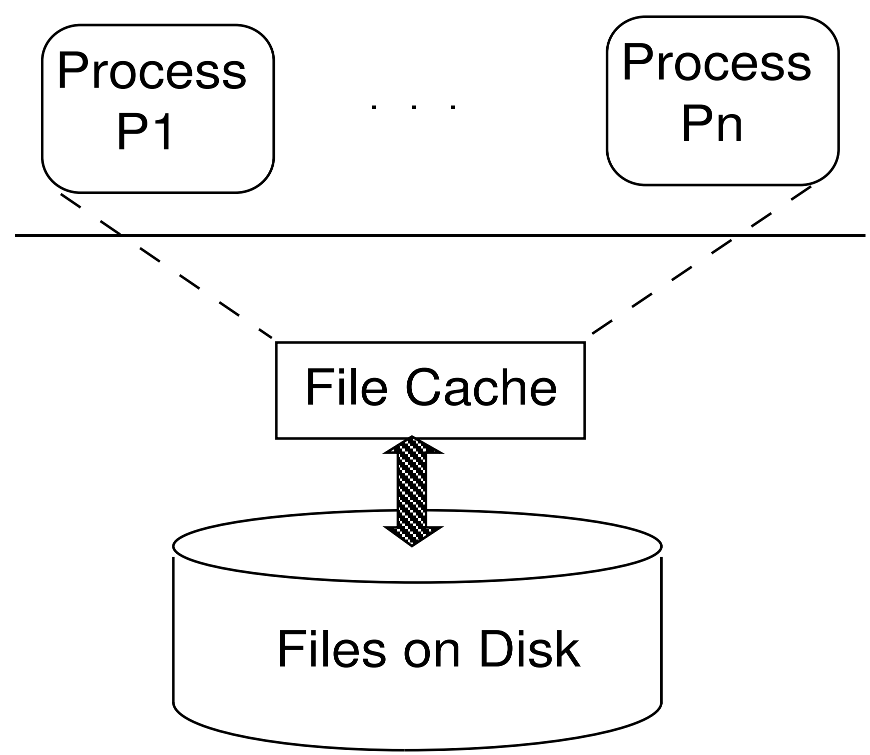
Application-Controlled File Caching and Prefetching

Pei Cao

Department of Computer Sciences
University of Wisconsin-Madison

The Increasing Performance Gap

Approach to Bridge the Gap



Improving File Cache Performance

- Application control on replacement
- Prefetching
- Disk scheduling

- Others: write-back policies, clustering, etc.

Research Problems

Application-controlled caching

- Good cache allocation policy that is no worse than existing scheme

Prefetching

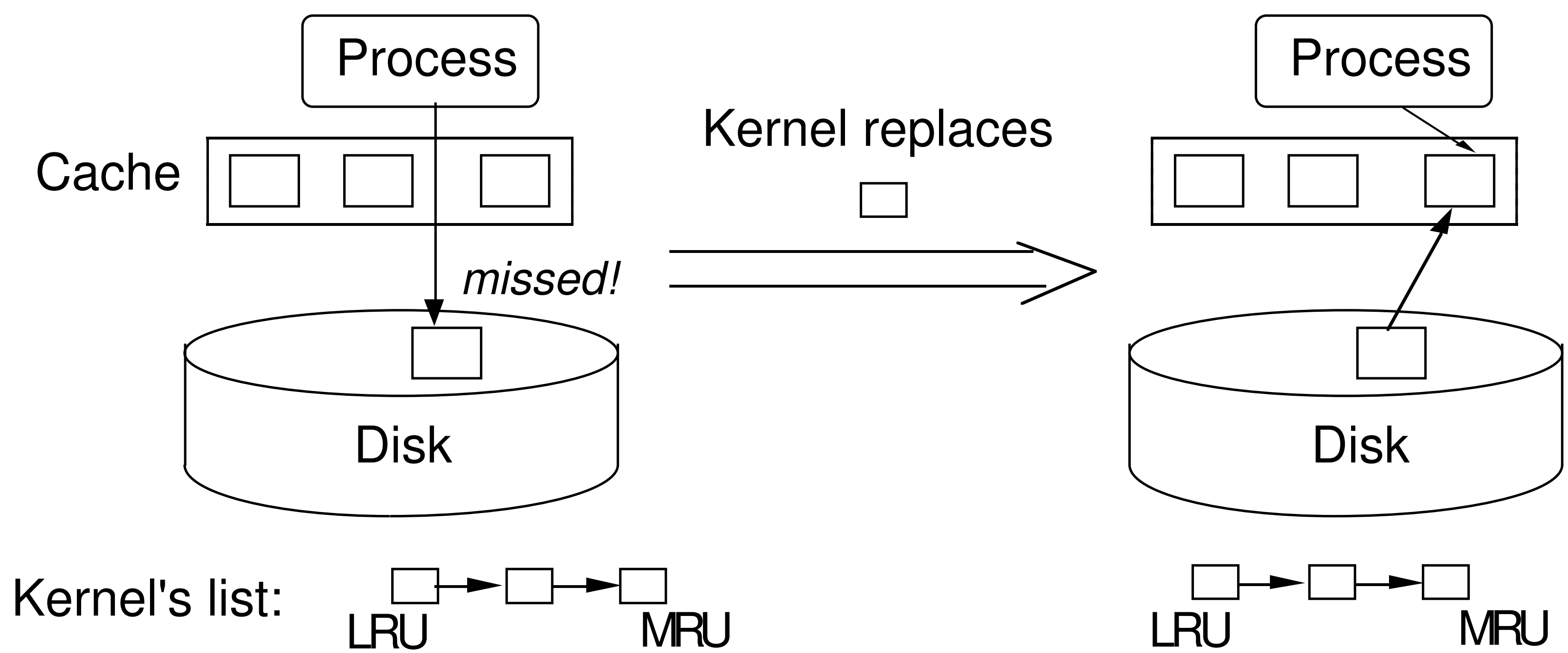
- Integration with caching and disk scheduling

Part I: Application Controlled Caching

- Why application control
- Allocation policy: LRU-SP
- Implementation and performance results

Traditional Scheme

Kernel controls replacement, uses global LRU



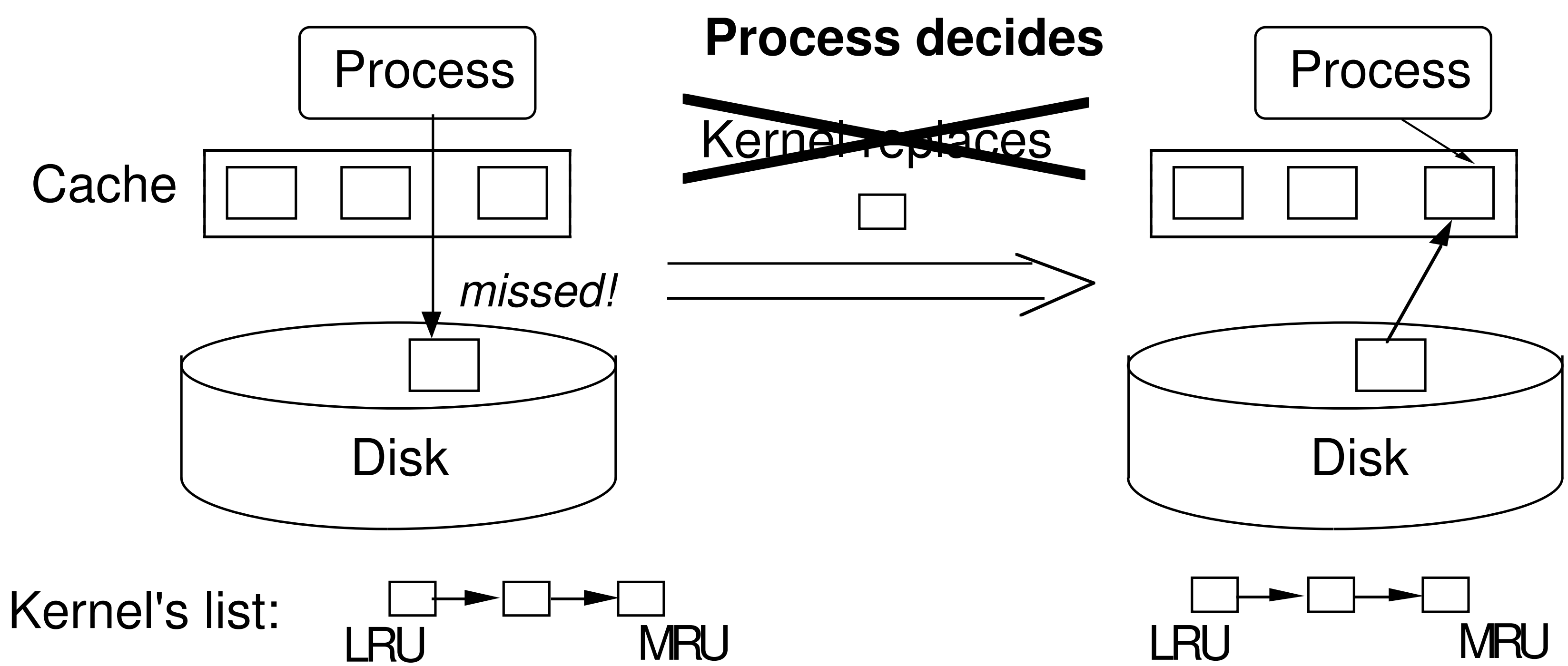
Why Application Control

"One policy fits all"

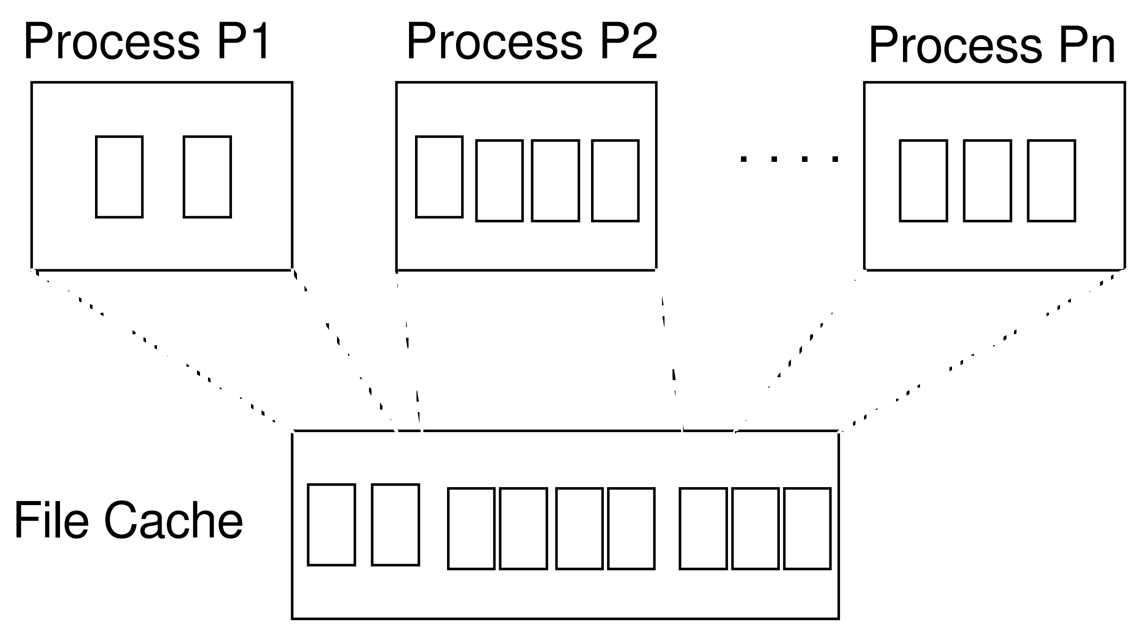
- Kernel policy is not always the best

- Applications often know better than kernel
  - Database systems, text information retrieval tools, multimedia, out-of-core computation, trace-driven simulations, ...

Application Control

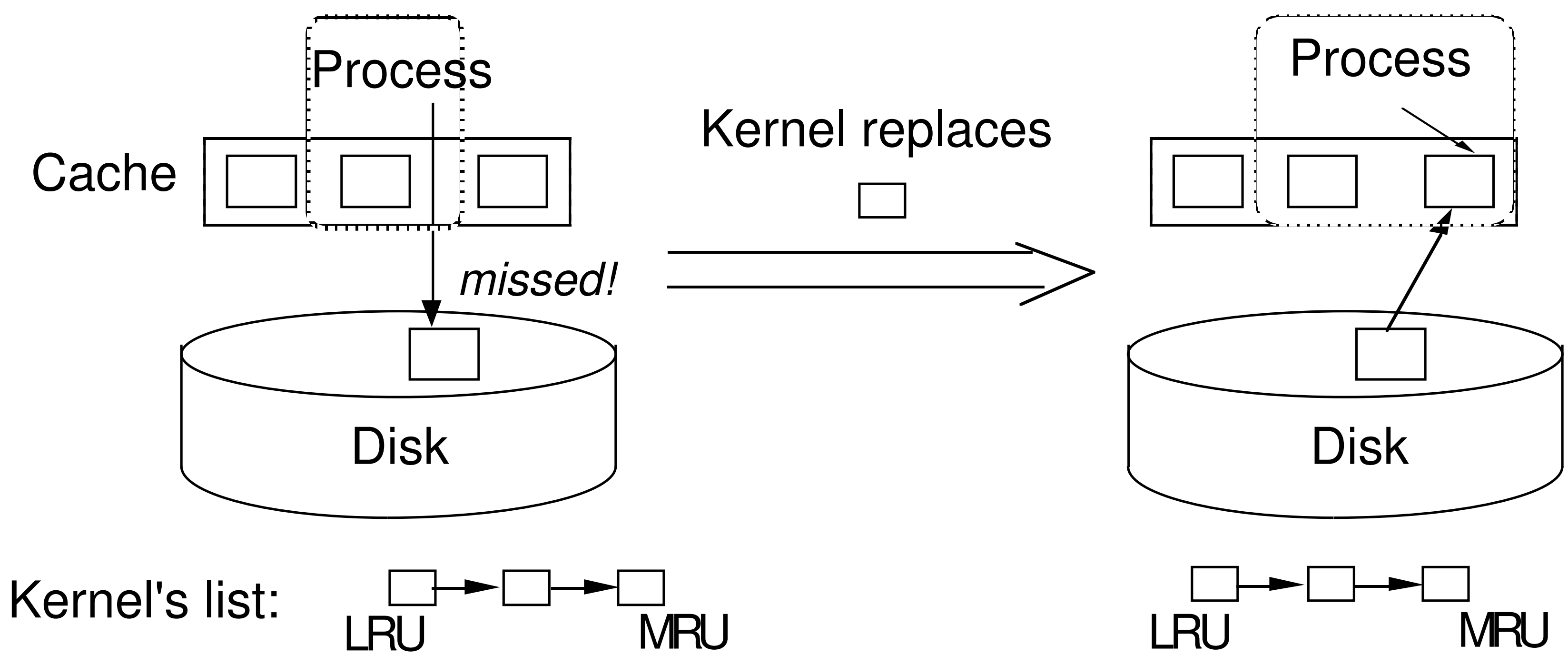


Problem: Cache Allocation

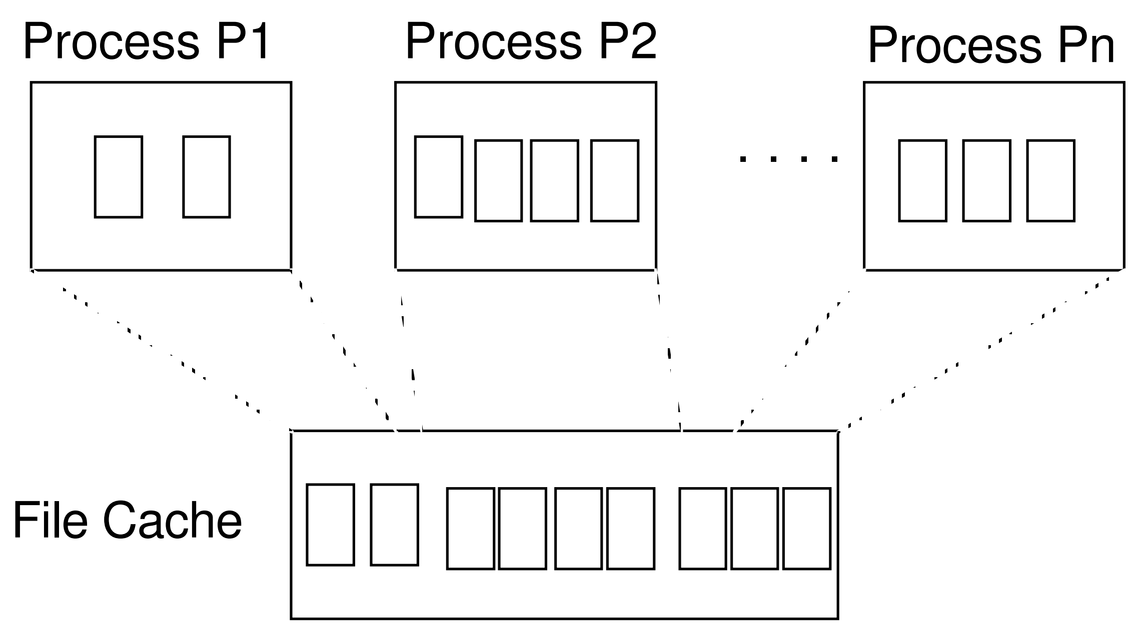


Traditional Scheme

Kernel is also doing allocation



Problem: Cache Allocation



Two Level Replacement

Kernel manages allocation

☞ – Kernel decides which <u>process</u> should give up a cache block upon miss

User processes manage replacement

– Processes decide which <u>block</u> to replace when asked by kernel

Criteria of Allocation Policies

Compared with the existing scheme

- **Processes that <u>choose to let kernel control</u> caching <u>do no worse</u>**
- **Processes that use <u>smart</u> replacement policies <u>benefit</u>**
- **Processes that use <u>stupid</u> replacement policies <u>cannot hurt others</u>**

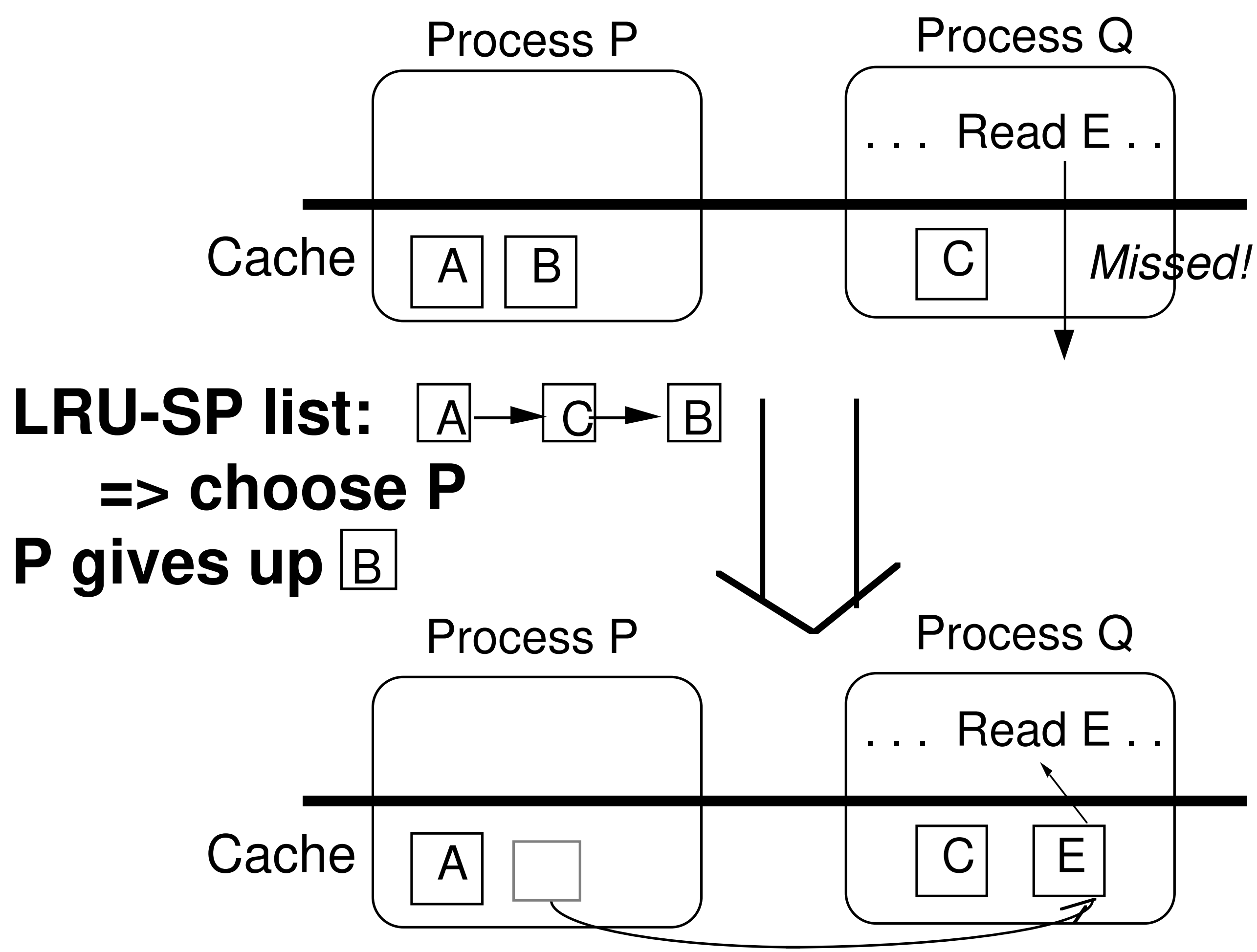
Our Allocation Policy: LRU-SP

Proven to satisfy the criteria

– under the assumption that the interleave of file accesses from processes is not affected by replacement decisions

- Maintains an “LRU” list of all cache blocks
- On a miss, chooses the process who owns the “LRU” block
- Makes two modifications to the “LRU” list: swapping & placeholders

LRU-SP

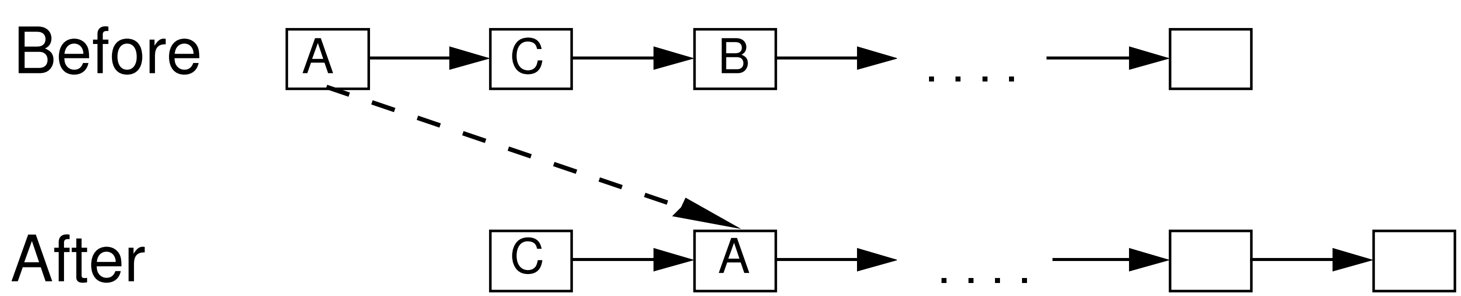

Process P
Process Q
. . . Read E . .
Cache
A
B
C
Missed!
LRU-SP list:
A
C
B
=> choose P
P gives up B
Process P
Process Q
. . . Read E . .
Cache
A
C
E

Swapping

Prevent discrimination against smart processes

A and B belong to process P, and A is the globally "LRU" block, but P replaces B

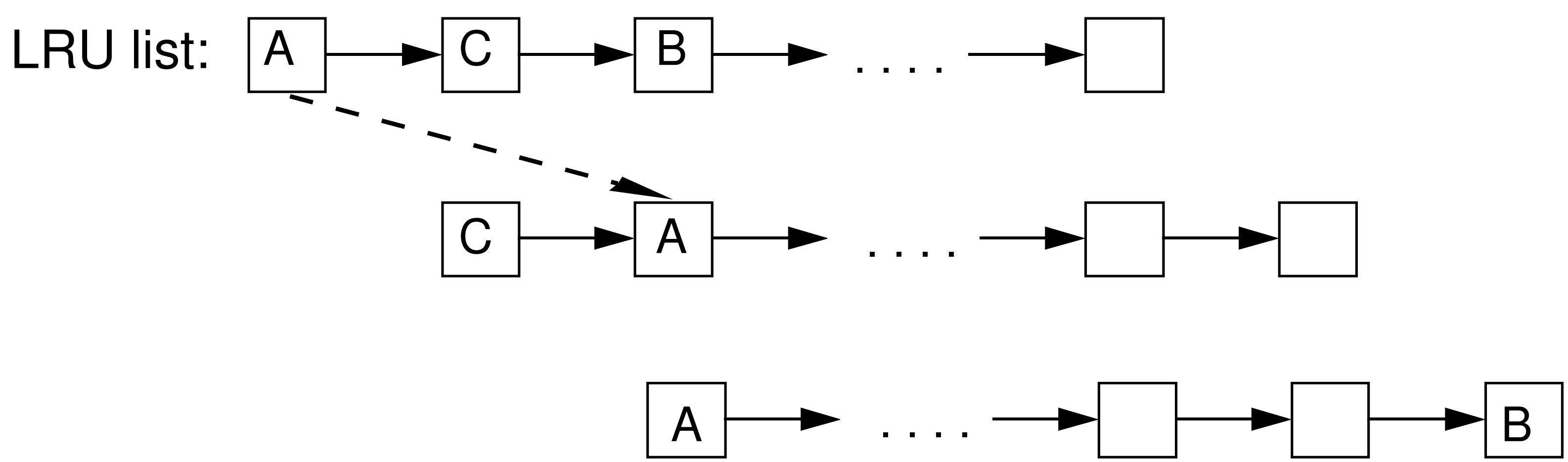
Swapping: Kernel puts A in B's position

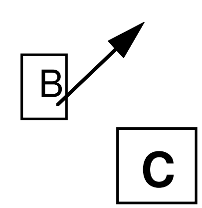


Placeholder

Prevent foolish processes from hurting others

Suppose P was foolish:
after replacing B, B is accesed next




B
C

LRU-SP

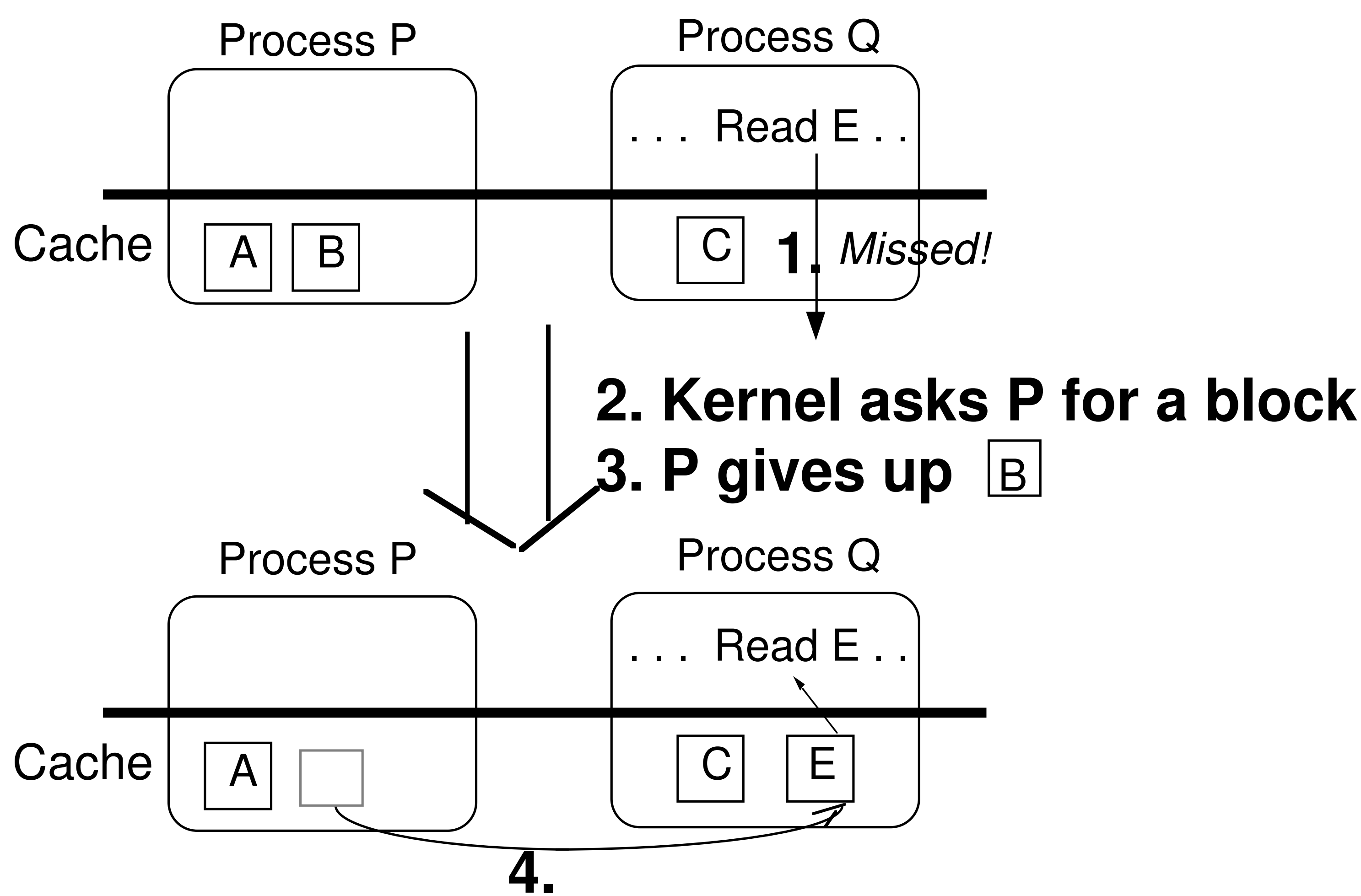
Proving that it satisfies the criteria:

- Swapping & placeholders make LRU-SP list mirror the LRU list of no application control
- Makes the same allocation as the traditional scheme most of the time

Part I: Application Controlled Caching

- Why application control
- Allocation policy: LRU-SP
- ☞ Implementation and performance results

Steps in Two Level Replacement



Design Alternatives

How to implement the user/kernel interaction?

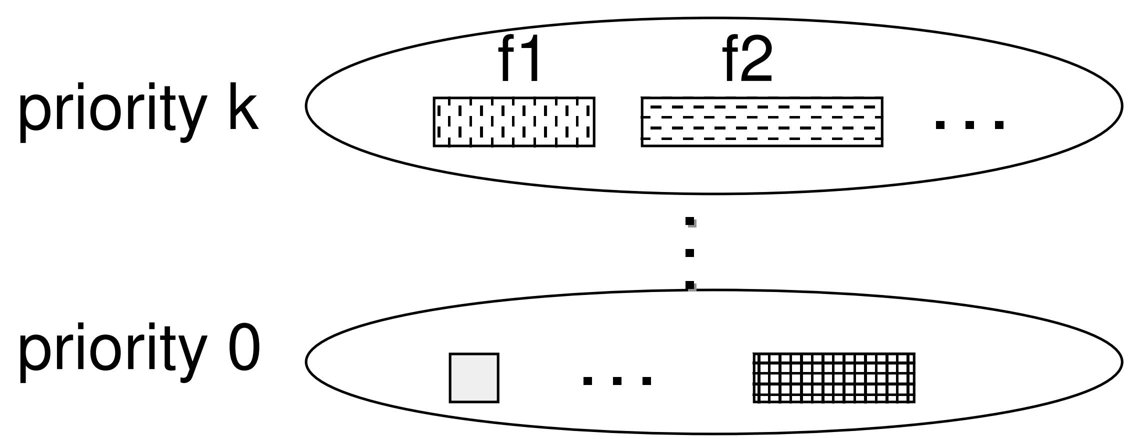
Options:

- **“Upcalls” --- expensive**
- **Fixed menu of policies --- inflexible**
- **Priorities of file blocks --- high overhead**
- ☞ **Combination of priorities and policies**

The Interface: Priorities + Policies

Applications can

- **Assign <u>priorities</u> to files and file blocks**
- **Specify <u>replacement policies</u> for each priority**
  (currently: LRU or MRU)



<u>Note</u>: priorities and policies are within a process

Examples

“Postgres”:

- Index blocks: priority 1 with LRU
- Data blocks: priority 0 with LRU

“Dinero”:

- Trace file: priority 1 with MRU

Implementation

- **“ACM”: ~ 3000 lines**

Experiments

Six applications: cscope, dinero, glimpse, ld, postgres, sort

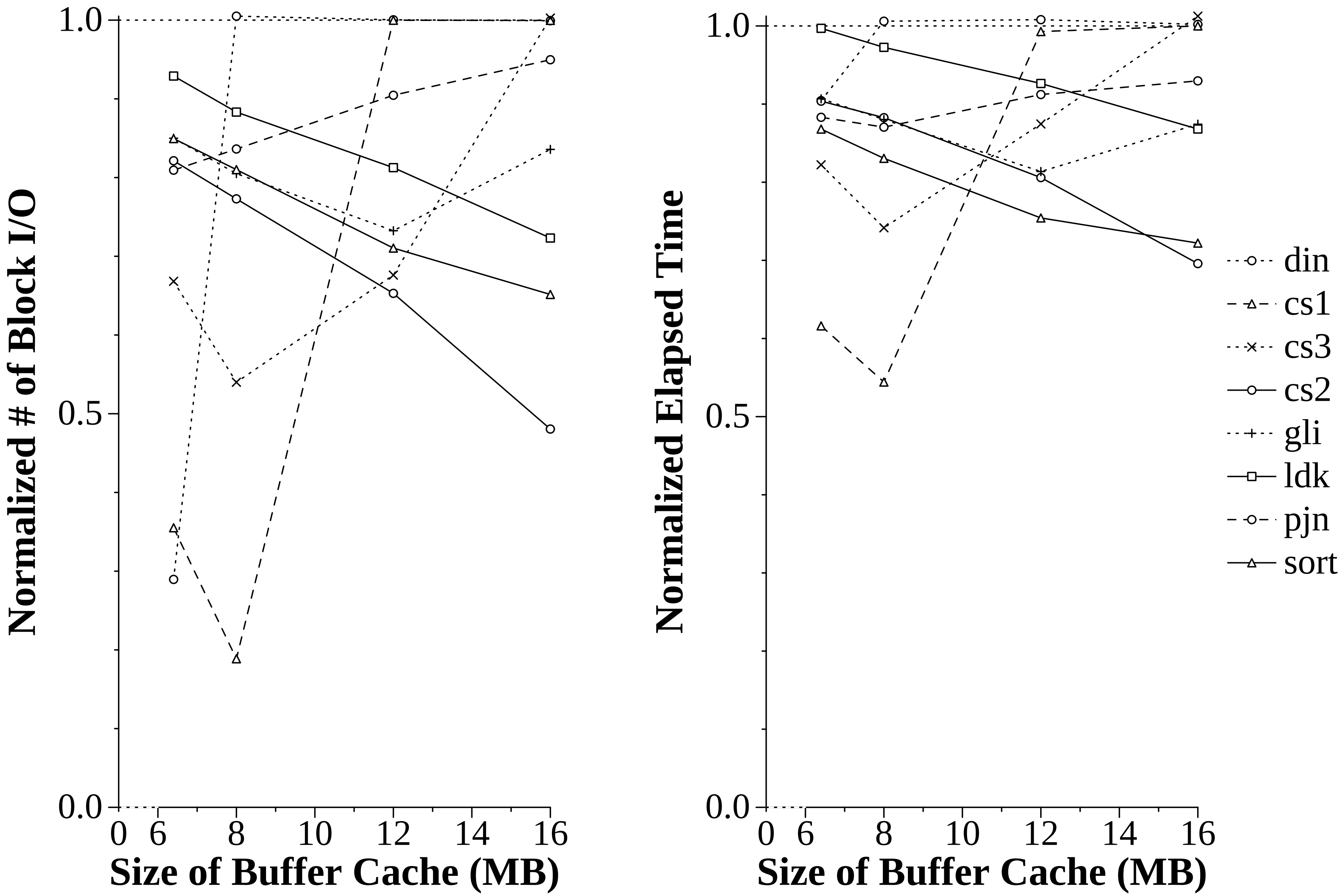
- Experiments: cs1, cs2, cs3, din, gli, ldk, pjoin, sort
- Four cache sizes: 6.4MB, 8MB, 12MB and 16MB
- Platform: DEC 5000/240

Results:

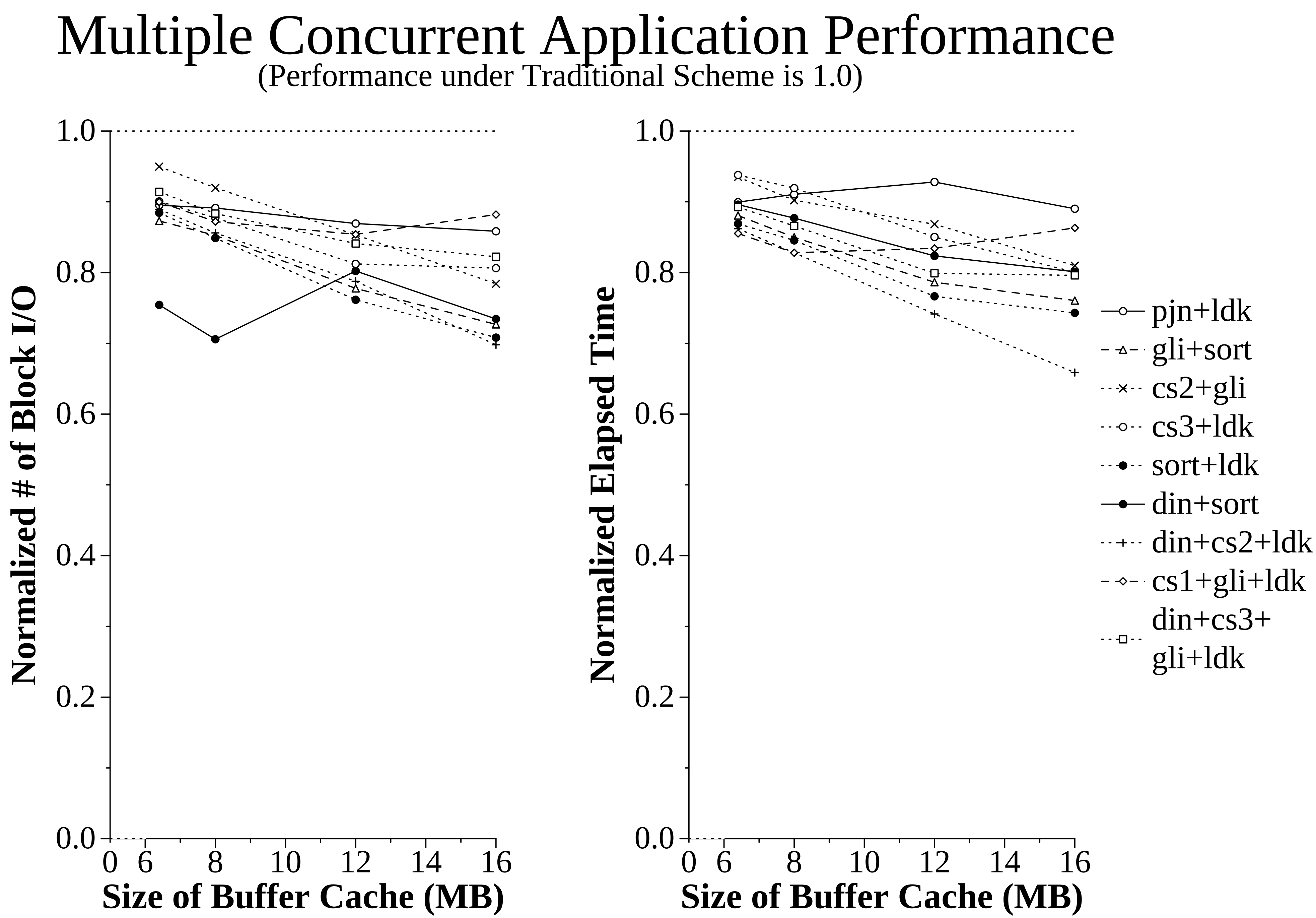
- Reduction in number of block I/Os (10–80%)
- Reduction in execution time (10–45%)

Single Application Performance

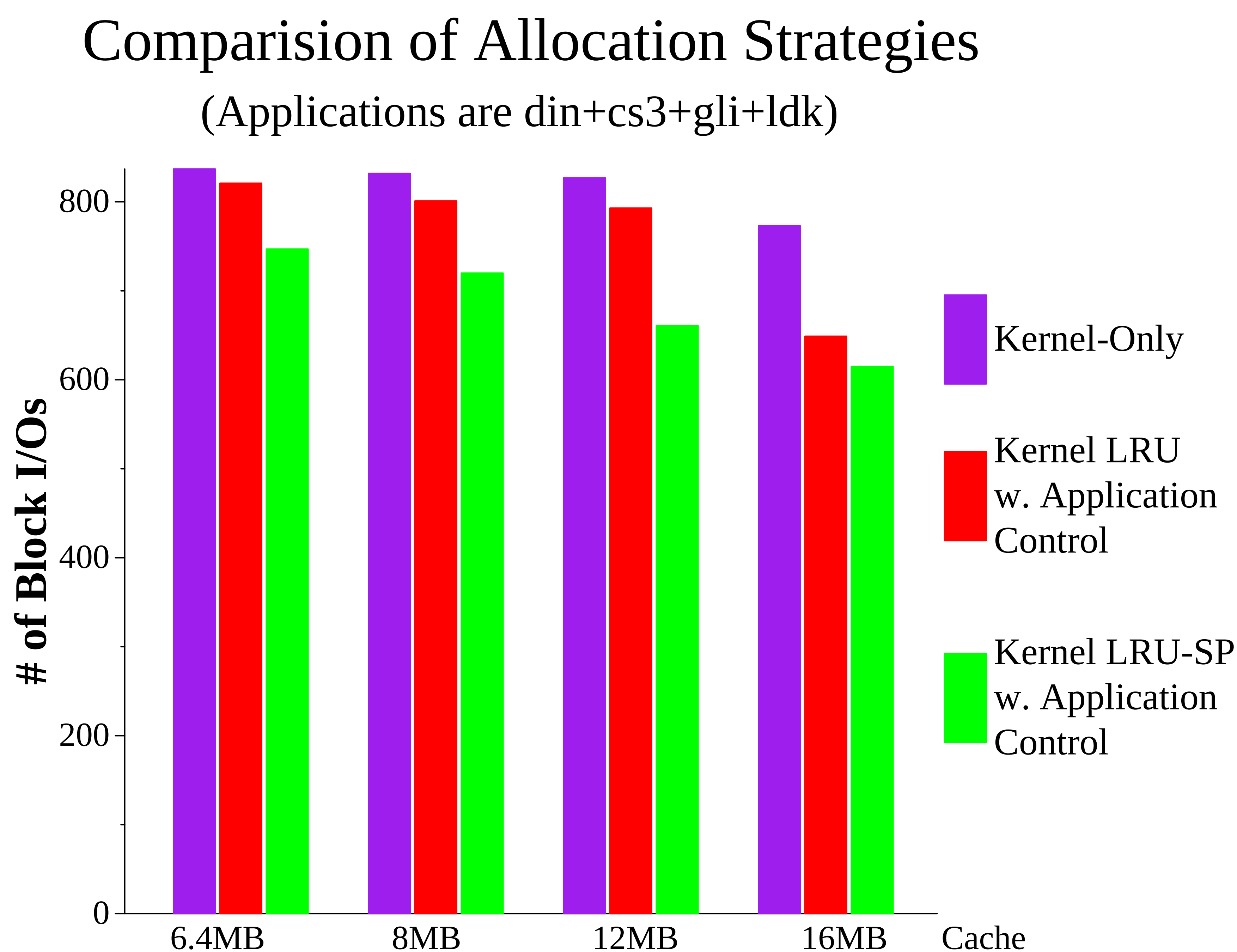
(Performance under Traditional Scheme is 1.0)



Does LRU-SP Work Well?

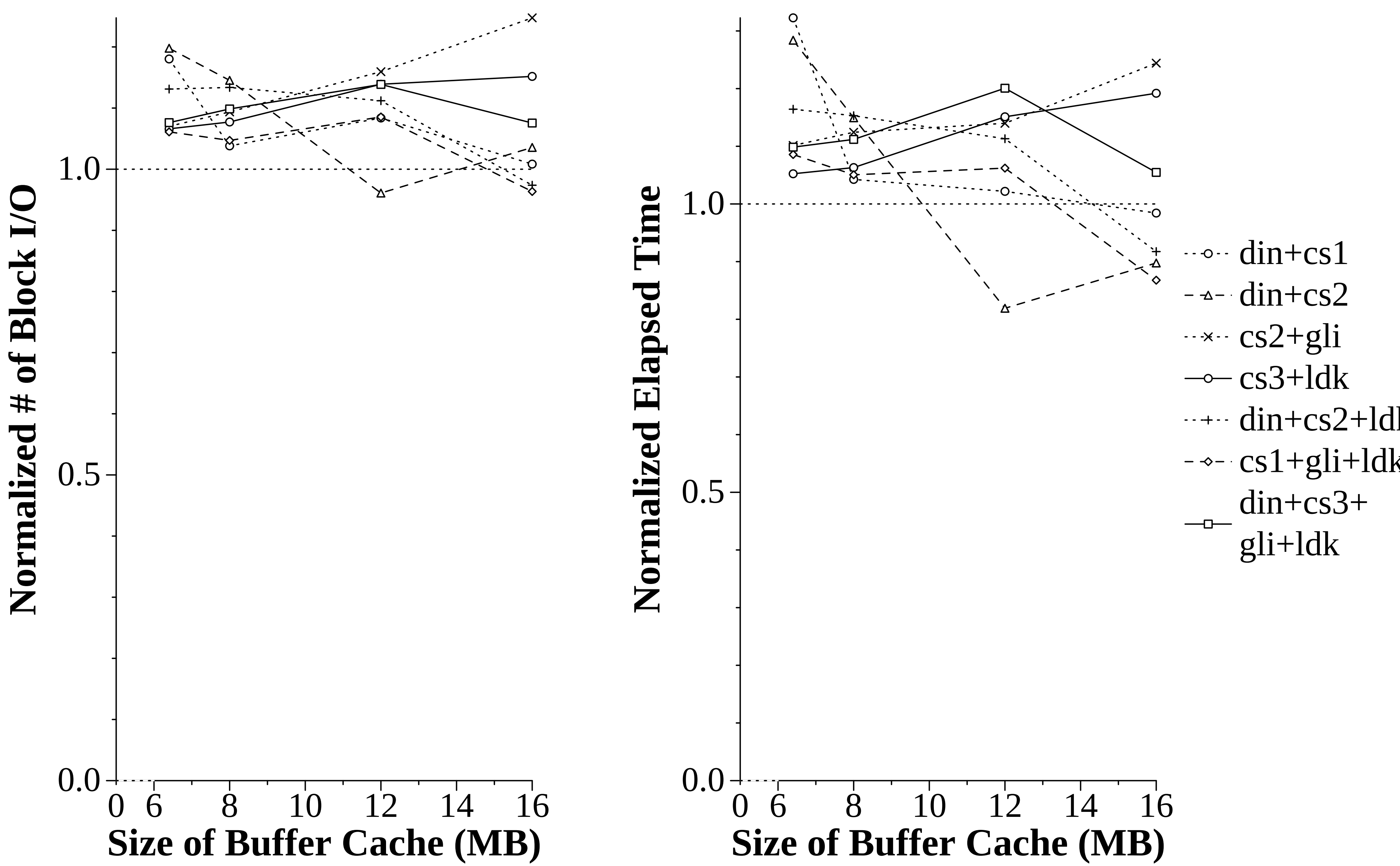

Multiple Concurrent Application Performance
(Performance under Traditional Scheme is 1.0)
Normalized # of Block I/O
Normalized Elapsed Time
Size of Buffer Cache (MB)
Size of Buffer Cache (MB)
0.0
0.2
0.4
0.6
0.8
1.0
0 6 8 10 12 14 16
pjn+ldk
gli+sort
cs2+gli
cs3+ldk
sort+ldk
din+sort
din+cs2+ldk
cs1+gli+ldk
din+cs3+
gli+ldk

Is LRU-SP Important?


Comparision of Allocation Strategies
(Applications are din+cs3+gli+ldk)
# of Block I/Os
800
600
400
200
0
6.4MB
8MB
12MB
16MB
Cache
Kernel-Only
Kernel LRU w. Application Control
Kernel LRU-SP w. Application Control

Comparing Allocation Strategies (SP vs no SP)

(Performance under LRU-SP is 1.0)



Part II: Prefetching

- Why prefetching
- The theoretical problem of integration
- Implementation and performance results

File Prefetching

Pre-fetching: "If you know about future accesses, you can fetch the block before the miss happens."

Why Prefetching

Easy to know about future accesses in many cases:

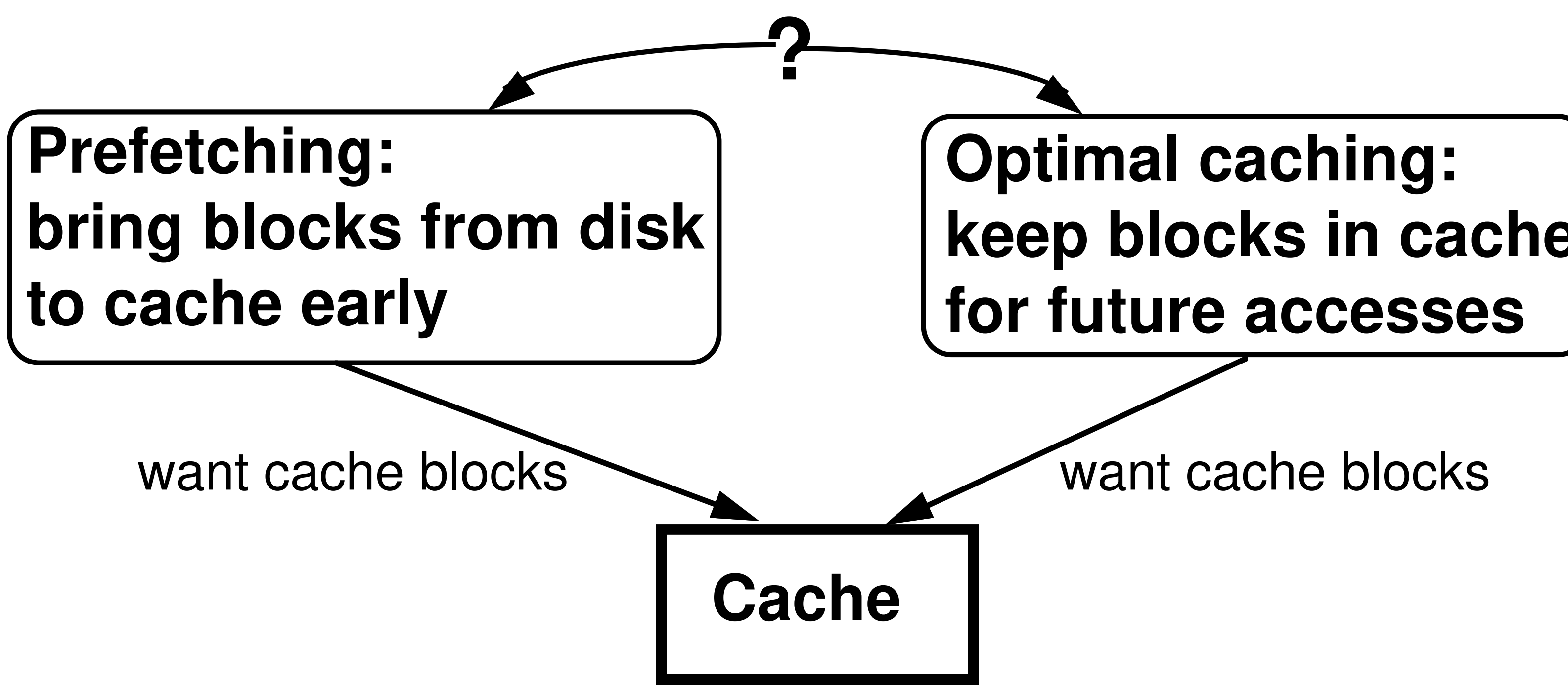
- Information from programs
  - e.g. grep, tar, mh, rn, ...
- Prediction by heuristics

Ways to use this knowledge:

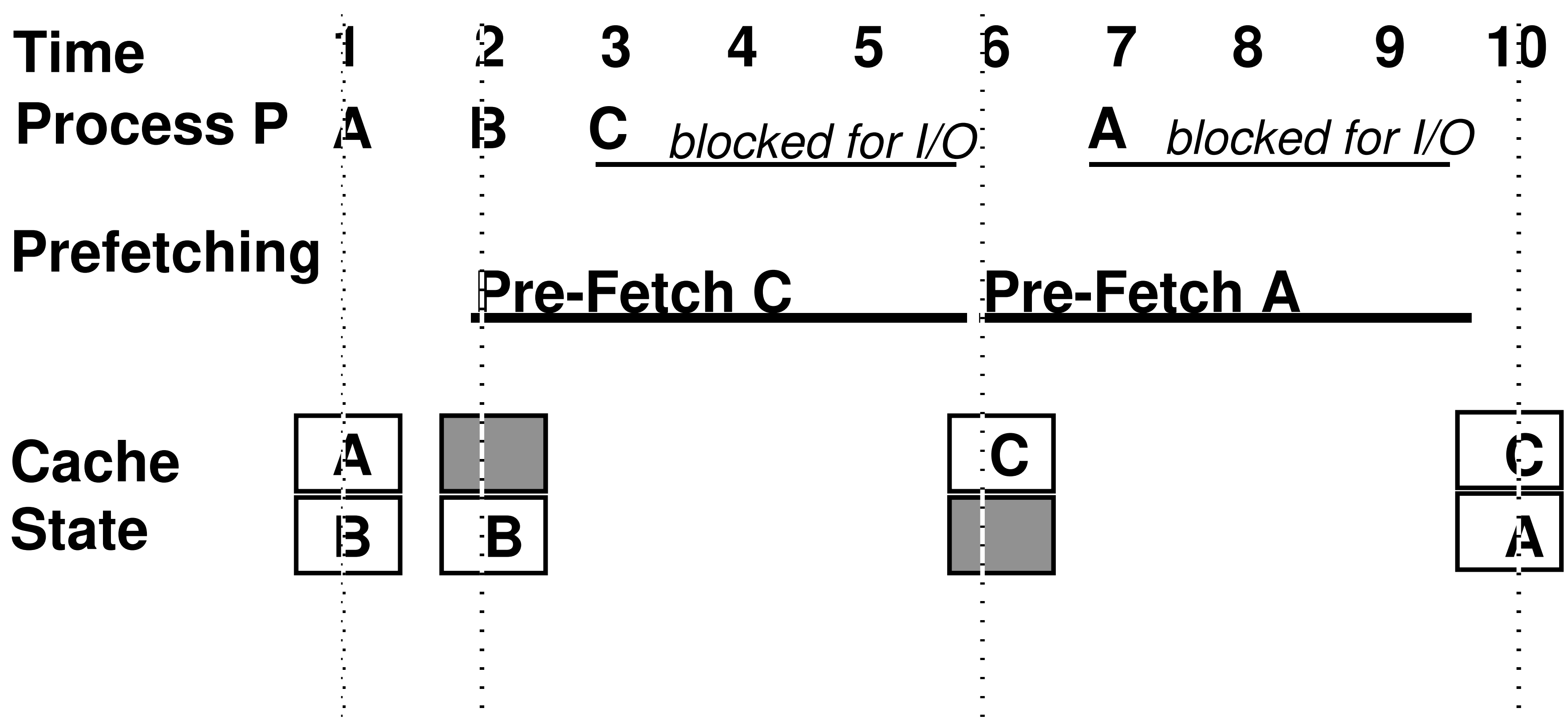
- Prefetching
- Optimal caching

File Prefetching (Cont')

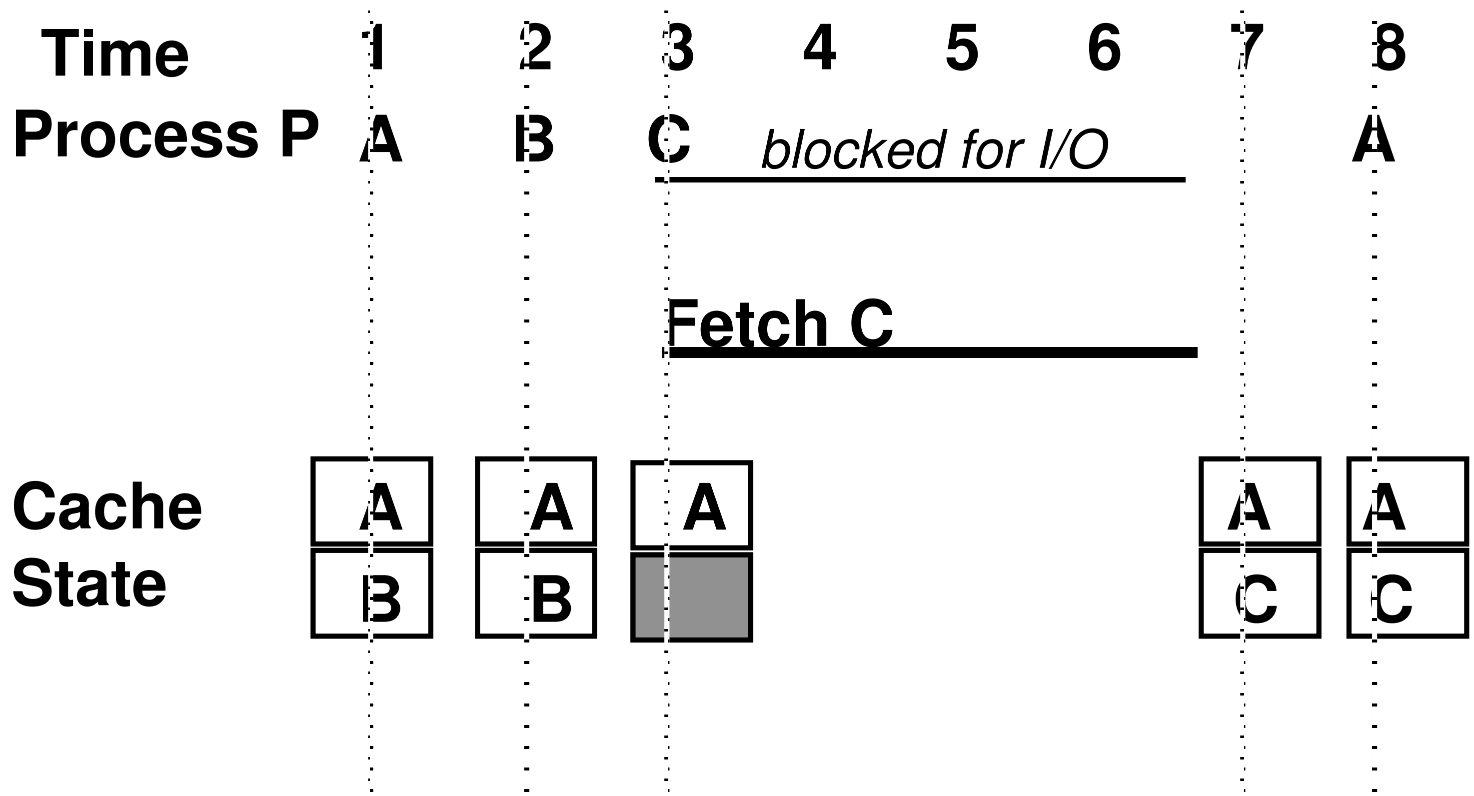
Knowing what process P's next accesses are:



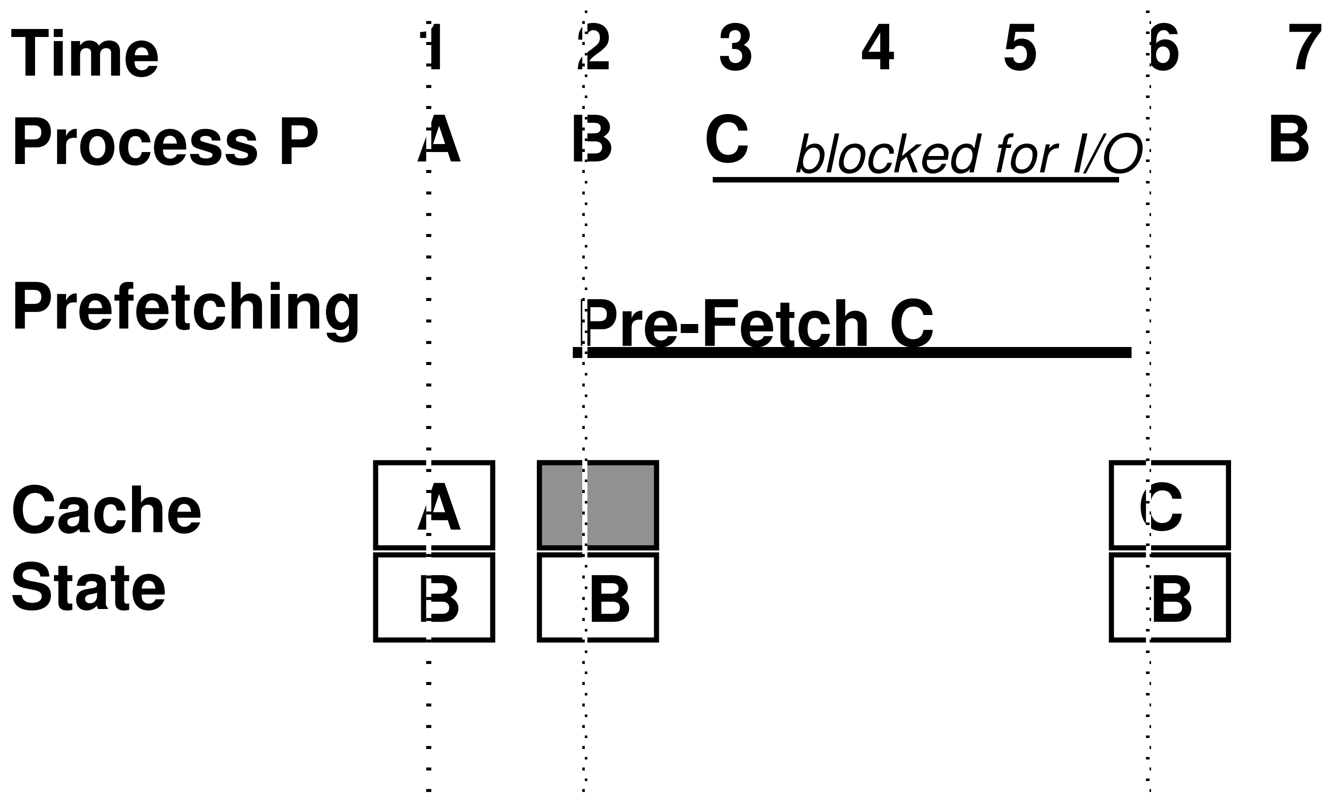
Prefetching



Optimal Caching



Prefetching



Tradeoff of Prefetching

- Overlaps CPU computation with disk I/O
- Earlier fetch

  => earlier replacement

  => worse replacement

  => more disk fetchs

Formulation of the Problem

Find an algorithm for the "cache manager":

- Knows:
  - <u>Future accesses</u>
  - Size of the cache: <u>K</u> blocks
  - Cost to fetch a block from disk: <u>F</u>
- Must decide:
  - when to fetch
  - fetch what
  - replace what

Goal: minimize <u>elapsed time</u>

Properties of Optimal Algorithms

1. Always fetch the next missing block
2. Always replace the block that will be referenced furthest in future
3. Never replace A to fetch B if A will be accessed before B
4. Do not delay a fetch unless waiting for better replacement

Controlled-Aggressive

- **Fetch at the earliest time**
  Allowed by the rules and
  When the disk is idle
- **Make optimal replacement**

Example:

P's next accesses: A, B, A, C, D, B, A, C
Cache has three blocks, initially A, B, C

A B A C D B A C

Approximation Ratio: 1+F/K

Theorem:

Elapsed time of Controlled-Aggressive $<$

(1+F/K) * min(elapsed time)

- If fetchs early, it can afford to do more fetches
- If making too many misses, then it can't fetch early, and its replacements tends to be optimal

=> For every phase, controlled-aggressive takes at most F time units longer

This bound is tight

Implementation

“PREF”: ~1400 lines

Interface for Prefetching

Programs tell kernel what their future file accesses are

- **File list**

```
file_list_start();
open(f1);
. . .
open(fn);
file_list_end();
```

- **block list**

Experiments

Applications: cscope, dinero, glimpse, postgres, mh

- Eight experiments: cs1, cs2, cs3, din, gli, psel1, psel2, mh
- Two cache sizes: 6.4MB, 12MB
- Platform: DEC 5000/240

Is Optimal Caching Important?

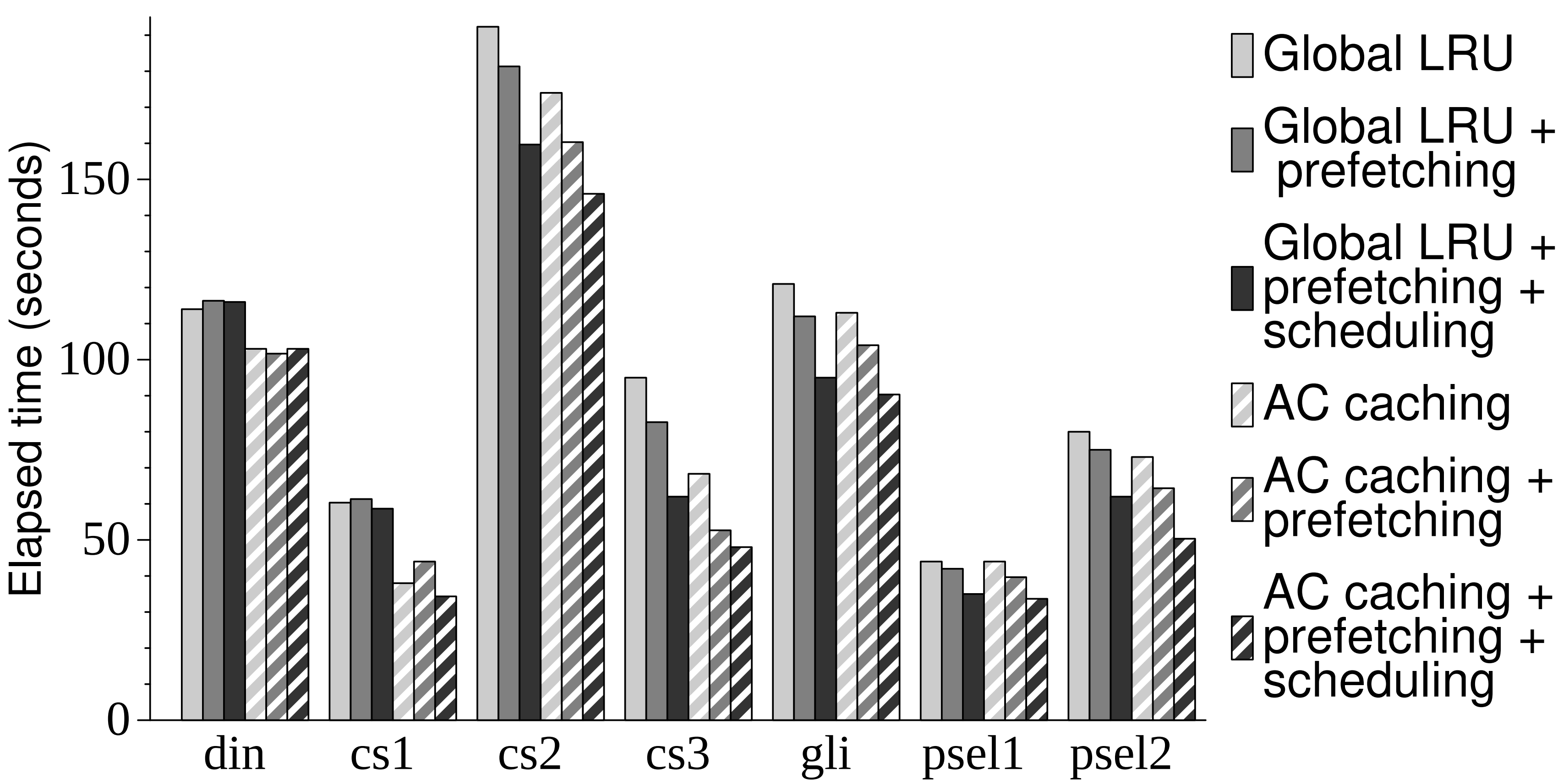
Add In Disk Scheduling

Disk scheduling: “If there are more than one requests to disk, properly order their services so that the total time is minized.”

Prefetching provides new oppurtunities for disk scheduling

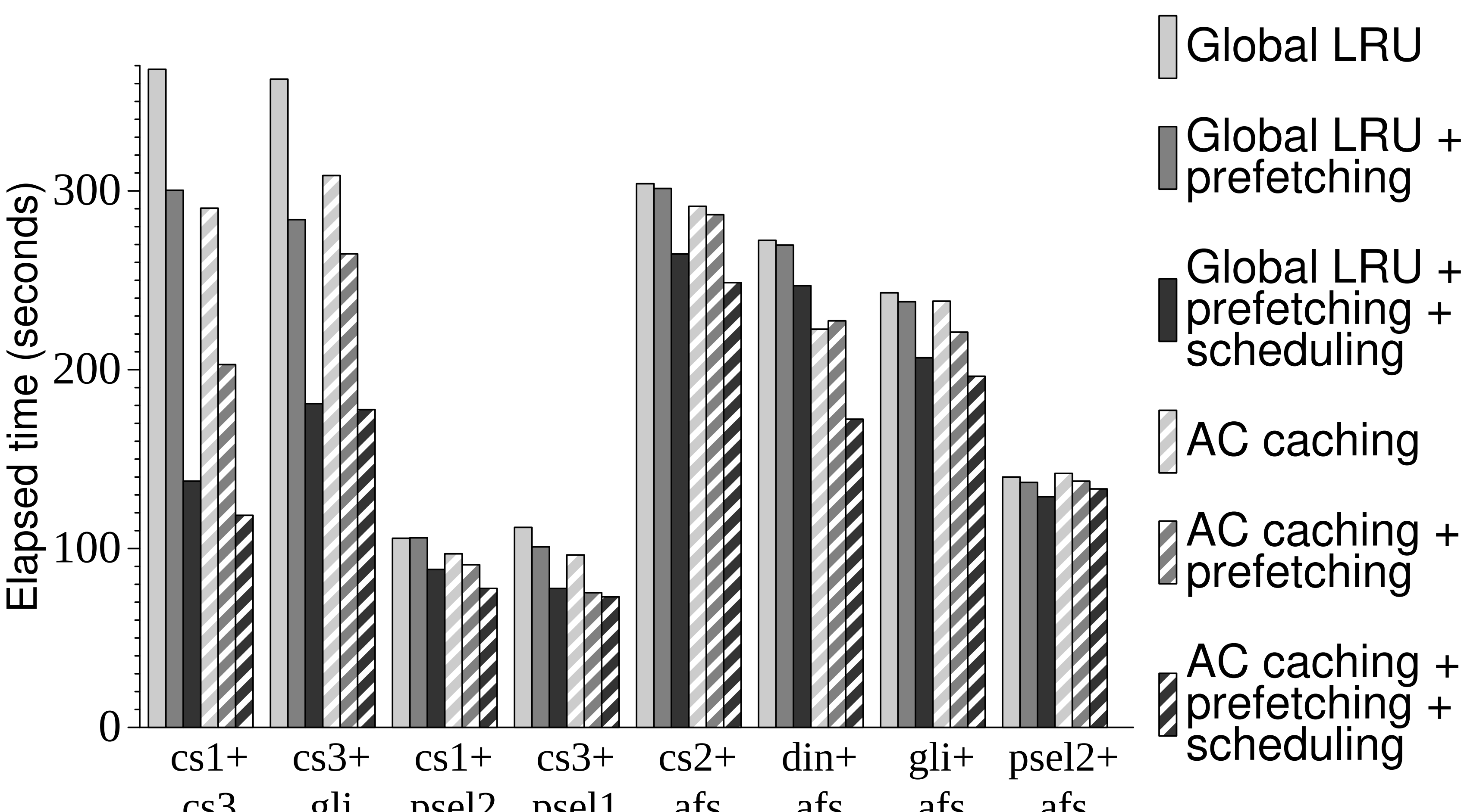
- Issue a batch of prefetch requests at a time
- Sort by ascending disk addresses

Single Application Performance (6.4MB Cache)



Multiple Application Performance

Multiple Process Performance (6.4MB Cache)



Related Work

- Improving file system performance: LFS, clustering, RAID, etc.
- Application-controlled buffer replacement and prefetching in database systems
- Application-controlled VM Caching