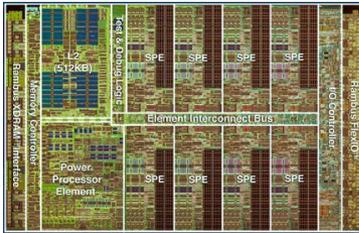


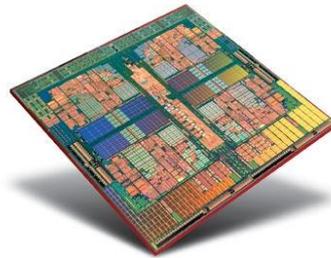
Core-Aware Memory Access Scheduling Schemes

Zhibin Fang, **Xian-He Sun**, Y. Chen, and S. Byna
Illinois Institute of Technology

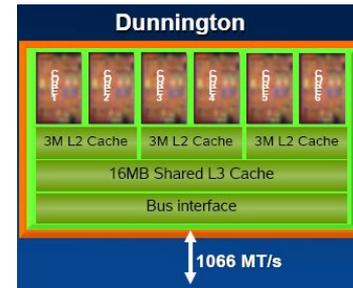
Multi-core is the Trend of High Performance Processors



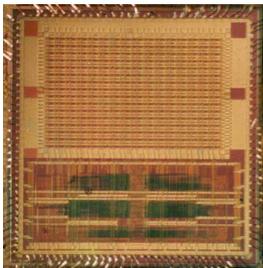
IBM Cell: 8 slave cores
+ 1 master core, 2005



AMD Phenom:
4 cores, 2007



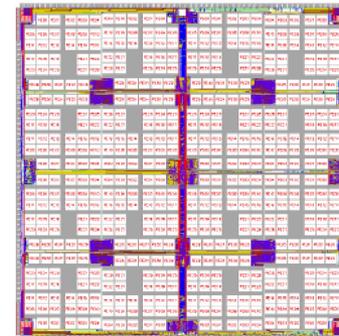
Intel Dunnington: 6 cores, 2008



Kilocore: 256-core prototype
By Rapport Inc.



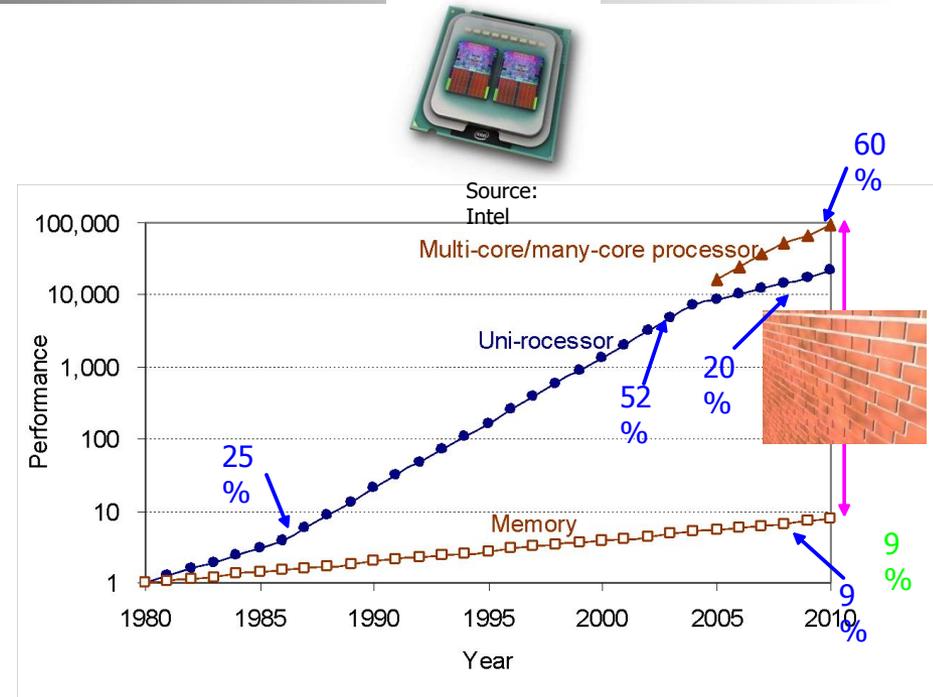
Quadro FX 3700M:
128-core, By nVIDIA



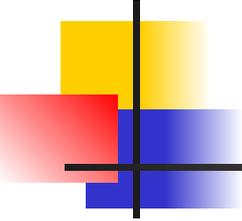
GRAPE-DR chip:
512-core, By Japan

Processor-memory Performance Gap

- Processor performance increases rapidly
 - Uni-processor: ~52% until 2004, ~25% since then
 - New trend: multi-core/many-core architecture
 - Intel TeraFlops chip, 2007
 - Aggregate processor performance much higher
- Memory: ~9% per year
- Processor-memory speed gap keeps increasing



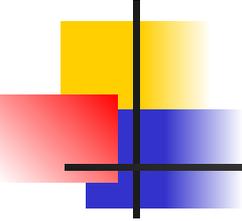
Source: OCZ



Motivation

- Data access is the problem
- Multi-core has changed the hardware structure
- Multi-core still uses single-core memory scheduling schemes
- Requires a rethinking of data access scheduling

Core-aware scheduling



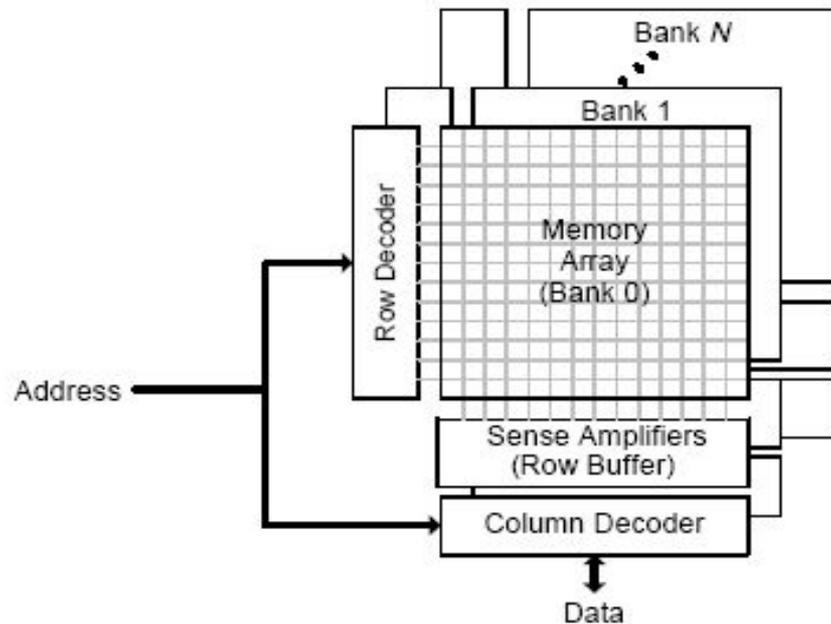
Outline

- 1 Conventional Memory access scheduling**
- 2 Core-aware memory access scheduling**
- 3 Performance evaluation and analysis**
- 4 Conclusion**

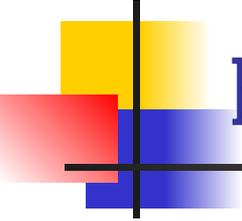
Memory Access

DRAM architecture: bank, row, column

- Three dimension
- Steps of accessing
 - pre-charge
 - row access
 - column access
- Faster
 - different bank
 - Same row



DRAM architecture



Existing

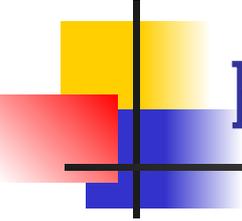
Memory Access Scheduling

Bank-first scheduling (*Rixner et al.*) : memory operations to different banks are allowed to proceed before those to the same bank

(Standard benchmark)

Row-first scheduling (*Shao et al.*) : memory operations are clustered into bursts that would access the same row within a bank

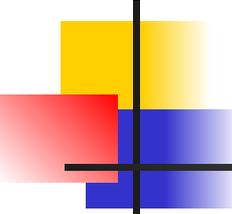
(Most advanced)



Multi-Core Memory Access

many cores are integrated into the same microprocessor, the number of memory requests increases abruptly

Competition for data access and transferring data among cores may increase the stall time and the length of the waiting queue



Core-aware Memory Scheduling

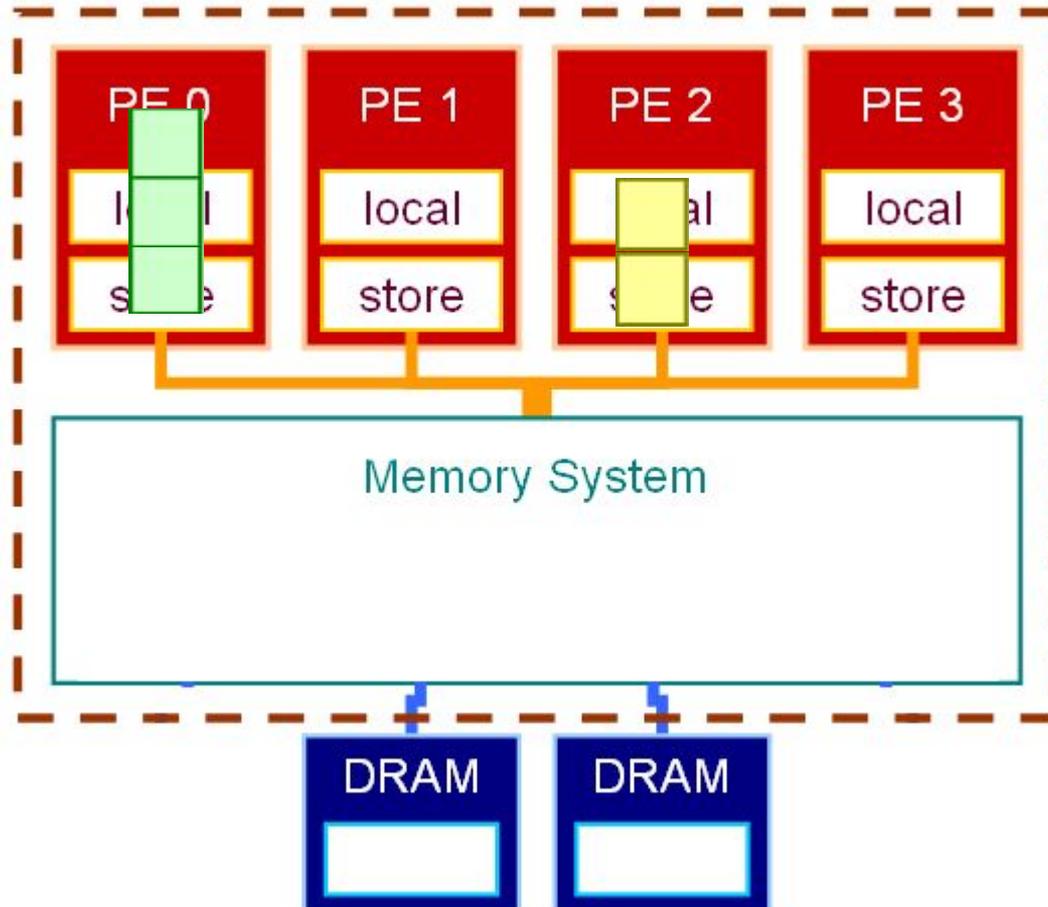
Idea: Scheduling scheme should consider the source of memory access

Reason: Requests from the same core can be combined and have a better locality; reading into the same core is faster

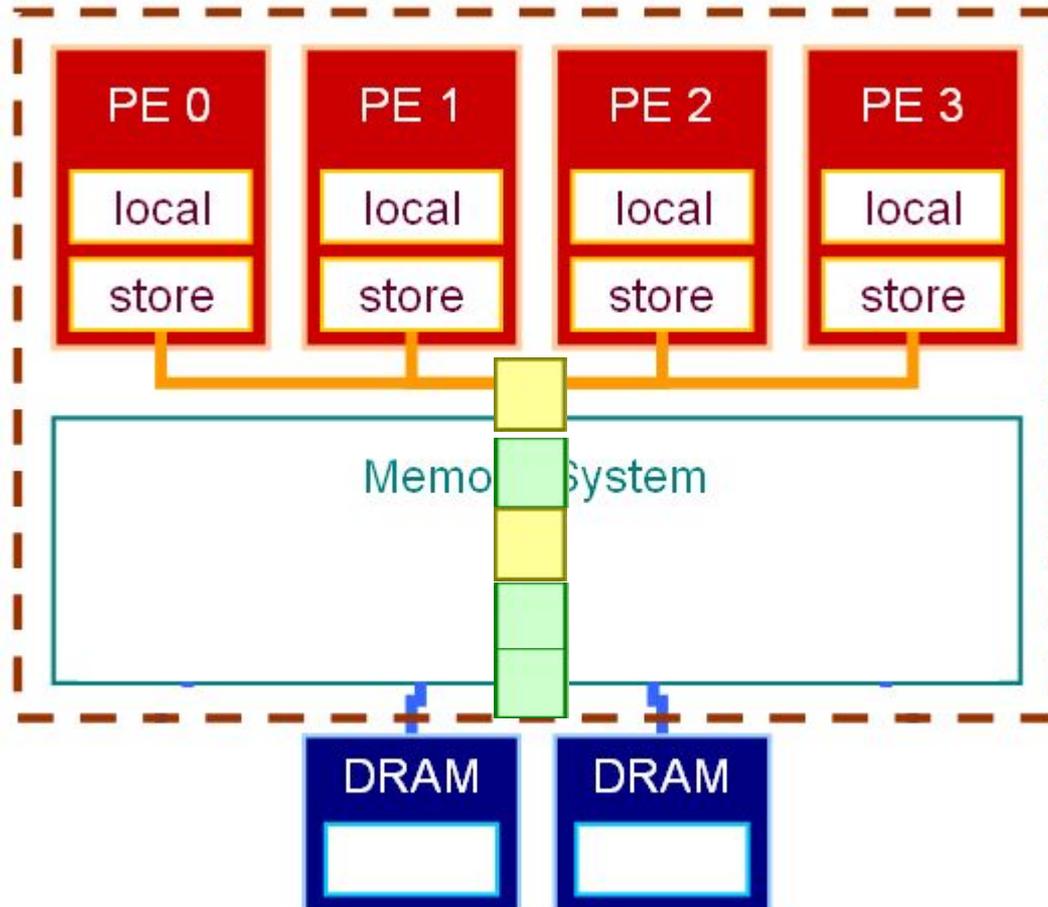
Feasibility: Core id - the lowest address bits of a cache request

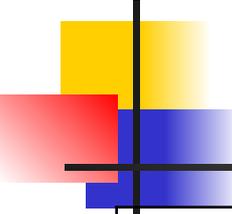
Solution: Issues outstanding requests from the same core together

Core-aware memory scheduling



Core-aware memory scheduling





Core-aware Scheduling Algorithm

ALGORITHM: CASA /*core-aware scheduling algorithm*/

INPUT: Random sequence of memory access requests from m cores

OUTPUT: Core-aware scheduled sequence of requests to memory controller

BEGIN

$k \leftarrow 0$; /* k indicates the core id */

While true

$n \leftarrow q$; /* q is the number of requests to be scheduled */

succeed $\leftarrow 0$;

Repeat

$cid \leftarrow k \bmod m$;

Select $s = \min(p, \text{number of outstanding requests from core } cid)$ requests from core cid , and enqueue them to the issue_queue; /* p is the size of issue_queue */

$n \leftarrow n - s$;

If ($n = 0$ OR ($k \bmod m = 0$)) **Then** succeed $\leftarrow 1$; /* 2nd condition prevents starvation and guarantees requests are issued within at most one iteration */

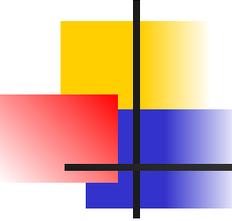
$k \leftarrow k + 1$;

Until succeed = 1

End While

END

Core-Aware Scheduling Algorithm



Core-aware Scheduling Example

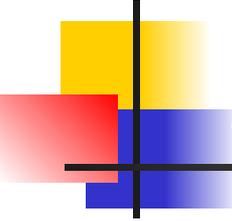
Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

Bank-first scheme sequence: A-C-D-F-E-B-G-H-J-I

Row-first scheme sequence: A-B-J-C-D-G-I-F-H-E

Core-aware bank-first scheme sequence: A-C-D-F-E-J-I-H-B-G

Core-aware row-first scheme sequence: A-J-B-C-D-I-G-F-H-E

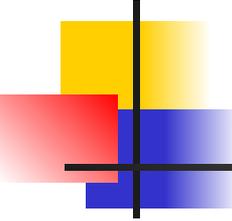


Bank-first scheme

Example:

Sequence	A	B	C	D	E	F	G	H	I	J
Bank	①	①	②	③	⑤	④	③	④	③	①
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

Scheduled sequence: A-C-D-F-E-B-G-H-J-I

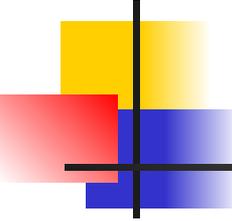


Row-first scheme

Example:

Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

Scheduled sequence: A-B-J-C-D-G-I-F-H-E

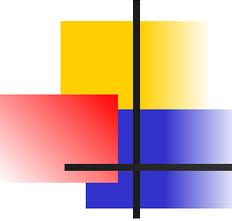


Core-aware bank-first scheme

Example:

Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

Scheduled sequence: A-C-D-F-E-J-I-H-B-G



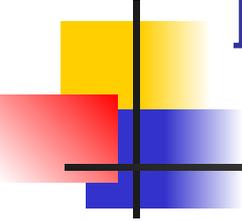
Core-aware row-first scheme

Example:



Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1

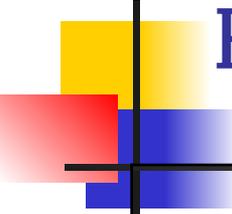
Scheduled sequence: A-J-B-C-D-I-G-F-H-E



Performance Evaluation and Analysis

- ❑ Simulator
 - ❑ Simics and GEMS (General Execution-driven Multiprocessor Simulator)
 - ❑ Enhanced the current GEMS
 - ❑ Set the threshold for one core as 16

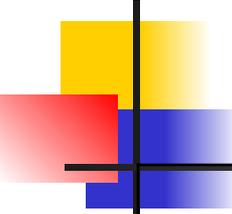
- ❑ Benchmark
 - ❑ NAS Parallel Benchmarks
 - ❑ Five kernels: EP, DC, CG, MG, FT



Performance Evaluation and Analysis

Component	Parameters
CPU	16 Sun SPARC processor cores, each core is 2GHz 4-way issue
L1 I-cache	16 KB, 4-way L1 cache on each core, 64 bytes cache line
L1 D-cache	16 KB, 4-way L1 cache on each core, 64 bytes cache line
L2 cache	256 KB, 4-way cache on each core, 64 bytes cache line
Cache Coherence protocol	Directory and MESI protocol
FSB	64 bit, 800MHz (DDR)
Main Memory	4GB DDR2 PC2 6400 (5-5-5), 64 bit, burst length 8 Memory page is 4 KB
Channel/Rank/Bank	2/4/4 (a total 32 banks)
SDRAM Row Policy	Open Page
Address Mapping	Page Interleaving
Memory Access Pool	32 queues for each bank, each queue size is 16 entries
OS	Solaris 10

**Machine Configuration on Simics/GEMS
simulator**



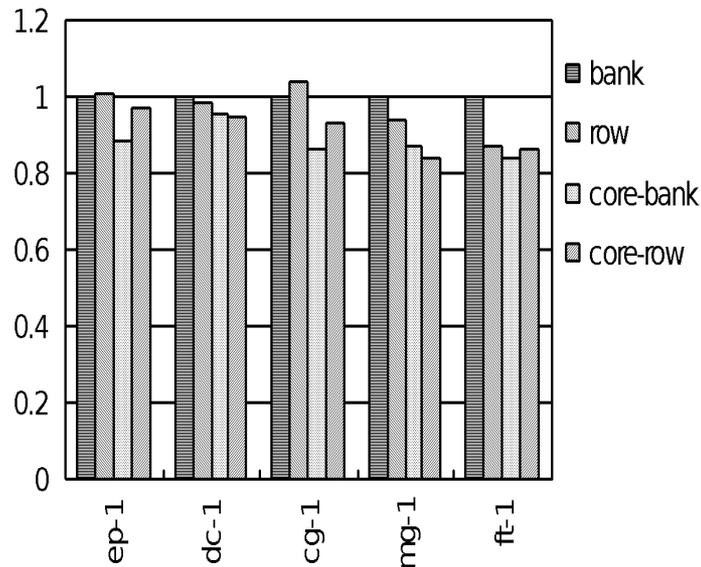
Performance Evaluation and Analysis

Set	Benchmarks
Single application with single thread	ep-1: EP running with one thread dc-1: DC running with one thread cg-1: CG running with one thread mg-1: MG running with one thread ft-1: FT running with one thread
Single application with multiple threads	ep-4: EP running with four threads dc-4: DC running with four threads cg-4: CG running with four threads mg-4: MG running with four threads ft-4: FT running with four threads
Multiple applications	mix-1: dc-1, cg-1,mg-1 and ft-1 running concurrently mix-4: dc-4, cg-4,mg-4 and ft-4 running concurrently

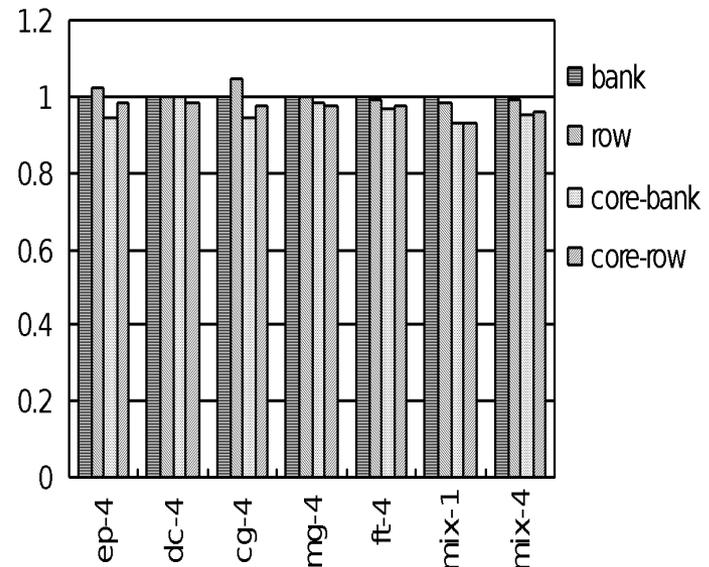
Experiment Configuration

Simulation Results

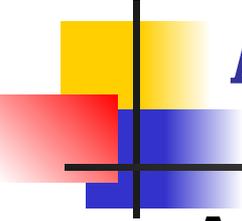
Number of memory requests



Single application with single thread. All values are normalized to bank-first scheme.



Single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.



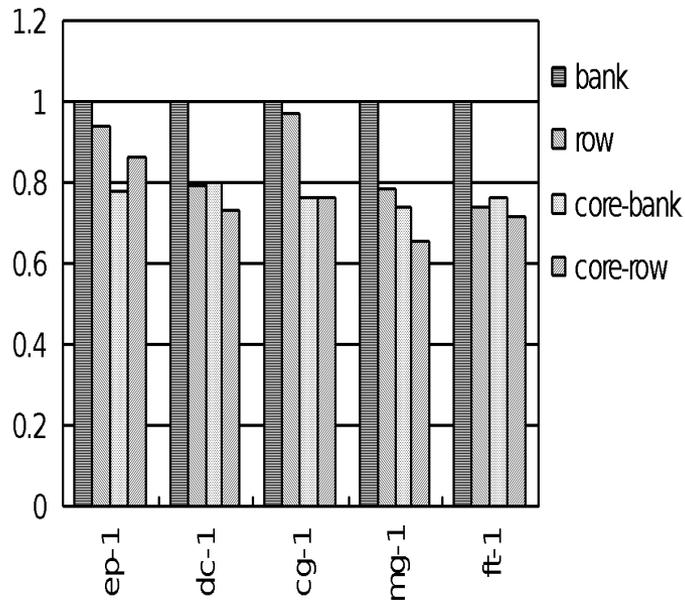
Analysis

Analysis of the number of memory requests

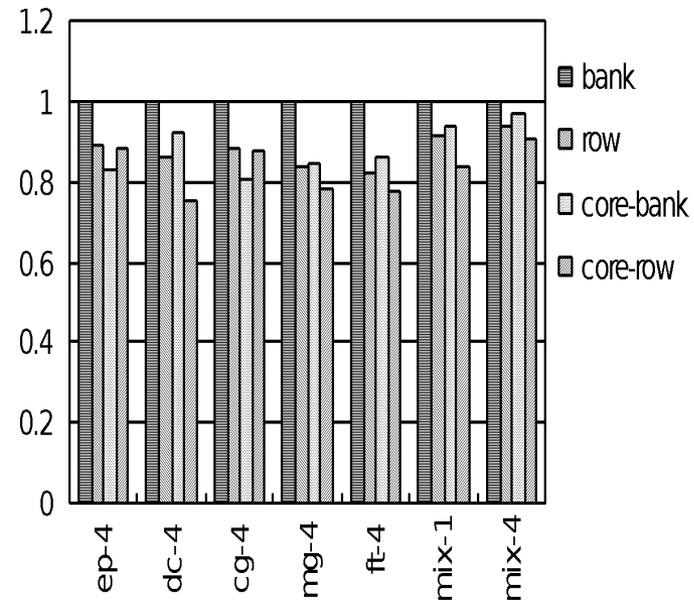
Compared with row-first scheme, the core-aware row-first scheme reduced memory requests by 5% on average for five benchmarks running with four threads, as shown in. This is mainly due to the core-aware row-first scheme improving the row-first policy by taking the request source into consideration.

Simulation Results

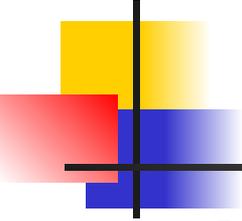
Waiting latency



Single application with single thread. All values are normalized to bank-first scheme.



Single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.



Analysis

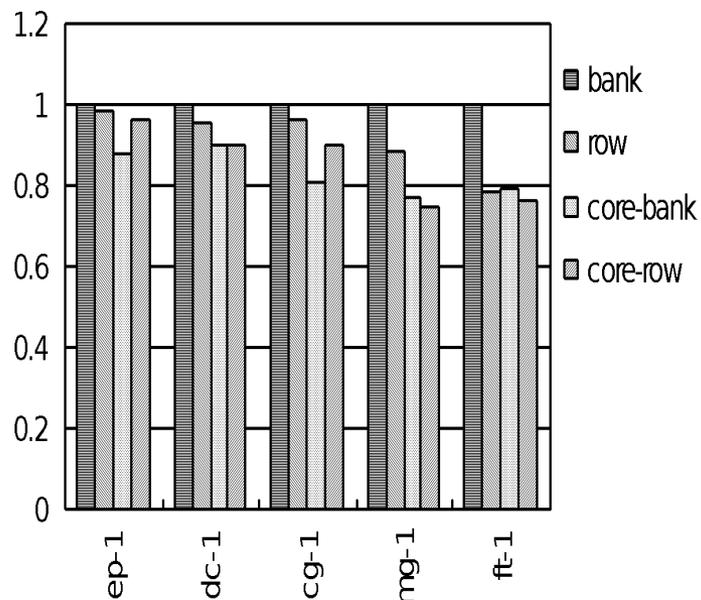
Analysis of waiting latency

When DC, MG and FT ran with four threads, the core-aware row-first decreased the latency by up to 25%, 22% and 23% compared with the bank-first scheme, and by up to 10%, 5% and 5% compared with the row-first scheme.

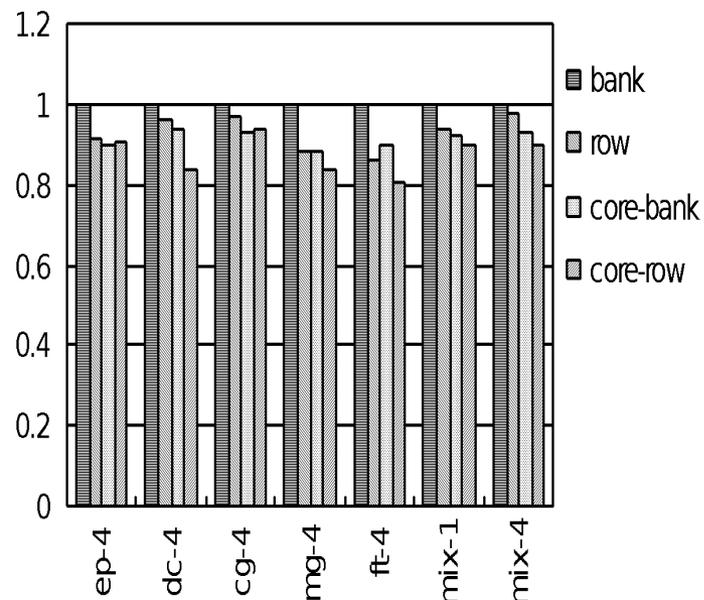
Intuitively, row-first already gets the locality. But, core-aware reduces the number of requests by 5% and 6% therefore reduced the waiting latency.

Simulation Results

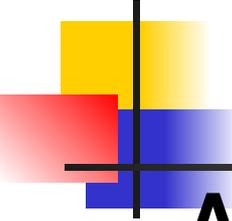
Execution time



Single application with single thread. All values are normalized to bank-first scheme.



Single application with multiple threads and multiple applications. All values are normalized to bank-first scheme.

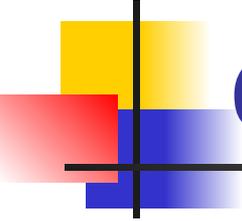


Analysis

Analysis of execution time

Core-aware improves the performance for all five benchmarks. Compared with the bank-first and row-first policy, the core-aware row-first policy reduced the execution time by 14% and 5% on average for all five benchmarks running with four threads, and by 10% and 5% on average for with multiple applications.

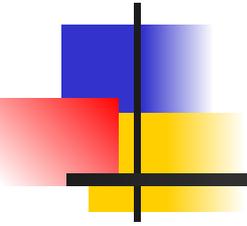
The performance improvement of one thread is better than that of four threads. We believe this is caused by OS noises.

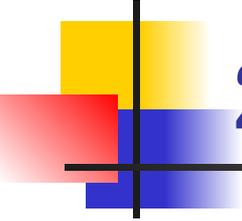


Conclusion

- The source core of memory requests is an important factor in scheduling data access
- Experimental results confirmed that the proposed core-aware scheduling schemes decreased the latency considerably
- Actual results could be even better than the simulated ones
- The core-aware scheduling is simple in both hardware and software, and the gain is big. It has a real commercial value.

Thanks !





2、Core-aware memory scheduling

Example:

Sequence	A	B	C	D	E	F	G	H	I	J
Bank	1	1	2	3	5	4	3	4	3	1
Row	1	1	2	1	3	4	1	4	1	1
Core	1	2	1	2	1	3	1	1	2	1