Improving Data Access Performance with Server Push Architecture

Xian-He Sun*, Surendra Byna, and Yong Chen

Scalable Computing Software Laboratory

Illinois Institute of Technology

Fermi National Laboratory*

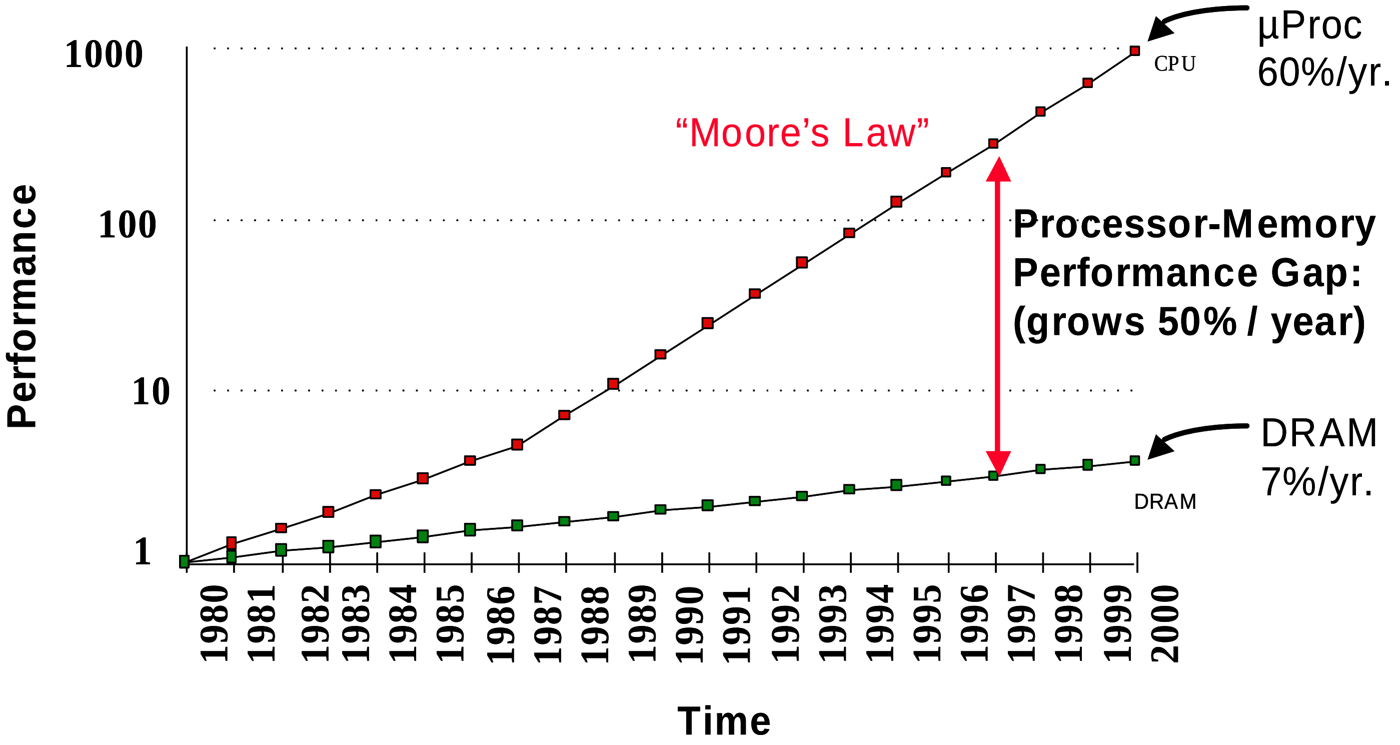
sun@iit.edu

March 25, 2007

NGS07

The Problem: Memory Wall

Processors are getting faster more quickly than memory

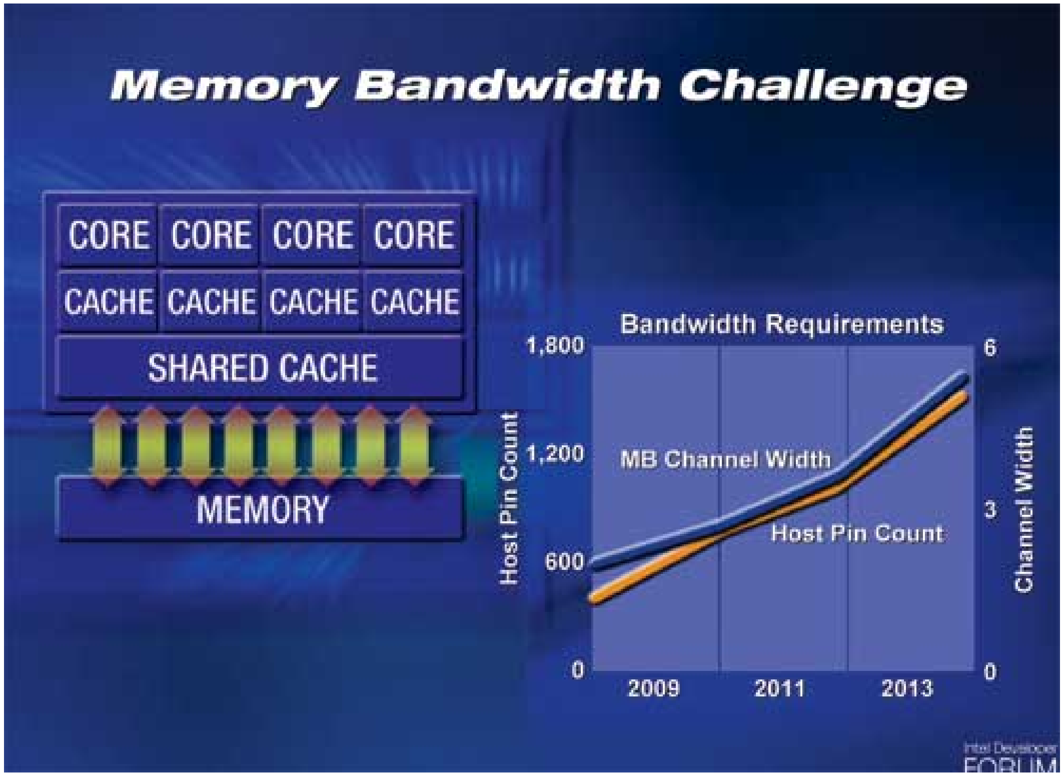


Solutions

- ❑ Improve hardware
- ❑ Cache memories
- ❑ Prefetching
- ❑ Multithreading

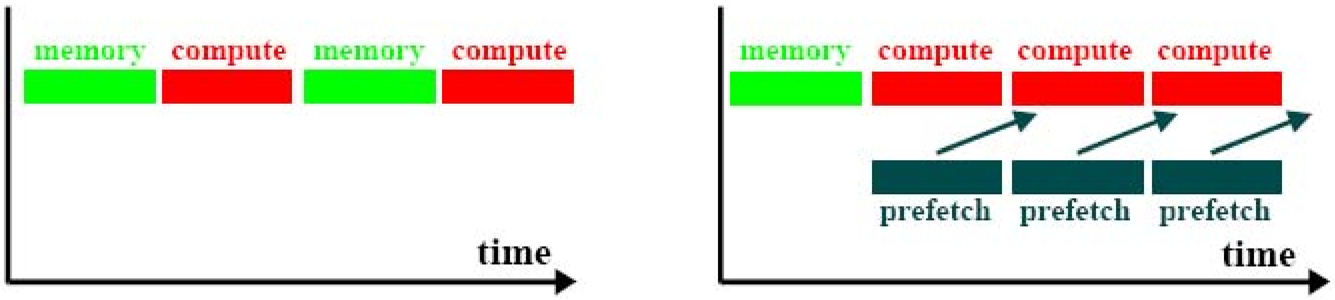
Current Solutions of Memory Wall

- Solutions
  - Wider front-side bus
  - Processor in memory
  - Send threads to memory – Threadlets
  - Memory Hierarchy: Adding an L4 cache
  - Prefetch, pre-execute



The Challenge of Prefetching

- Move data closer to the processor before it is demanded
- Prefetch data as close as possible to the processor in the memory hierarchy
- Challenges
  - What data should be prefetched?
  - When should prefetching occur?



What to Fetch? and When to Fetch?

- **What:** Requires prediction of what data the processor is going to access in the future
- Prefetching Strategies
  - Sequential, Adaptive Sequential, Strided, Markov Prefetching, Distance Prefetching

- **When:** Not too early and not too late
  - Best, if time between now and next access is equal to prediction time + overhead to fetch the data (performance evaluation)
- Prefetching Strategies
  - Prefetch-on-miss, Prefetch Always, Tagged Prefetching

- Limitation: Only practical for very simple methods

Our Solution:

The Data-access Memory Server (DMS)

- Separate data access with data processing, have a dedicated computing power for data access
- Goals
  - Proactively prefetches the data closer to the processor, on time
  - Adapts to various prefetching strategies based on application data access patterns
  - Adaptive replacement policies based on prediction
  - Special architectures are designed. Aggressive Prefetching, data access pattern identification, and performance modeling

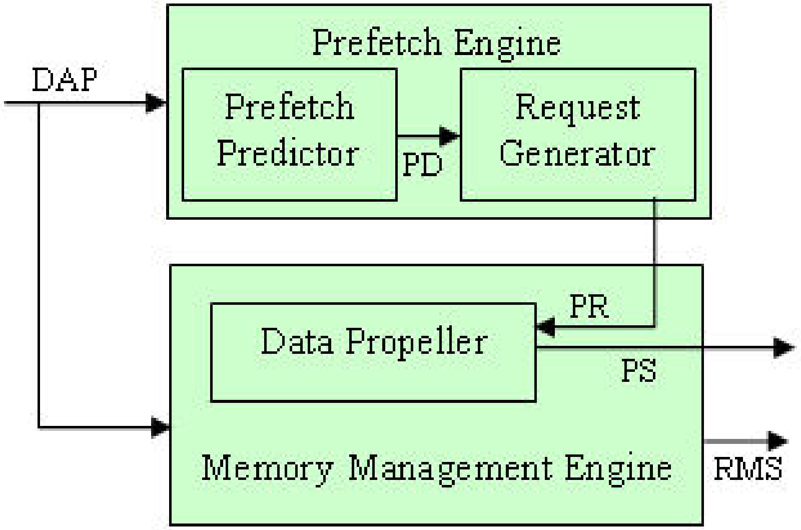
DMS – Prefetch Strategy

- Prefetch Engine (PFE)
  - Prefetch predictor (*What*)
  - Request generator (*When*)

(software solution)

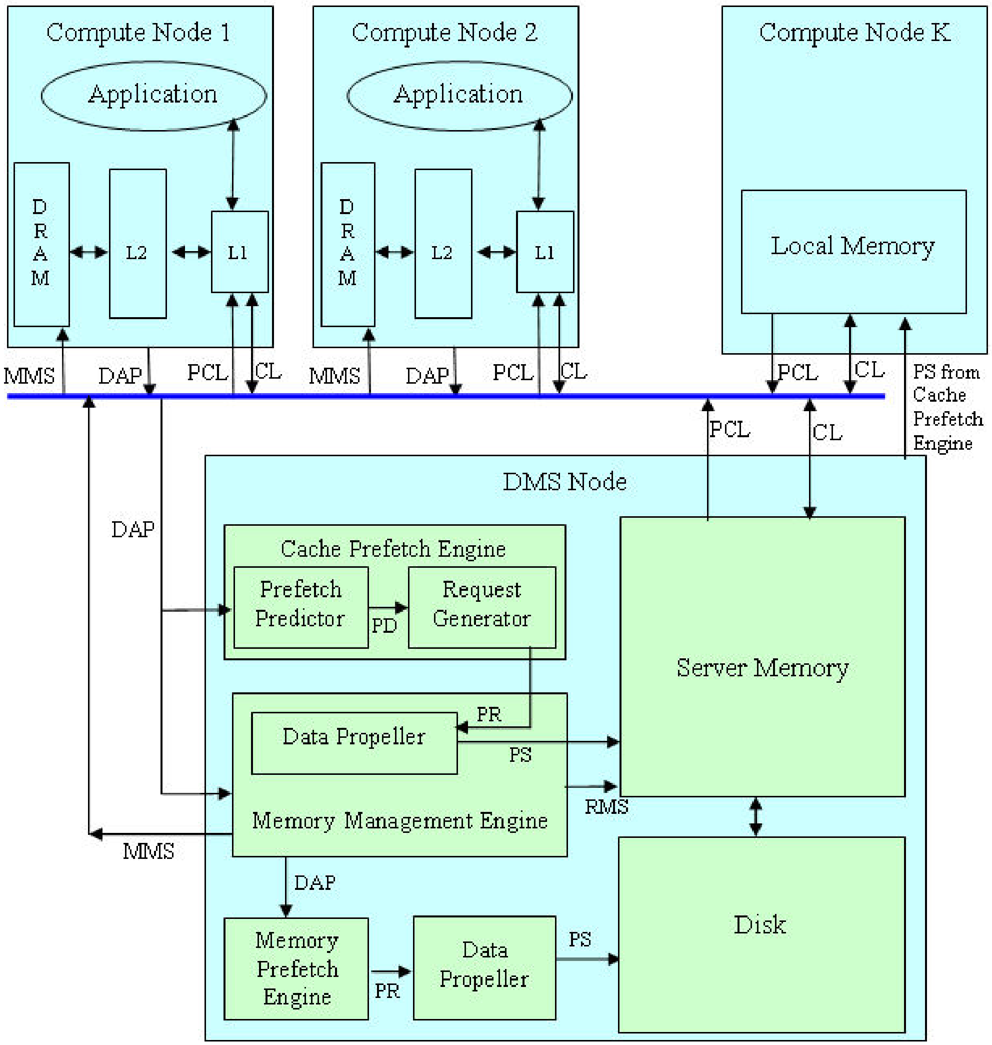
- Memory Management Engine (MME)
  - Data Propeller: Issues the prefetch instructions
  - Pushes the data from the server to the clients
  - Deals with raw cache misses or page faults

(hardware support)



DMS – Architecture Design

- Multiprocessor Platforms
  - Clusters
  - SMP
  - Multicore Processor
- Classified based on the functionality of PFE and MME
- I/O Server Model



Challenges in Implementing the DMS

Performance modeling, evaluation, optimization

- Classification and Reorganization of data access patterns
- Aggressive and in-time prefetching
- Fetch and replacement policies

Hardware support

- Support of prediction
- Support of push data

Challenge: What data to push?

Performance prediction

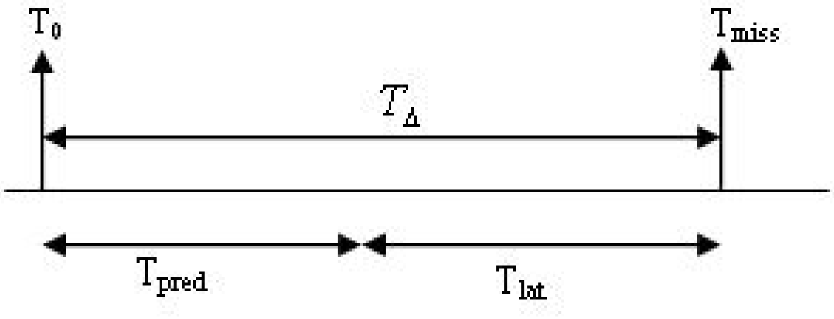
- Multi-dimension
  - location of data, the amount of data, the mode of accessing data, and strides
  - Time between any two accesses, between successive accesses to a specific data block
- Aggressive Prefetching
  - Overhead to predict the future accesses is no longer a issue
  - New aggressive methods 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

Challenge: When to push?

Performance modeling

Three factors

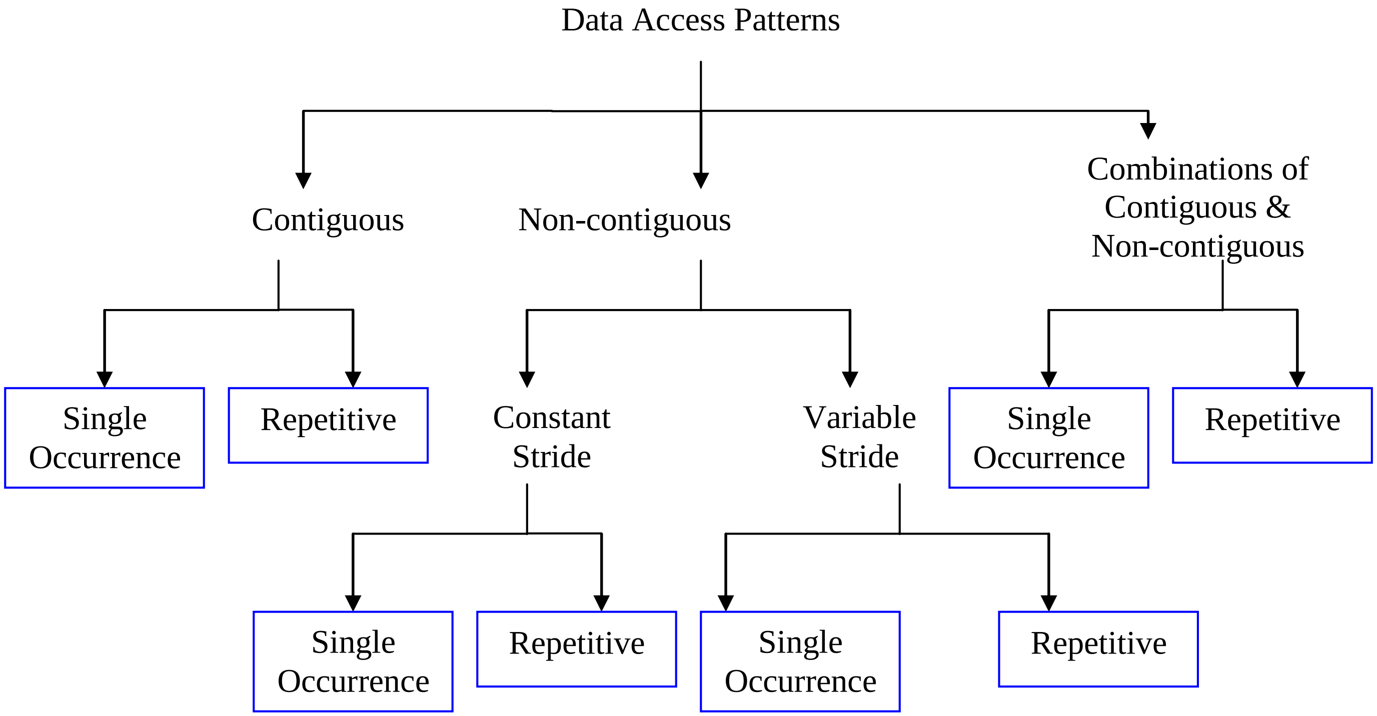
- Time to predict the future accesses
  - Based on the chosen prefetching method
- Data transfer latency
  - Data access delay model
- Time till next cache miss
  - Data access model
- Overlapping the network latency by increasing the prefetch distance
- Adapting the prefetch distance based on the network latency variation



Identify and Match Access Pattern

Byna, Sun, Gropp, Thakur 04,07

- Classification of data access patterns based on non-contiguity between accesses and the repetitive behavior of patterns



Predicting Memory Access Cost

Cameron and Sun 03,07

Average memory access cost = Hit time + Miss Rate * Miss Penalty

= (Number of TLB hits * Time to access TLB) +

(Number of TLB misses * TLB miss penalty) +

(Number of L_1 hits) * (Time to access L_1) +

(L_1 misses * L_1 penalty) + (L_2 misses * L_2 penalty)

+ … +

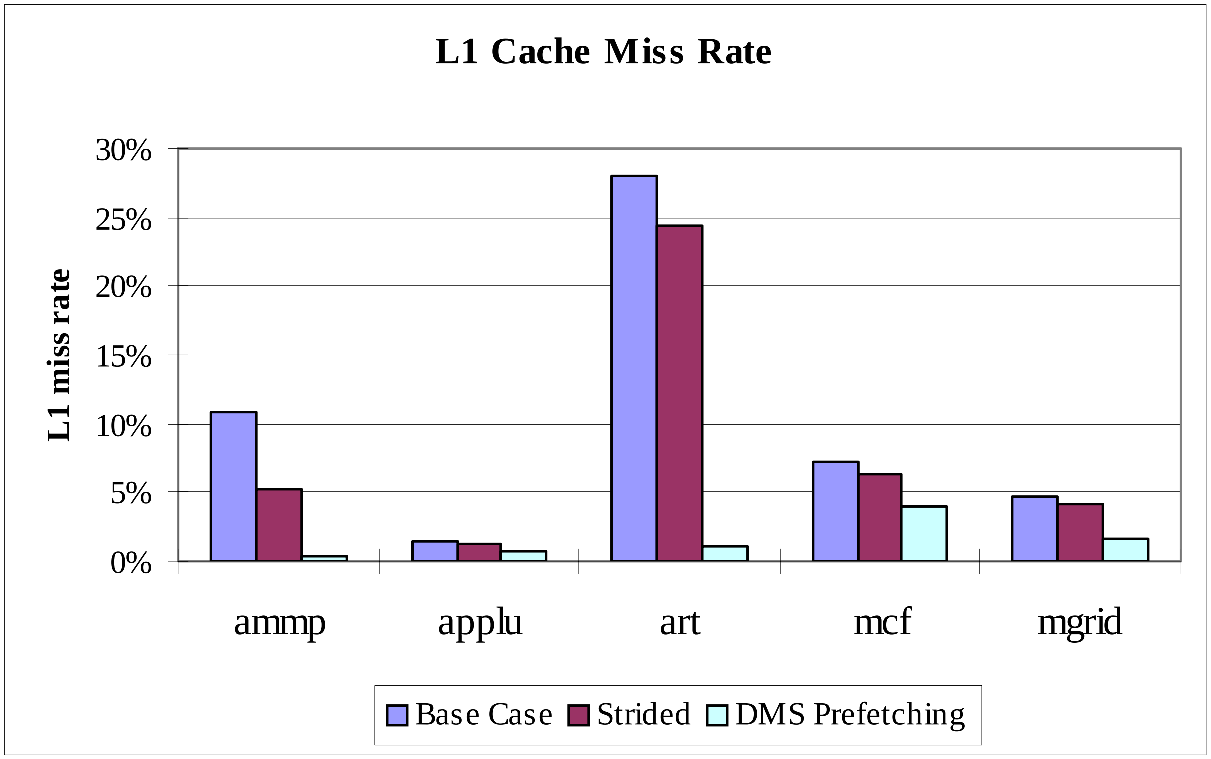
(L_M misses * $L_M$penalty)

Total Miss penalty: $T_m = \sum_{k=1}^{M} (M_k * T_k) - \alpha$

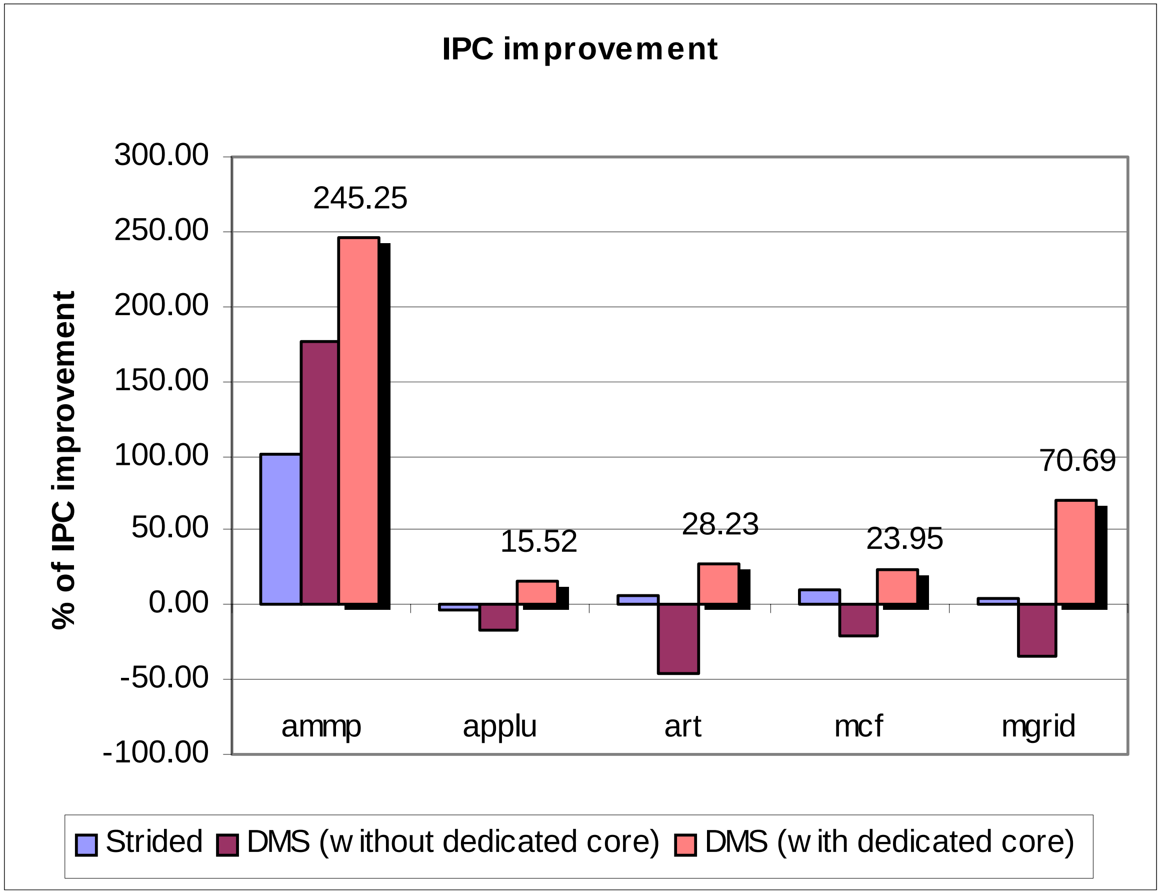
$$M_k = \sum_{i=1}^{m} M_{(k,i)}^{c} + \sum_{i=1}^{n} M_{(k,i)}^{n}$$

L1 Cache Miss Rate – SPEC2000 Benchmark

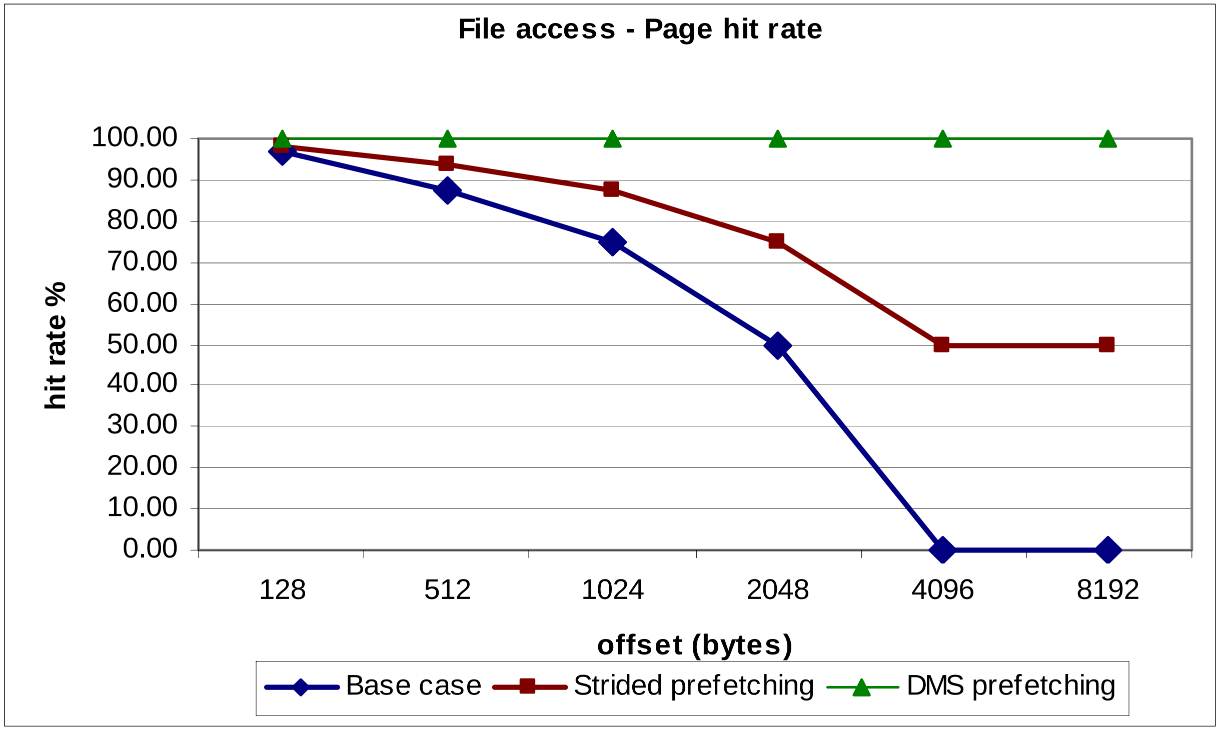
- (Enhanced) Simplescalar simulator
- SPEC2000 benchmarks with high L1 cache miss rates



Benchmark – IPC Improvement



Potential of DMS – File accesses



Conclusion

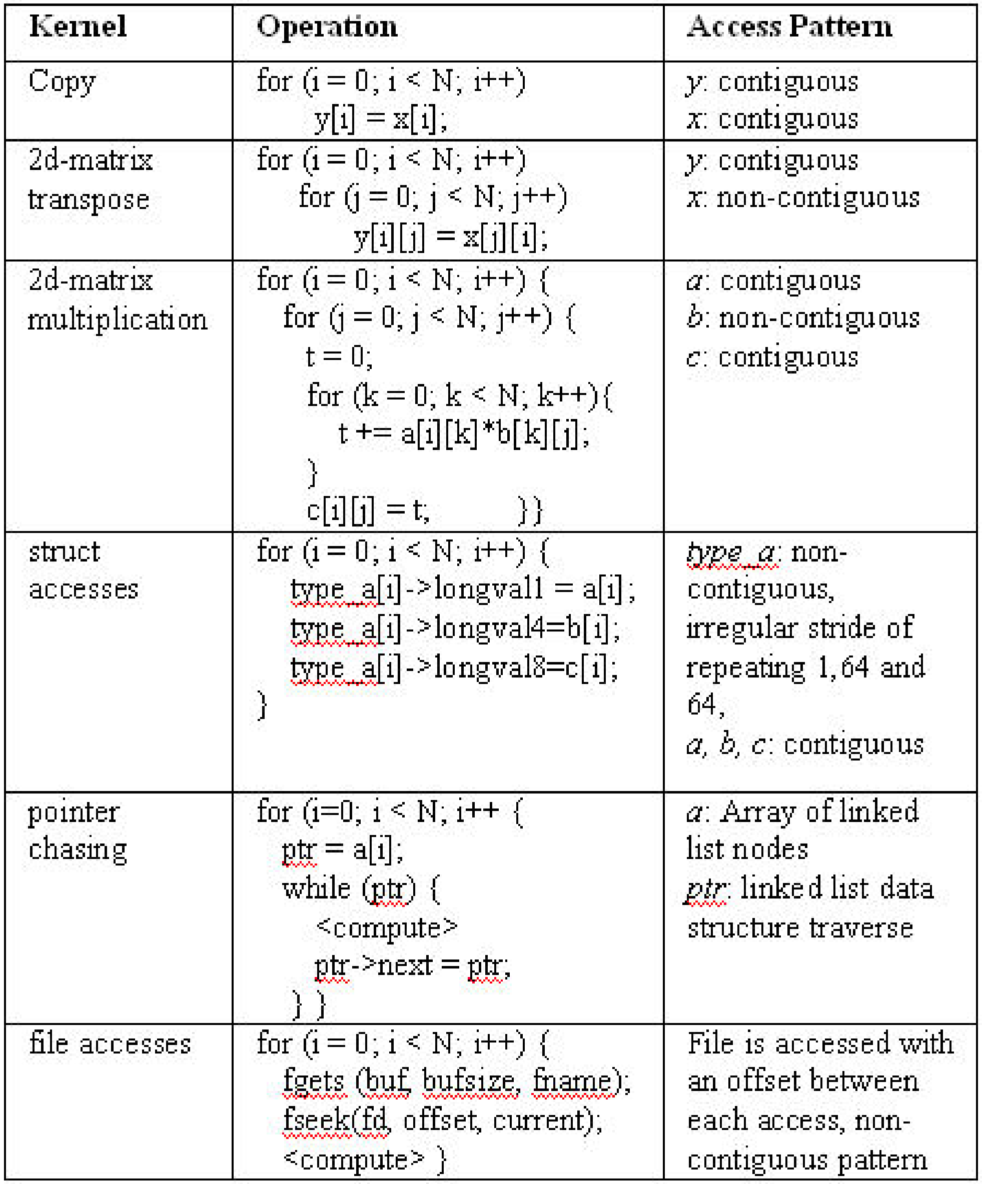
- Memory (I/O) as a service
  - DMS proactively and adaptively pushes the data closer to the processor
  - Adaptive and timely prefetching strategies
  - Has the potential to avoid CPU stall time
- Key technology: Performance measurement, evaluation, and optimization (PMEO)
  - What to fetch, When to fetch
  - System software solution with hardware support
- Current and future work
  - I/O server

Questions?

Potential of DMS – Benchmark Kernels

- Kernels from SPEC 2000, BLAS, Stream benchmarks
- Represent various data access patterns
- Copy – contiguous
- Combinations of contiguous and non-contiguous patterns
- Irregular patterns
- Irregular pointer chasing accesses
- I/O accesses

Table 1. Benchmark kernels



Kernel	Operation	Access Pattern

Copy	for (i = 0; i < N; i++)
	y[i] = x[i];
		y: contiguous
		x: contiguous

2d-matrix transpose	for (i = 0; i < N; i++)
	for (j = 0; j < N; j++)
	y[i][j] = x[j][i];
		y: contiguous
		x: non-contiguous

2d-matrix multiplication	for (i = 0; i < N; i++) {
	for (j = 0; j < N; j++) {
	t = 0;
	for (k = 0; k < N; k++){
	t += a[i][k]*b[k][j];
	}
	c[i][j] = t; }}
		a: contiguous
		b: non-contiguous
		c: contiguous

struct accesses	for (i = 0; i < N; i++) {
	type_a[i]->longval1 = a[i];
	type_a[i]->longval4=b[i];
	type_a[i]->longval8=c[i];
	}
		type_a: non-contiguous, irregular stride of repeating 1,64 and 64,
		a, b, c: contiguous

pointer chasing	for (i=0; i < N; i++ {
	ptr = a[i];
	while (ptr) {
	<compute>
	ptr->next = ptr;
	} }
		a: Array of linked list nodes
		ptr: linked list data structure traverse

file accesses	for (i = 0; i < N; i++) {
	fgets (buf, bufsize, fname);
	fseek(fd, offset, current);
	<compute> }
		File is accessed with an offset between each access, non-contiguous pattern