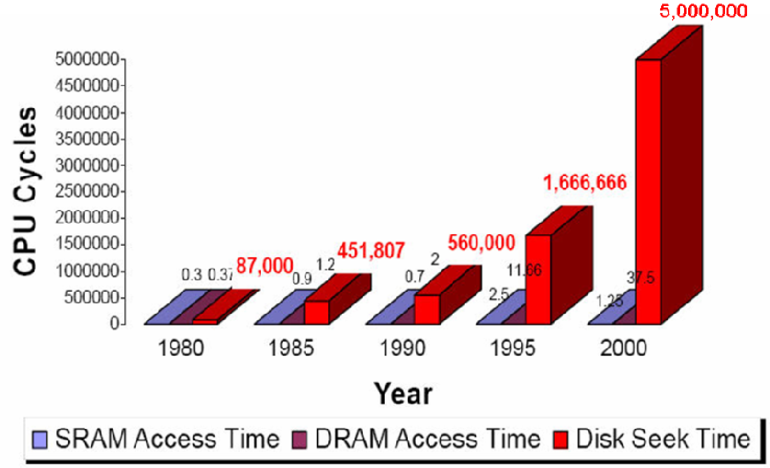
Xian-He Sun

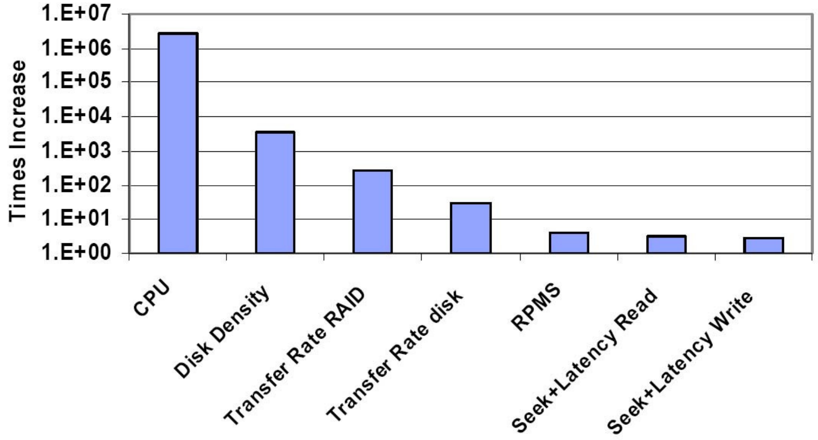
Rajeev Thakur, William Gropp

Illinois Institute of Technology
Argonne National Laboratory
sun@iit.edu

THE PROBLEM: I/O BOTTLENECK

- Poor Parallel I/O performance for complex non-contiguous (small) access
- Improving the performance of large number of small I/O requests is a necessity
- Prefetching – fetch data before a client demands for it
- Limitations of Existing Prefetching
  - Conservative and limited to static prediction strategies
  - Only works for simple access patterns with locality



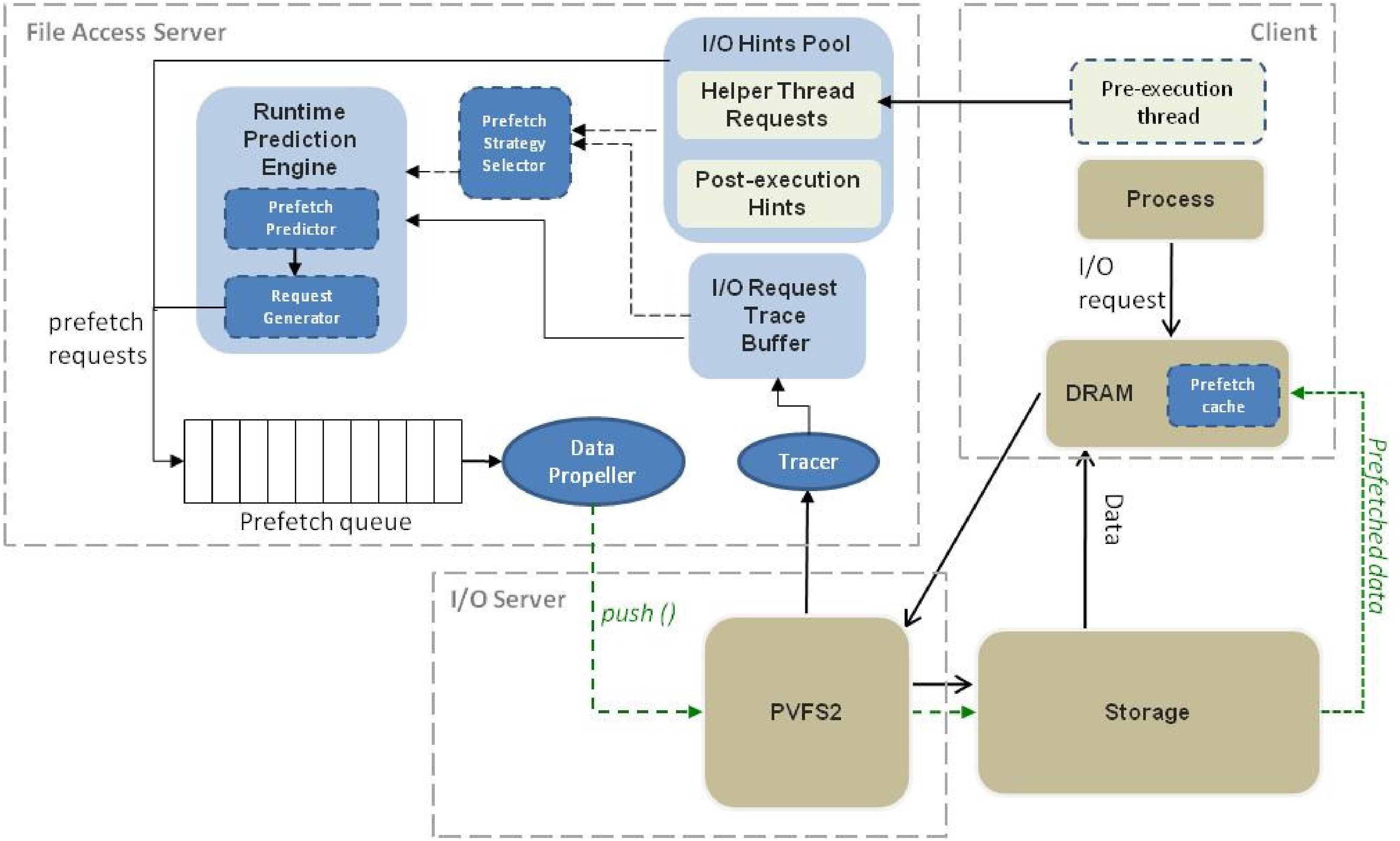


OUR SOLUTION: FILE ACCESS SERVER (FAS)

- A server pro-actively "pushes" required data in time
  - Push: data is sent before the client's I/O request
  - In time: data arrives the destination within a window of time
- Use of adaptive and advanced prediction algorithms
  - Selects I/O access prediction algorithms adaptively
- Prefetch Engine
  - What to prefetch
  - When to prefetch
- Pushing data
  - Server issues prefetch instructions
  - Pushes the data from disk to prefetch cache at client
- **Push Server:** Parallel processing for prefetching

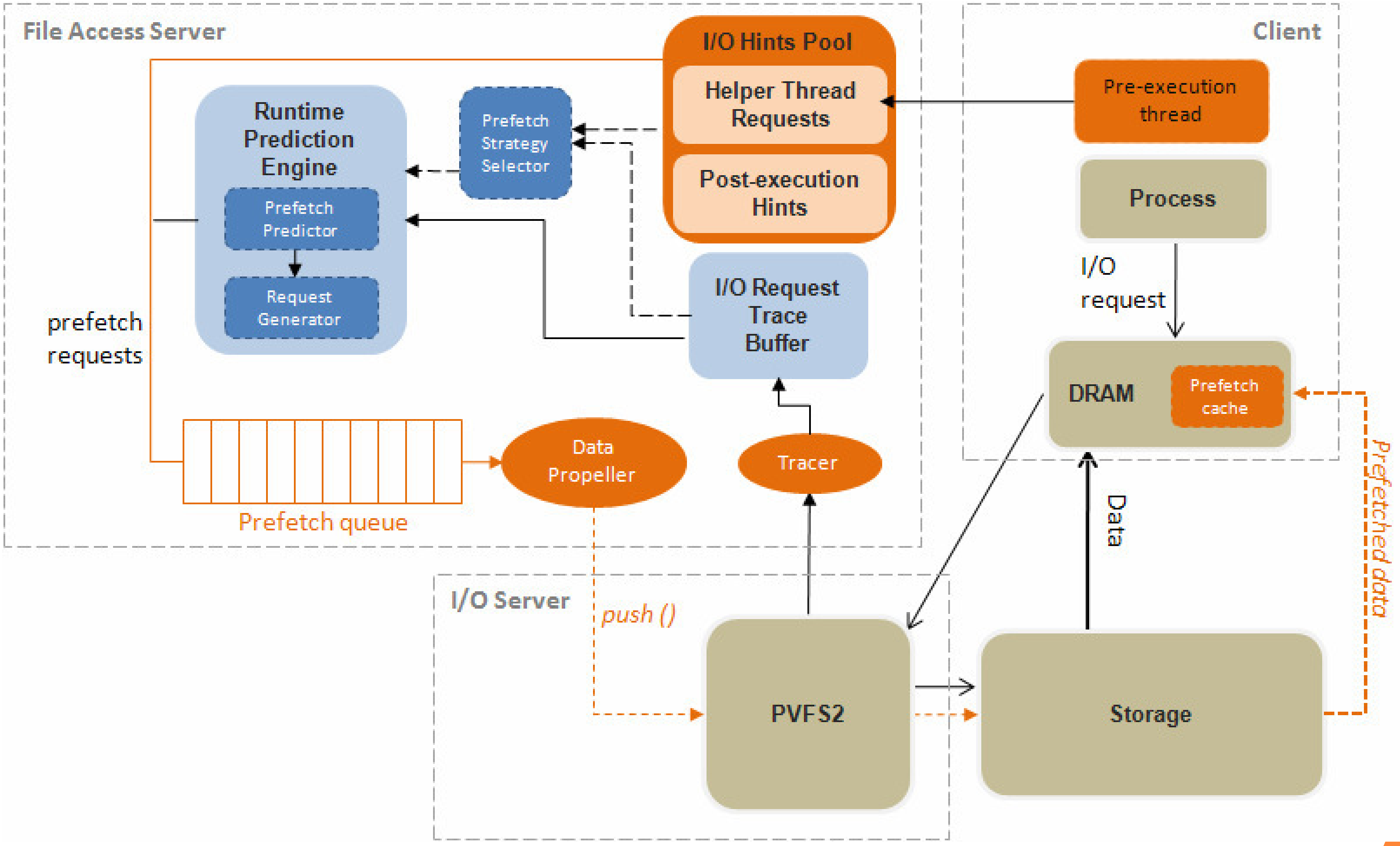
FAS ENABLED PARALLEL I/O

- File Access Server initiates prefetching requests
- Collect hints to predict future I/O needs
- Push data from disk to compute nodes

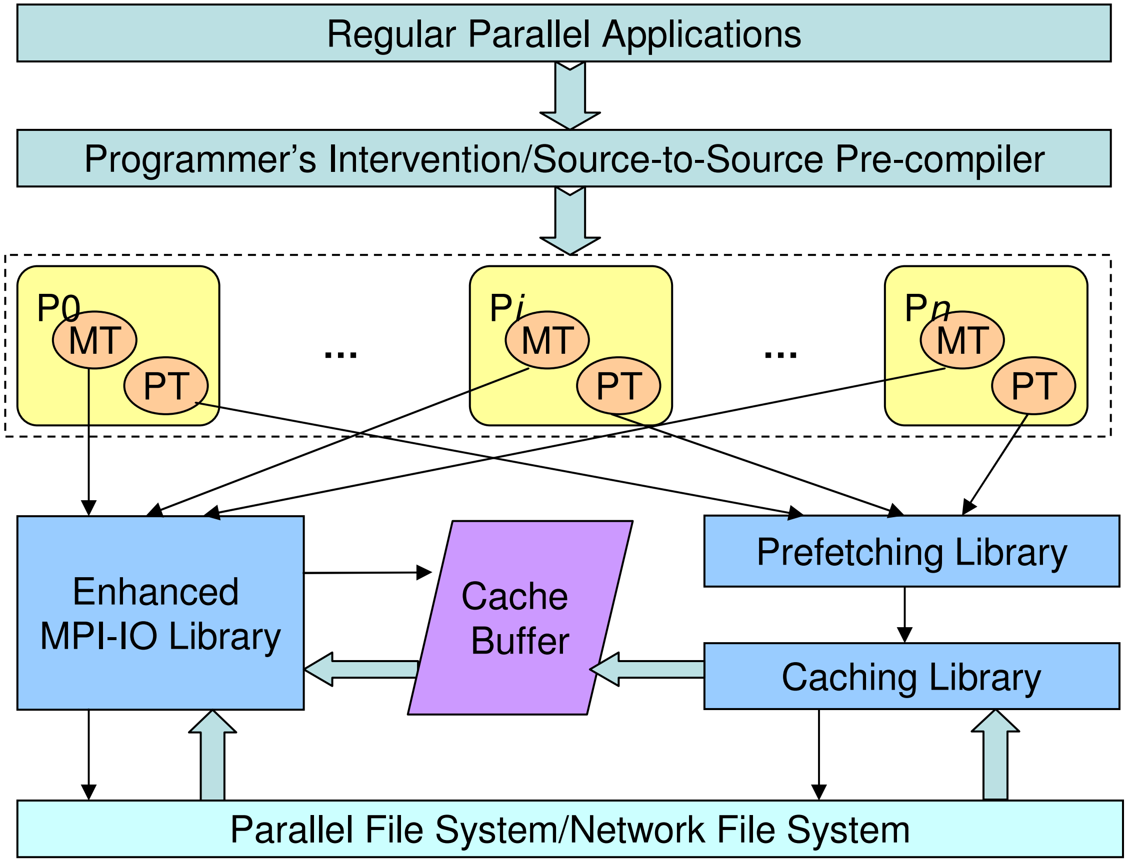


08/05/2008

CURRENT IMPLEMENTATION PROGRESS



PREFETCHING: Implementation under MPI



Regular Parallel Applications

Programmer's Intervention/Source-to-Source Pre-compiler

Enhanced MPI-IO Library

Cache Buffer

Prefetching Library

Caching Library

Parallel File System/Network File System

MT: main thread (computation thread)
PT: (pre-execution) prefetching thread

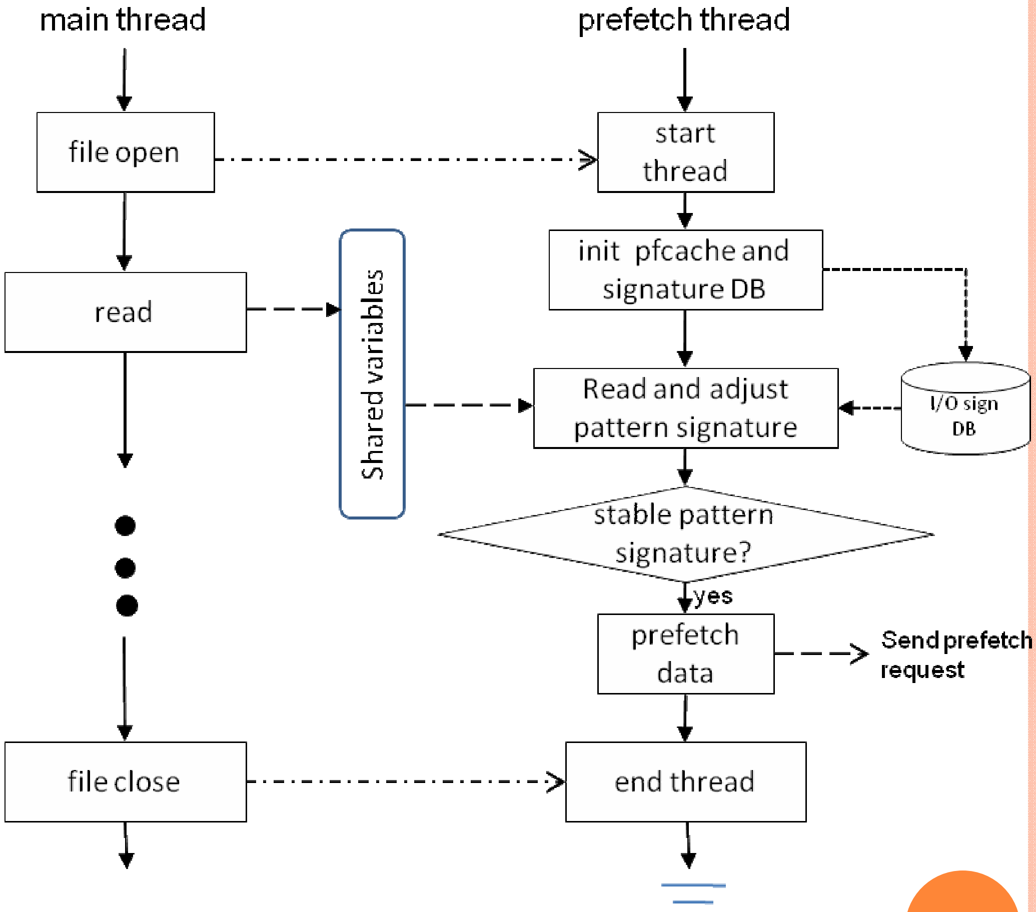
→ Request
⇨ Data

Yearly Major Achievements:

- A better understanding of I/O data access. Implemented prefetchings in MPI-IO ADIO layer.
- Implementation of signature based prefetching
  - A hybrid of offline analysis and runtime adaptation for predicting future I/O accesses
  - Development of Signature Notation for I/O Workloads (characterization & adaptation)
  - Publication in SC 2008
- Implementation of pre-execution based prefetching
  - A pre-execution thread predicts future I/O accesses and initiates prefetching data
  - Publication in SC 2008 (**best paper** nomination)
- Non-conventional approaches, many new research issues

Post Analysis: IO Signature-based Prefetching

- Generate IO signature (post analysis)
- Prefetching thread initiates prefetch cache and reads signature, picks prefetching scheme
- Adjusts signature based on running process and current pattern information



Challenges

- Identify, represent, and detect of I/O signatures
  - Characterize I/O access
  - Patterns and Notation, Trace Signature, Pattern Signature
- Collecting runtime information to adjust signatures
  - PT initiates a prefetch cache, share I/O read accesses info of the main thread
- Maintaining coherence when data is updated
- Support for prefetching thread (PT) and prefetch cache
- Prefetching library to separate from I/O accesses

I/O Signature: Patterns and Notation

- Comprehensive I/O access pattern classification

Spatial Patterns

- ☐ Contiguous
- ☐ Non-contiguous
  - Fixed strided
  - 2d-strided
  - Negative strided
  - Random strided
  - *k*d-strided
- ☐ Combination of contiguous and non-contiguous patterns

Request size

- ☐ Fixed
- ☐ Variable

- ☐ Small
- ☐ Medium
- ☐ Large

Temporal Intervals

- ☐ Fixed
- ☐ Random

Repetition

- ☐ Single occurrence
- ☐ Repeating

I/O Operation

- ☐ Read only
- ☐ Write only
- ☐ Read/write

Trace Signature

- Description of a sequence of I/O accesses in a pattern
- Form: *{I/O operation, init position, dimension, ([{offset pattern}, {request size pattern}, {pattern of number of repetitions}], […]), # of repetitions}*

Pattern Signature

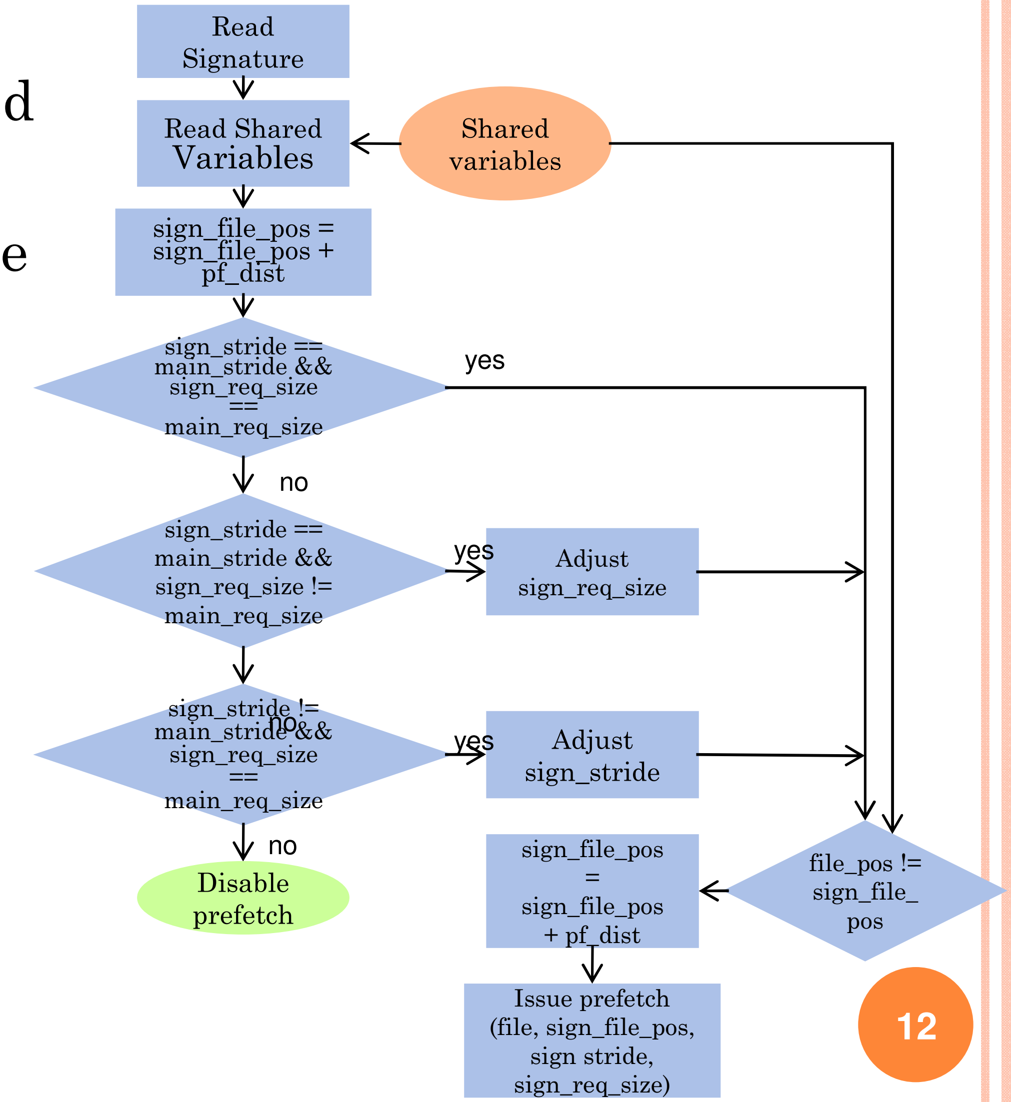
- provides a simple description that explains the nature of a pattern
- Form: *{I/O operation, <Spatial pattern, Dimension>, <Repetitive behavior>, <Request size>, <Temporal Intervals>}*

Pattern Detection

- Developed a pattern detection tool
- Five pattern detectors for finding patterns among initial positions, offsets, request sizes, temporality, and repetitions
- Outputs I/O Signature that can be used for prefetching

ADJUSTING I/O SIGNATURES

- MT comm. to PT via shared variables
- SVs include file handle, file location, request size & protected by a POSIX mutex
- PT verifies the signature via SVs generated by the MT (prefetch distance)
  - Confirmed: update file location
  - Not: does not prefetch



Read Signature

Read Shared Variables

Shared variables

sign_file_pos = sign_file_pos + pf_dist

sign_stride == main_stride && sign_req_size == main_req_size

yes

no

sign_stride == main_stride && sign_req_size != main_req_size

yes

Adjust sign_req_size

sign_stride != main_stride && sign_req_size == main_req_size

no

yes

Adjust sign_stride

no

Disable prefetch

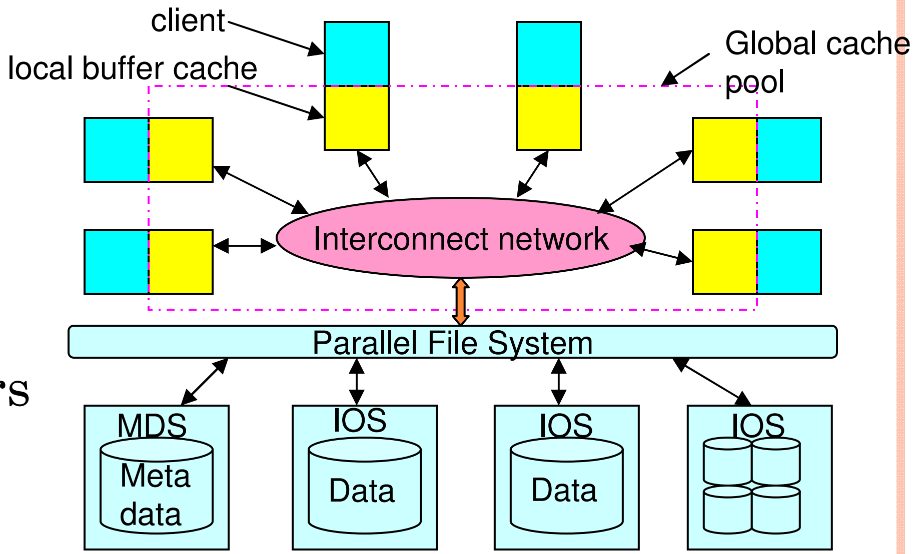
file_pos != sign_file_pos

sign_file_pos = sign_file_pos + pf_dist

Issue prefetch (file, sign_file_pos, sign stride, sign_req_size)

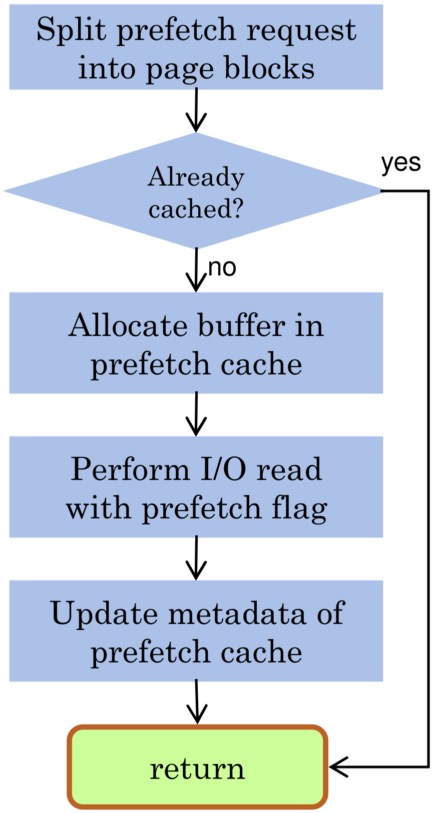
CACHING LIBRARY SUPPORT

- Collective caching
  - W.K. Liao et al., Northwestern
  - Global cache pool comprised by cache buffer from all clients
  - Coordinate to manage cache data w/o involving I/O servers
  - Avoid coherence problem by keeping at most one copy
- Our customization
  - Enable read caching only
  - Direct caching policy with prefetching result



Collective Caching
from Northwestern Research Group

PREFETCHING LIBRARY AND MAINTAINING COHERENCE



Split prefetch request into page blocks

Already cached?

yes

no

Allocate buffer in prefetch cache

Perform I/O read with prefetch flag

Update metadata of prefetch cache

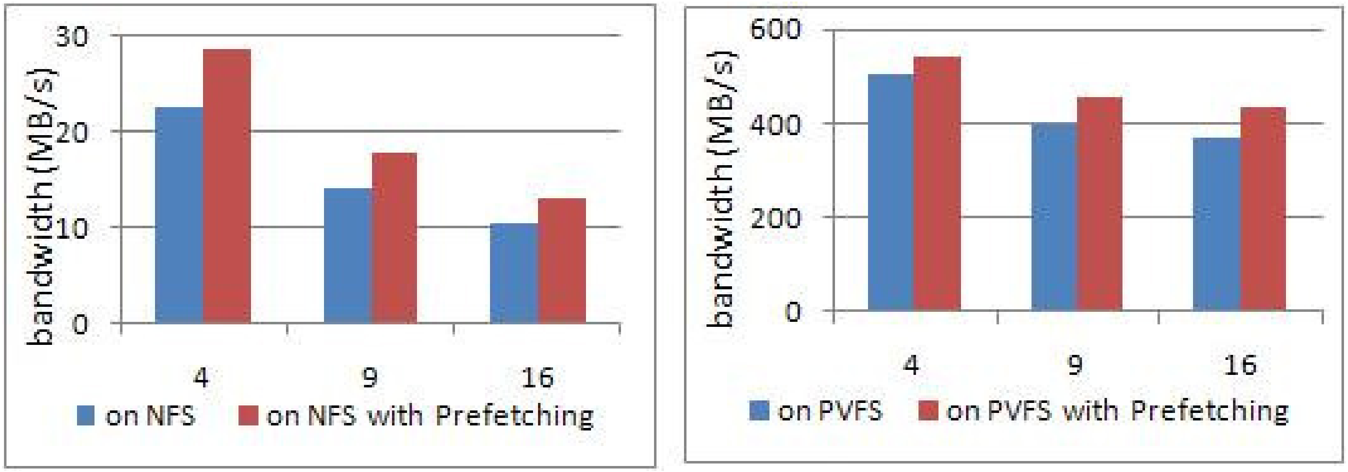
return

- Prefetching library verifies if data is already cached
- If not, disk read is issued

- Coherence is maintained by invalidation
- MPI-IO write operation is modified
- MPI-IO Write looks up prefetch cache and invalidates the page, if found

PERFORMANCE RESULTS

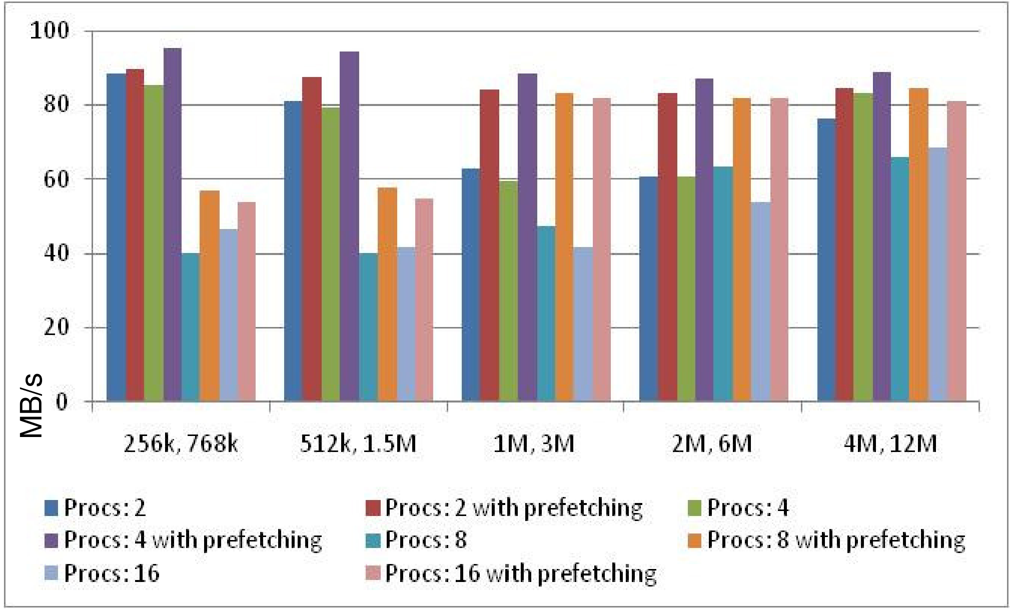
- NAS Parallel Benchmarks, MPI version, BTIO, Class B
- 1-d strided reads
- On NFS, the I/O read performance gain is 25%
- On PVFS, the I/O read performance gain is 8% with 4 processors, and 15% with 9 and 16 processors



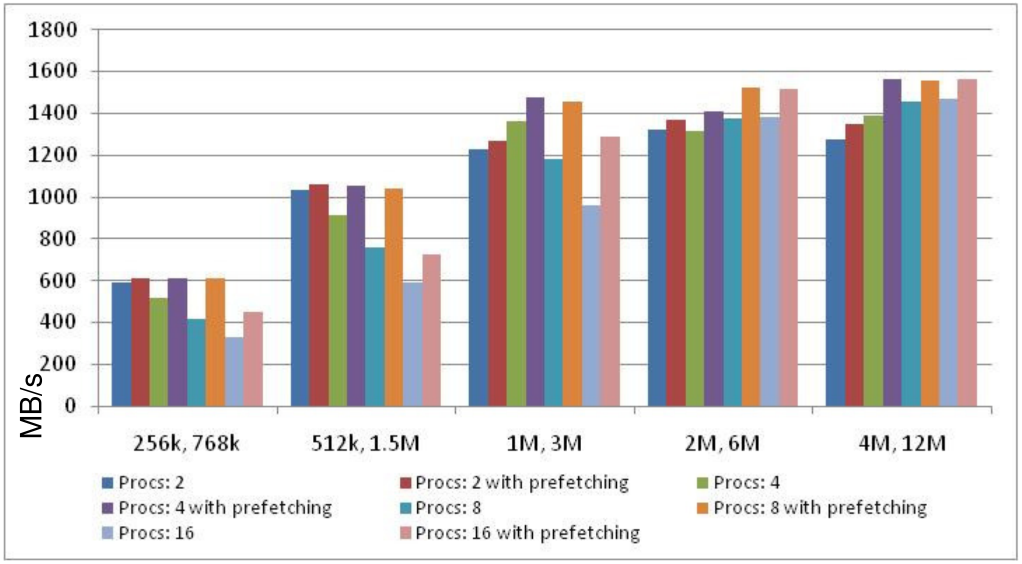
Performance On NFS (left) and on PVFS (right)

PERFORMANCE RESULTS

- PIO-Bench, 2-d nested strided
- On average I/O read performance improves 27% on NFS
- On PVFS, the performance gain on average is around 18%



Performance On NFS (above) and on PVFS (below)



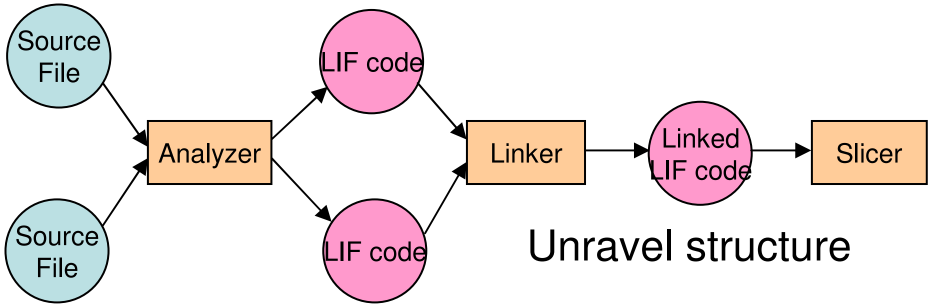
Runtime: Pre-execution based I/O Prefetching

- Idea
  - A pre-execution thread runs ahead and prefetch for the main thread
- Challenges
  - Guarantee expected program behaviors
  - Effective pre-execution (kept running ahead)
  - Coordination between main and the pre-execution thread
  - Compiler support (automatic)
  - System support (Caching library, Prefetching library)

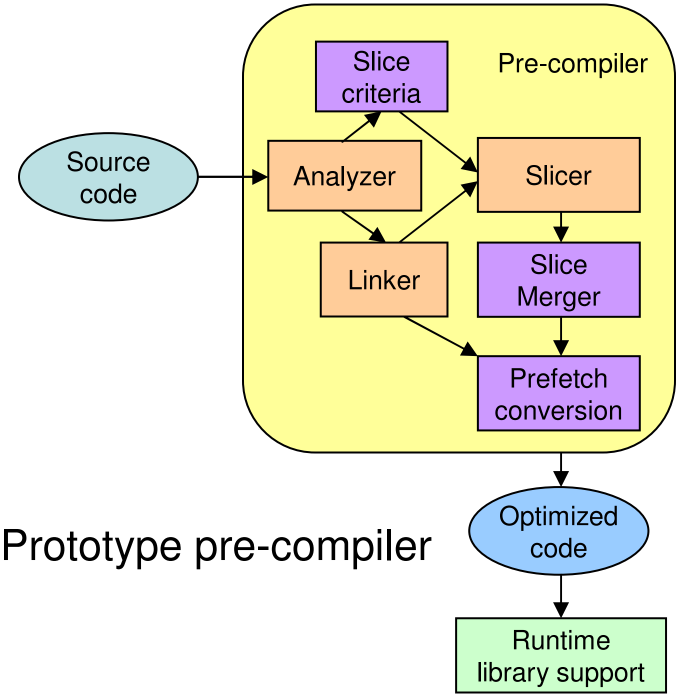
Solution

- Pre-execution thread
  - Code cloning
  - Code slicing: Non-IO related code is sliced away
  - Prefetch calls replace normal calls
    - Avoids the copy to user buffer
    - Could be non-blocking
- Thread-safety
  - Prefetch thread never commit writes
  - A separate prefetch file pointer into the opaque file handle
- Coordination and coherence
  - Delayed synchronization
- Prefetch conversion
  - Convert reads/writes/seeks to prefetch counterparts
  - Convert open/close/sync/deletion to sync points

Code slicing



Unravel structure



Prototype pre-compiler

- Automate construction: program slicing
  - Pre-execution thread is a subset of original program
  - Tool: Unravel
    - Analyzer, linker, slicer
  - Prototype pre-compiler
    - Slice criteria: I/O statements
    - Slice merger: OR bitmask of corresponding code lines
    - Prefetch conversion

SLICING ALGORITHM FOR PRE-EXECUTION

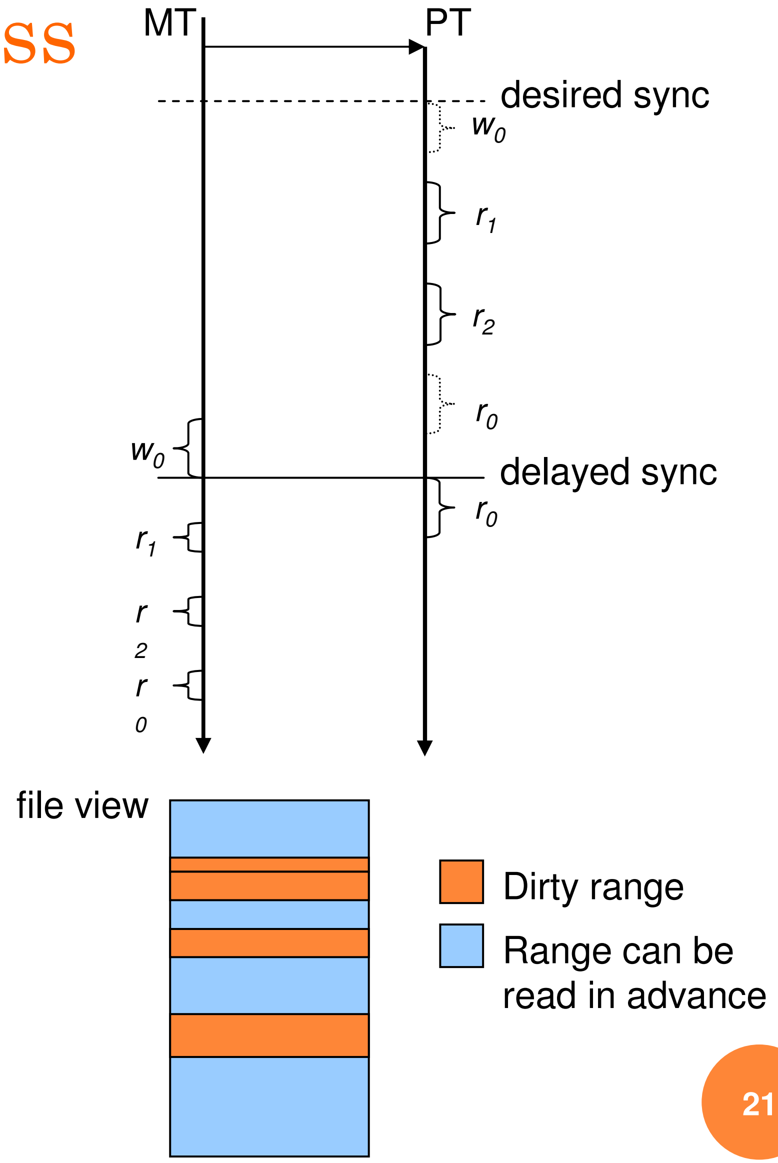
$$S_{<m,v>} = \begin{cases} S_{<n,v>} & \text{if } v \notin defs(n) \\ \{n\} \cup \left(\bigcup_{x \in refs(n)} S_{<n,x>} \right) \cup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in req(n)} S_{<k,y>} \right) & \text{otherwise} \end{cases}$$

<*m, v*>: slice criterion; m: statement, v: variable
n: all predecessor statements of *m*

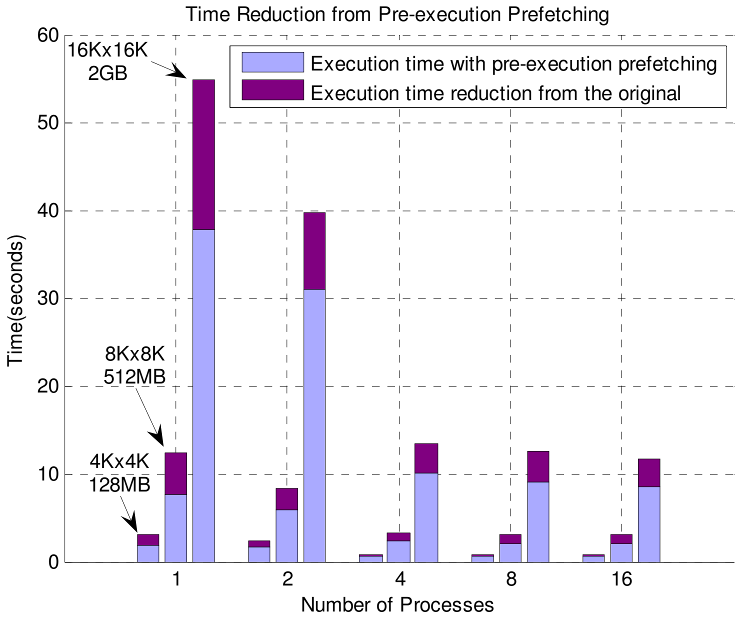
- If *n* does not assign *v*, recursively evaluate *S<n,v>*
- Otherwise
  - Include *n*
  - Include the slice on all referenced variables *x* in *n*
  - Include the slice on all referenced variables *y* in all statements *k* that control statement *n*

Coherence/Effectiveness

- Consistence
  - Synchronize on write
- Delayed synchronization
  - Detect dependency at runtime
  - Record write byte ranges, dirty range, then continue
  - Perform the delayed synchronization when conflict occurs
  - Dirty range is combined/split as writes/reads/syns go on
  - Dependency analysis table
- Preserves MPI-IO consistency for parallel I/O
  - Locking is performed for PT



PBENCH RESULT: TIME REDUCTION

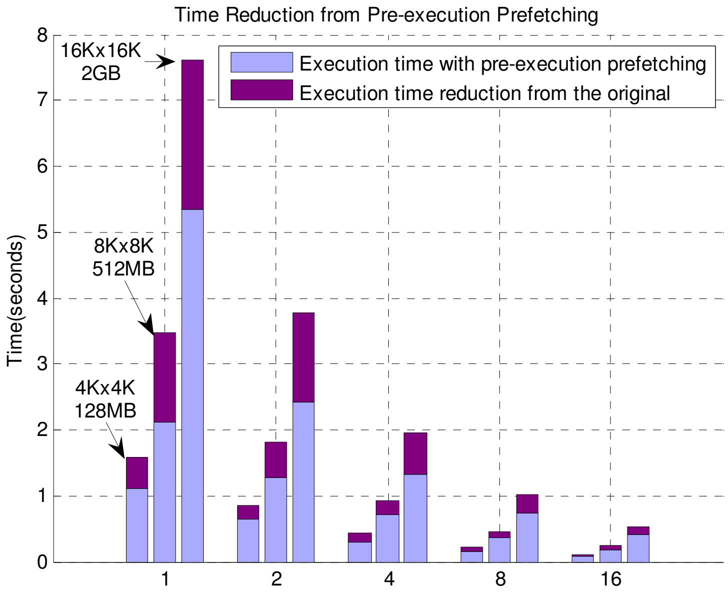


PBench Result on NFS

NFS result:
Time reduction: up to 37.9%
Average: 29.8%, 33.2%, 26.5% in three cases

- PBench test cases
  - 4Kx4K, 128MB
  - 8Kx8K, 512MB
  - 16Kx16K, 2GB

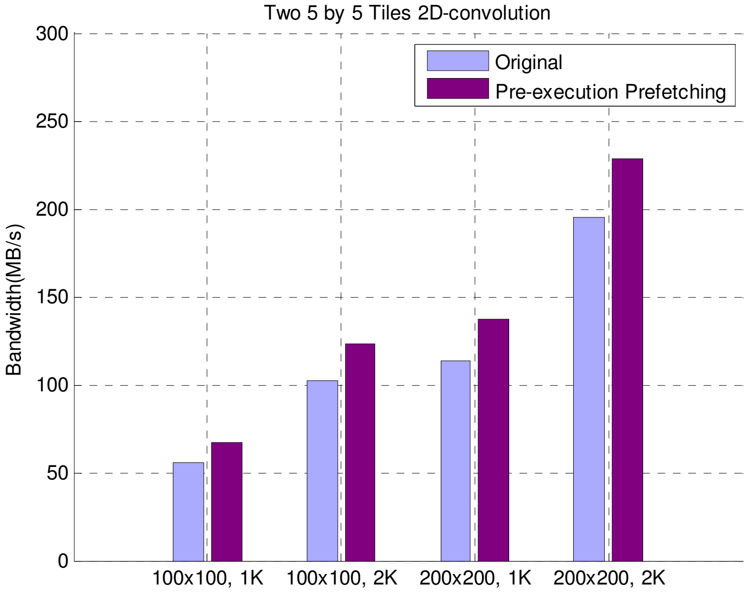
PVFS result:
Time reduction: up to 39.5%
Average: 28.1%, 28.4%, 30.2% in three cases



PBench Result on PVFS

TILE 2D-CONVOLUTION RESULT: SUSTAINED BANDWIDTH

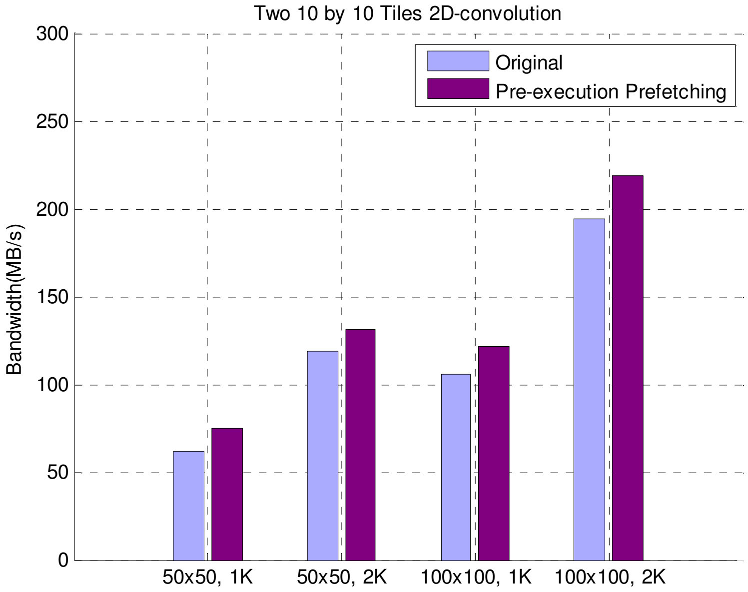
- Tile 2D-convolution
  - 5x5 titles, with 100x100 and 200x200 elements respectively, with size 1KB and 2KB respectively
  - 10x10 titles, with 50x50 and 100x100 elements respectively, with size 1KB and 2KB respectively
  - Data size: 256MB, 512MB, 1GB, 2GB



Two 5 by 5 tiles on PVFS

Two 5 by 5 tiles result:
- Bandwidth improvement: up to 20.6%
- Average: 18.4%

Two 10 by 10 tiles result:
- Bandwidth improvement: up to 20.3%
- Average: 14.7%



Two 10 by 10 tiles on PVFS

CONCLUSIONS

- Current Progress (New I/O architecture)
  - Understanding parallel I/O workloads
    - Introduction of new signature to represent I/O workloads
  - New approaches for improving parallel I/O performance
    - Prefetching with the use of I/O signature
    - Prefetching by using pre-execution of I/O accesses
  - Research and development
- Need to do
  - Implementing prefetching strategy in PVFS2
    - Parallel data access pattern, scheduling, PVFS
  - Improvement
    - Better prefetching algorithms, code slicing, analysis of coherence
  - Integrated approach