

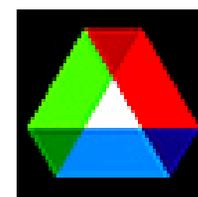


HECURA: *THE SERVER PUSH-IO ARCHITECTURE FOR HIGH-END COMPUTING*

Xian-He Sun

William Gropp, Rajeev Thakur

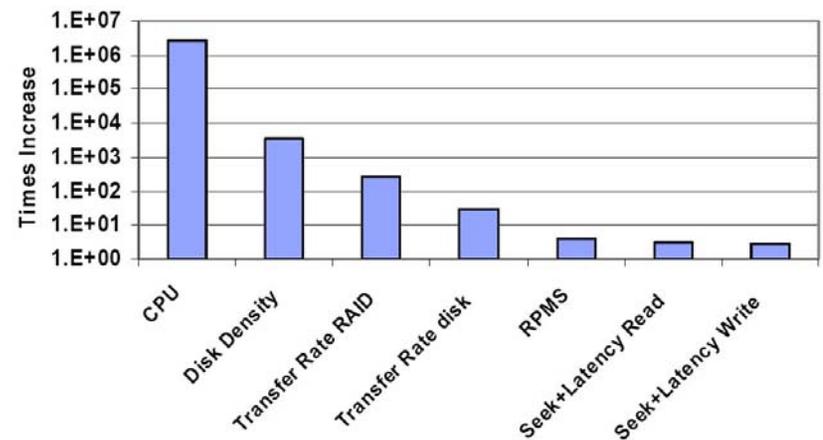
Illinois Institute of Technology
Argonne National Laboratory
sun@iit.edu





The Problem: I/O Bottleneck

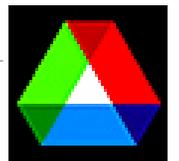
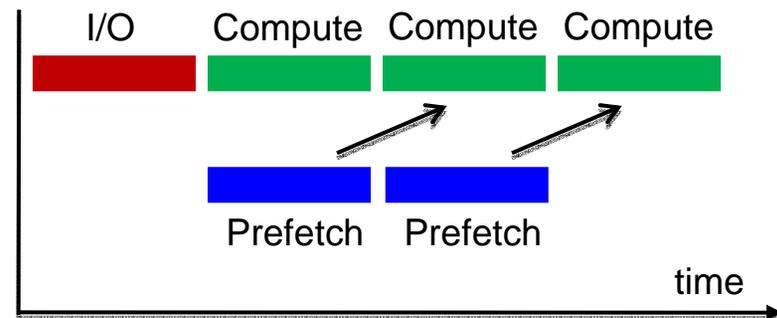
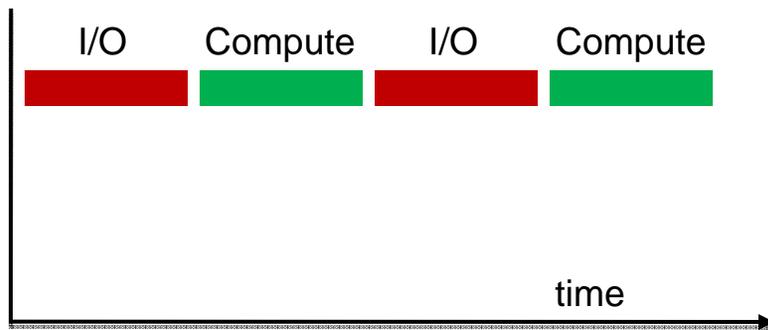
- Poor **Parallel I/O** performance for complex non-contiguous access
- **Improving** the performance of large number of small I/O requests is a necessity
- **Prefetching** – fetch data before a client demands for it
- **Limitations of Existing Prefetching**
 - Conservative and limited to static prediction strategies
 - No prediction strategy on **when** to prefetch
 - Only works for simple access patterns with locality





The Challenge Of Prefetching

- ▶ Move data closer to the processor before it is demanded
- ▶ **Challenges**
 - ▶ **What** data should be prefetched?
 - ▶ Costly, only use the simple one
 - ▶ **When** should prefetching occur?
 - ▶ Costly, no try





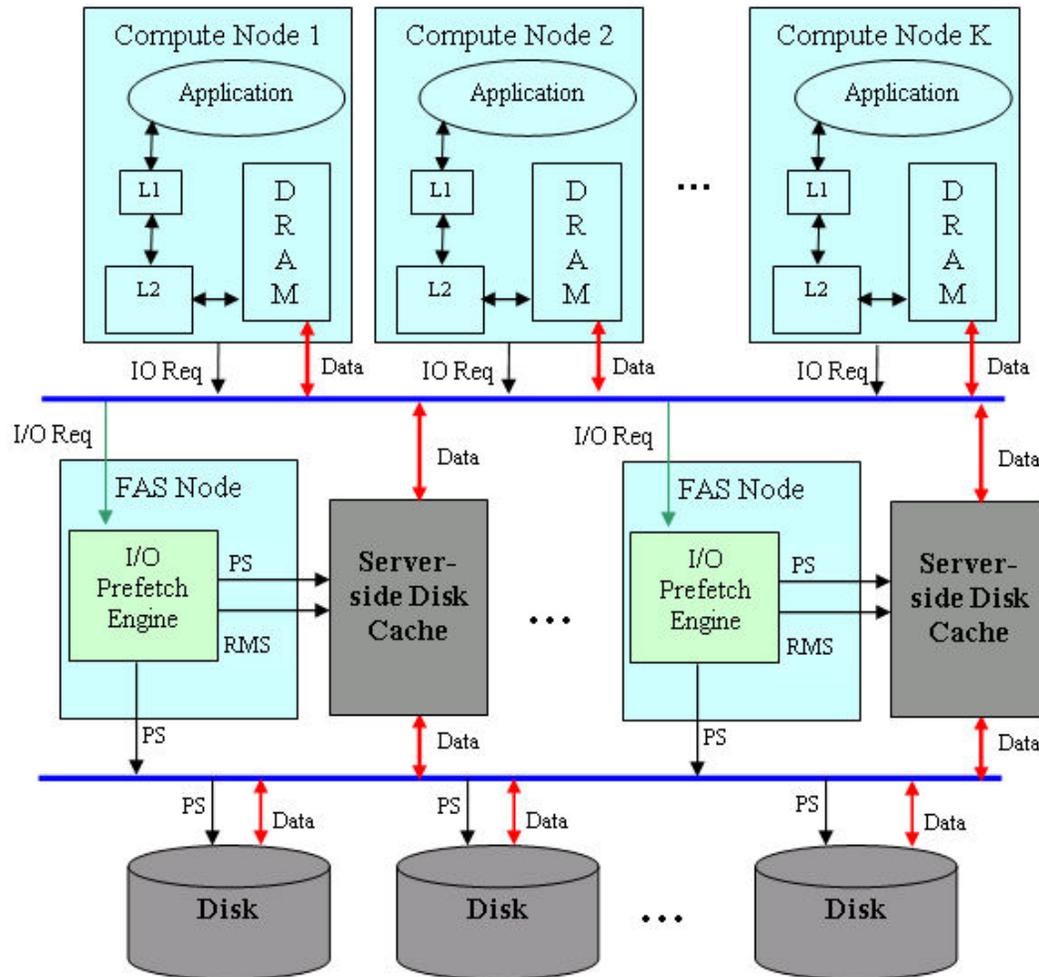
Our Solution: File Access Server (FAS)

- ▶ Trade computing power with data access
- ▶ A “dedicated” 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





FAS Enabled Parallel I/O



- ▶ File Access Server is on I/O servers
- ▶ Push data from disk to compute nodes





One Year Achievement

Project Organization

▶ Research Team

- ▶ PI and Co-PIs: Drs. Xian-He Sun, Bill Gropp, Rajeev Thakur
- ▶ 1 full-time postdoc researcher: Dr. Surendra Byna
- ▶ 2 Ph.D. students: Yong Chen, Gregor Tamindzija

- ▶ Dr. Byna is located at ANL
- ▶ Dr. Sun is an ANL guest faculty, Mr. Chen has a long term ANL pass

▶ Communication

- ▶ Push-IO Wiki
- ▶ Biweekly Meeting at Argonne





Research Activity

- ▶ Survey of applications and benchmarks
- ▶ Software architecture
 - ▶ Integrated global design
 - ▶ Low-level component design
 - ▶ New component: Helper thread
- ▶ What data to fetch
 - ▶ Pattern classification
 - ▶ Prediction algorithm selection
- ▶ Implementation
 - ▶ Trace collection
 - ▶ Data access pattern identification
 - ▶ Prediction algorithm
 - ▶ Client cache
 - ▶ Pre-execution threads





Survey: I/O benchmarks and applications

- ▶ Studied various benchmarks to find large number of small I/O accesses, complex I/O patterns
- ▶ Benchmarks
 - ▶ pio-bench, mpi_tile_io, mpi_io_test, flash-io-bench, NPB BTIO, LU
- ▶ Applications
 - ▶ mpqc, NaSt3dGP, mpiBLAST
 - ▶ CCSM (Community Climate System Model) application
- ▶ Suggestions on more applications with complex I/O patterns are welcome





Adaptive and Aggressive Prefetching

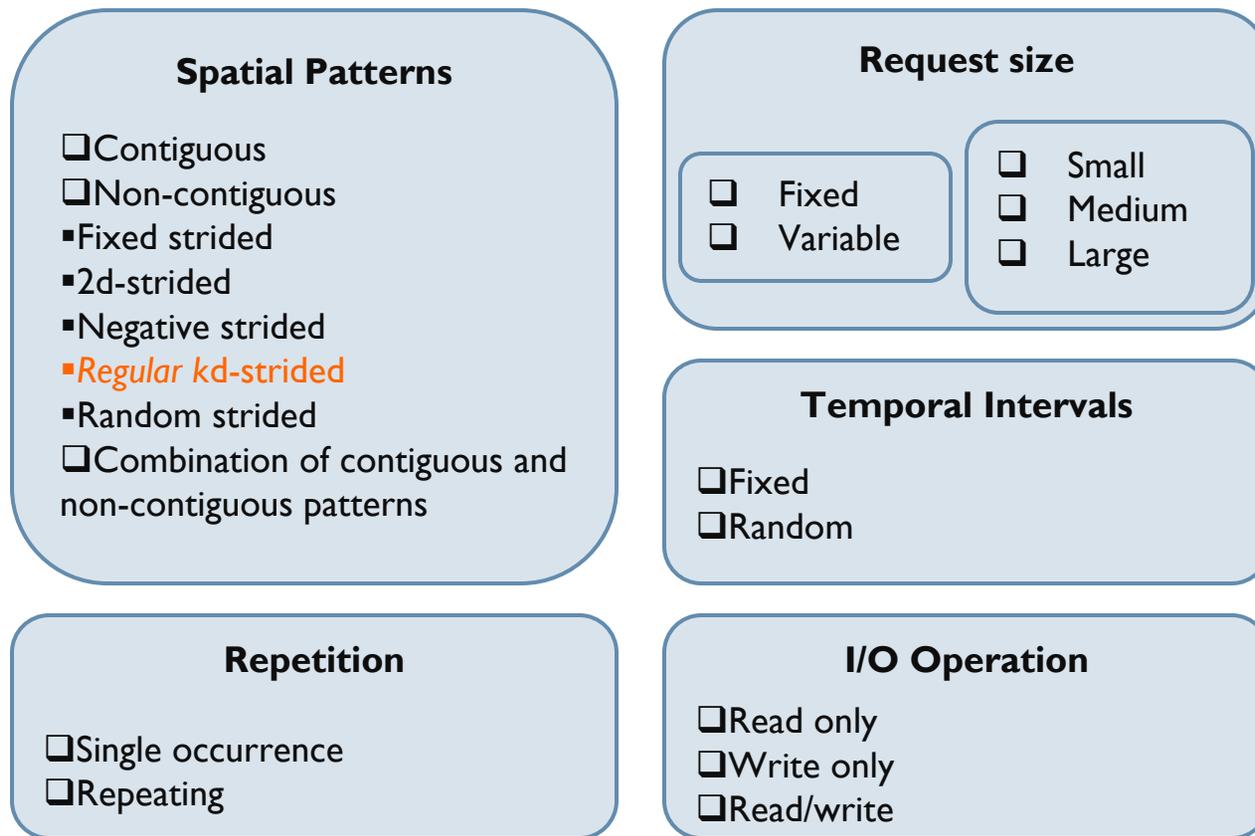
- ▶ **Multi-dimensional**
 - ▶ location of data, the amount of data, the mode of accessing data, and strides
 - ▶ Time between any two accesses, between successive accesses to a specific data block
- ▶ **Aggressive Prefetching**
 - ▶ Overhead to predict the future accesses is no longer a issue
 - ▶ New aggressive methods to predict irregular data accesses
- ▶ Adapt a prefetch strategy based on the **data access pattern**
- ▶ Reduce prediction time by using hints provided by compiler and application/user





Pattern Classification

► Comprehensive I/O access pattern classification





Pattern Prediction Algorithms

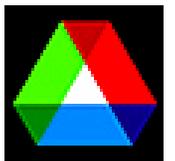
- ▶ Simple stride prediction algorithm
- ▶ k-d stride prediction algorithm
- ▶ Markov model prediction
- ▶ Multi-level difference table
- ▶ Time series analysis
- ▶ Artificial Neural Networks





Prediction Method Selection

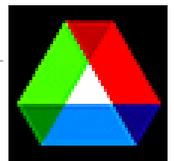
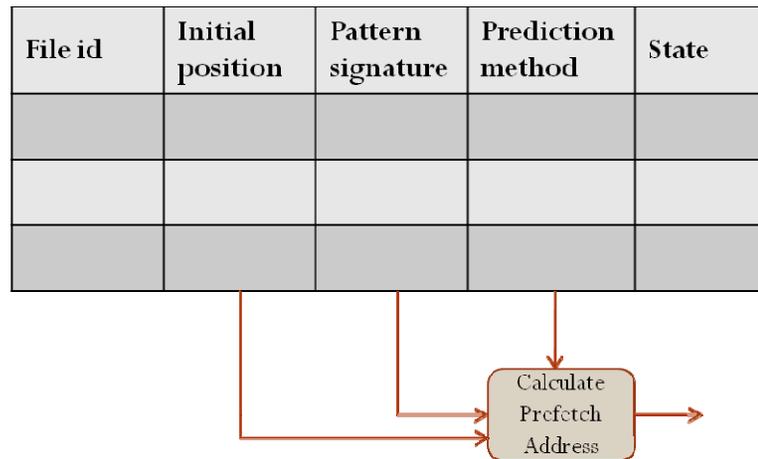
- ▶ Selection algorithm for choosing prediction method
- ▶ Prediction engine has learning, prediction and supervision states
- ▶ When a file is opened for file operations, prediction method enters learning state
- ▶ Once a steady pattern is found, prediction state calculates future I/O accesses
- ▶ Supervision state observes pattern changes and feedback from traces in order to choose a different prediction algorithm on-the-fly





In the prediction state...

- ▶ Pattern signature: Defines pattern
 - ▶ {init position, dimension, ({stride pattern}, {request size pattern}, {number of repetitions pattern}), [...]}, # of repetitions
 - ▶ Example: {1024, 1, (4096, 1024, 99), 1}, {2048, 1, (8192, 2048, 99) 1}, {4096, 1, (16384, 4096, 99),1}, {8192, 1, (32768, 8192, 99),1},
 - ▶ Example: {4096, 2, ([2048, 1024, 1], [6144, 1024, 1]), 99}, {8192, 2, ([4096, 2048, 1], [12288, 2048,1], 99)}
- ▶ Supervision state updates # of reps
- ▶ As the number of repetitions increase, the sampling distance of trace observation increases





Implementation of FAS

- ▶ Implementation of Prediction Engine
- ▶ Implementation of creating pre-execution thread
- ▶ Testing full cycle of FAS
 - ▶ Test results will be published
- ▶ Implementation of hints pool and using hints

- ▶ Long term goal
 - ▶ Implementation FAS into PVFS2





I/O Access Tracing

- ▶ Capturing I/O requests at clients
- ▶ File I/O calls
 - ▶ Wrapper functions for open, read, fread, fseek, and close
- ▶ Using Profile MPI (PMPI) to wrap the following MPI-IO functions
 - ▶ MPI_File_read, MPI_File_iread, MPI_File_read_at, MPI_File_read_all, MPI_File_seek
- ▶ Fields of I/O Request Trace Buffer
 - Process ID, File Descriptor, File Position, Number of bytes, Timestamp, File operation





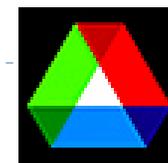
Sample trace

Proc ID	Rank	File #	File Pos	# of Bytes	Time(s)	I/O Op
29074	0	16	0	1024	7.501304	MPI_READALL
29074	0	16	2048	1024	7.545113	MPI_READALL
29074	0	16	8192	1024	7.588820	MPI_READALL
29074	0	16	10240	1024	7.629080	MPI_READALL
29074	0	16	16384	1024	7.674620	MPI_READALL
29074	0	16	18432	1024	7.717406	MPI_READALL
29074	0	16	24576	1024	7.761096	MPI_READALL
29074	0	16	26624	1024	7.805323	MPI_READALL
29074	0	16	32768	1024	7.849006	MPI_READALL
29074	0	16	34816	1024	7.889342	MPI_READALL
29074	0	16	40960	1024	7.933130	MPI_READALL
29074	0	16	43008	1024	7.977495	MPI_READALL
29074	0	16	49152	1024	8.021255	MPI_READALL
29074	0	16	51200	1024	8.065650	MPI_READALL
29074	0	16	57344	1024	8.109380	MPI_READALL
29074	0	16	59392	1024	8.153768	MPI_READALL
29074	0	16	65536	1024	8.197396	MPI_READALL
29074	0	16	67584	1024	8.241783	MPI_READALL
29074	0	16	73728	1024	8.285514	MPI_READALL
29074	0	16	75776	1024	8.331166	MPI_READALL
29074	0	16	81920	1024	8.373678	MPI_READALL
29074	0	16	83968	1024	8.418154	MPI_READALL
29074	0	16	90112	1024	8.461751	MPI_READALL

pio-bench with nested stride test

2k stride

6k stride





Initial Performance Results

- ▶ Prefetching using offline I/O hints
- ▶ Testing environment
 - ▶ Collective caching code borrowed from Wei-keng Liao at Northwestern Univ.
 - ▶ Cache size: 32 MB, Cache page size: 64 KB, File system: NFS

Benchmark	Pattern signature	# of I/O Reads	Page fault rate (%)	Page fault rate with FAS (%)	Prediction overhead (seconds)
PIO-Bench, simple strided (4k stride)	{INIT, 1, (4096, 1024, 199)}	200	7%	2%	0.00036
PIO-Bench, simple strided (16k)	{INIT, 1, (8192, 1024, 199)}	200	25%	2%	0.00036
PIO-Bench, simple strided (32k)	{INIT, 1, (16384, 1024, 199)}	200	50%	2%	0.00036
PIO-Bench, simple strided (> 64k)	{INIT, 1, (32768, 1024, 199)}	200	100%	2%	0.00036
PIO-Bench, nested strided (4k, 12k)	{INIT, 2, ({-, 1, (4096, 1)}, {-, 1, (12288, 1)}, 99)}	200	15%	4%	0.0004
PIO-Bench, nested strided (16k, 48k)	{INIT, 2, ({-, 1, (16384, 1)}, {-, 1, (49152, 1)}, 99)}	200	50%	4%	0.0004
PIO-Bench, nested strided (64k, 192k)	{INIT, 2, ({-, 1, (65536, 1)}, {-, 1, (196608, 1)}, 99)}	200	100%	4%	0.0004
LU decomposition, out-of-core (8192 x 8192 double precision matrix)	{0, 3, ({1049088, 1, (524544, 1)}, {0, 1, (522368, 1)}, {524544, 1, ({518272, (-4096)}, {1, (1)}), 125}	8252	76%	0%	0.0091
BTIO (Class B, 16 processors)	{INIT, 1, (42450944, 5308416, 39)}	40	100%	10%	0.00024



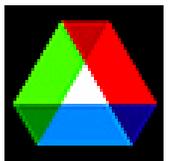
Conclusions

▶ Progress

- ▶ Design of Software Architecture, with new component
- ▶ Exciting technical finding: pattern, signature, algorithm
- ▶ Implementation progress well (prediction, pre-execution)
- ▶ Benefit of Academic-Lab collaboration, more collaboration

▶ Next steps

- ▶ Finish the one-cycle implementation
- ▶ Testing and improvement
- ▶ Include more components, such as the hint generators,
- ▶ Finish and integrate pre-execution hints
- ▶ Full development





Thank you!

Questions?





▶ Backup slides





FAS: COMPONENTS

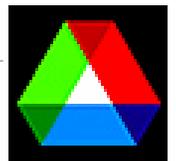
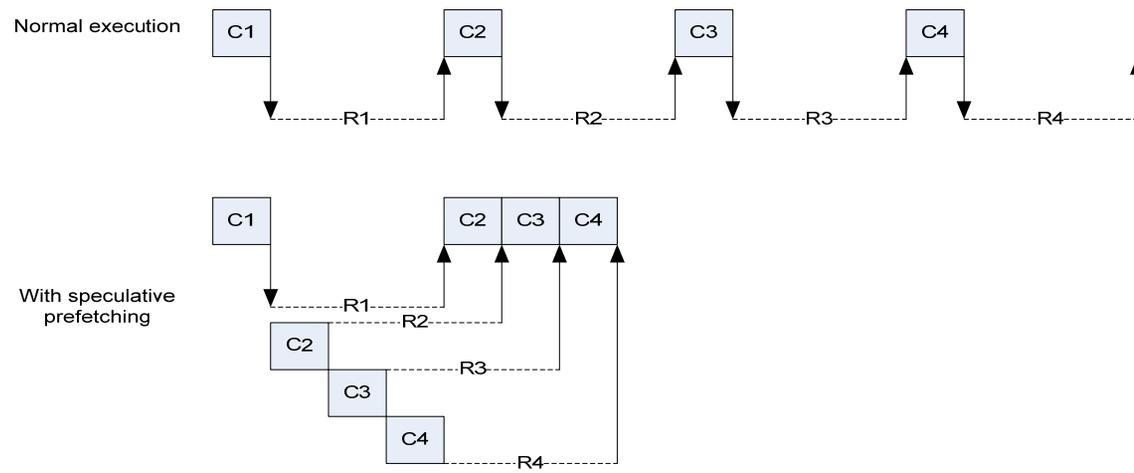
- ▶ **Prefetch Strategy Selector:** adaptively selects an appropriate method to predict future accesses from
 - ▶ Compiler hints
 - ▶ Post-execution analysis
 - ▶ Pattern prediction algorithms
- ▶ **Hint2Request converter:** converts hints to I/O requests and keeps them in prefetch queue
- ▶ **Tracer:** traces I/O requests and stores them in an I/O request trace buffer
- ▶ **Prefetch predictor:** decides *what* data to push using pattern prediction algorithms
- ▶ **Request generator:** decides *when* to push the data
- ▶ **Data propeller:** validates prefetching requests for expiration and issues push instruction to move data from disk to prefetch cache





More on Pre-execution Hints

- ▶ Pre-execution is useful, when access patterns are unknown or accesses are irregular or random
- ▶ An example, where periodic reads (R2, R3 and R4) latency are completely masked
- ▶ Pre-execution thread can run
 - ▶ Utilizing idle cycles
 - ▶ Competing with regular computation process
 - ▶ Pre-executing remotely





A Simple System: Current Focus

- ▶ Working on understanding I/O access patterns
- ▶ Developing algorithms for adaptive prefetching method selection
- ▶ Testing “push” strategy from PVFS to client nodes

