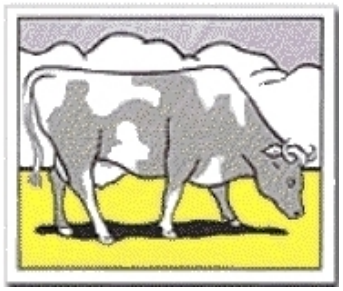
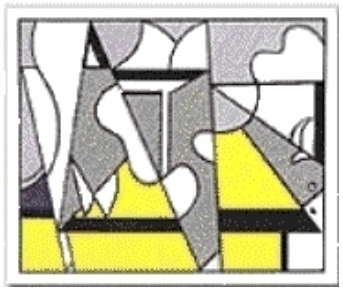
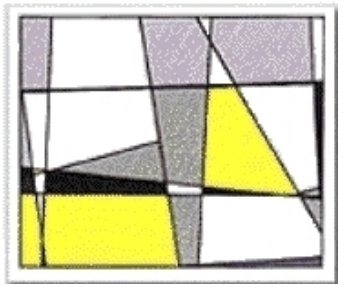




Silent Stores for Free

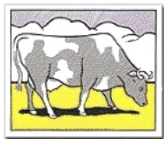
(or, Silent Stores Darn Cheap)

Kevin M. Lepak

Mikko H. Lipasti

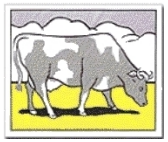
University of Wisconsin—Madison

http://www.ece.wisc.edu/~pharm

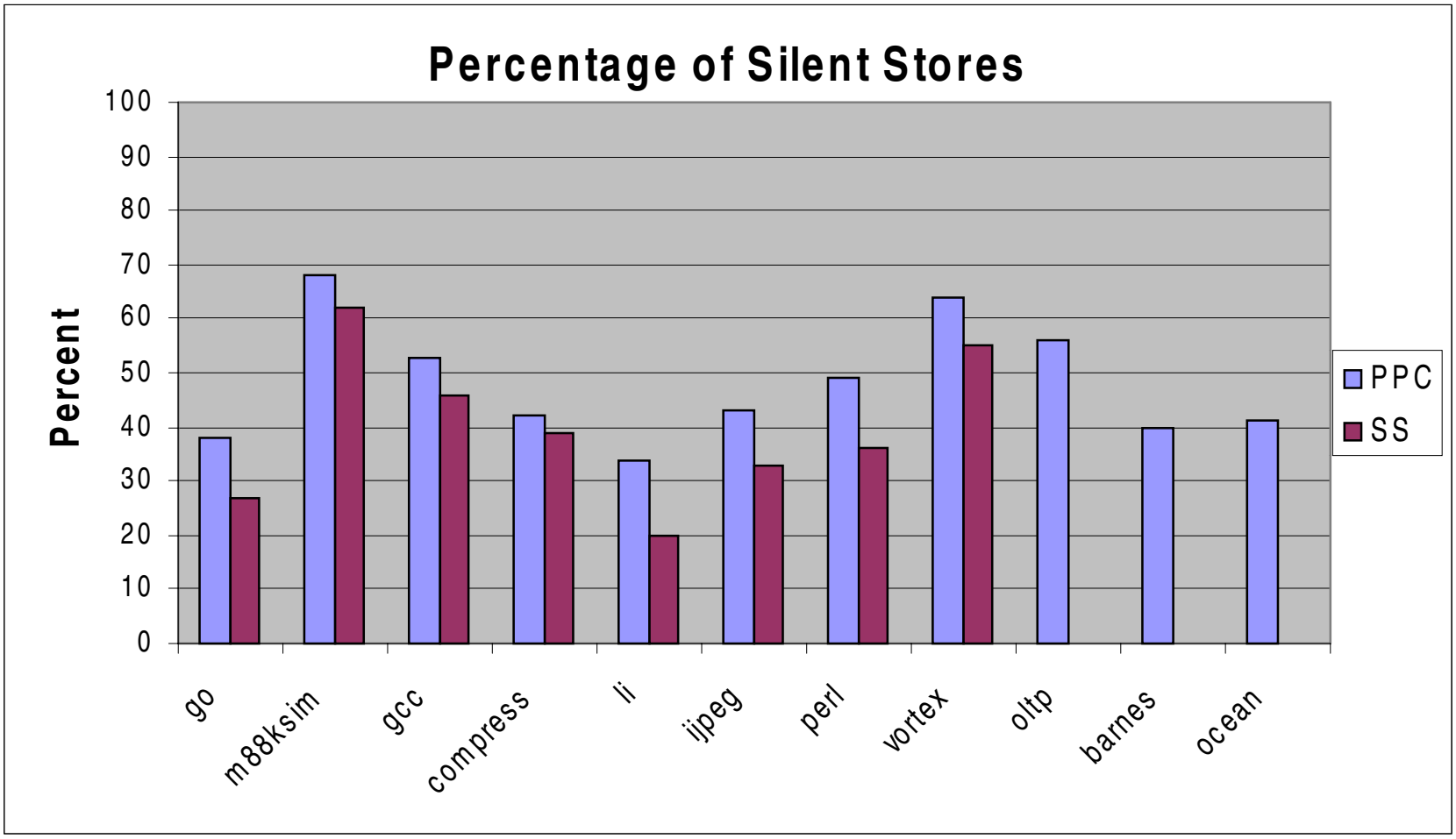


Introduction

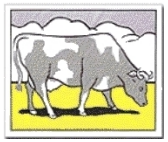
- Recent work shows that many memory writes do not update the system state
  - "Silent Stores" are memory writes which are writing the same value that already exists at that memory location
  - Intuitively, we might be able to exploit this observation for performance benefit



Silent Stores—Is this for Real?

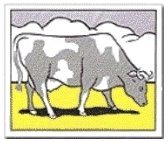


Percentage of silent stores is non-trivial in all cases, 20%-68%



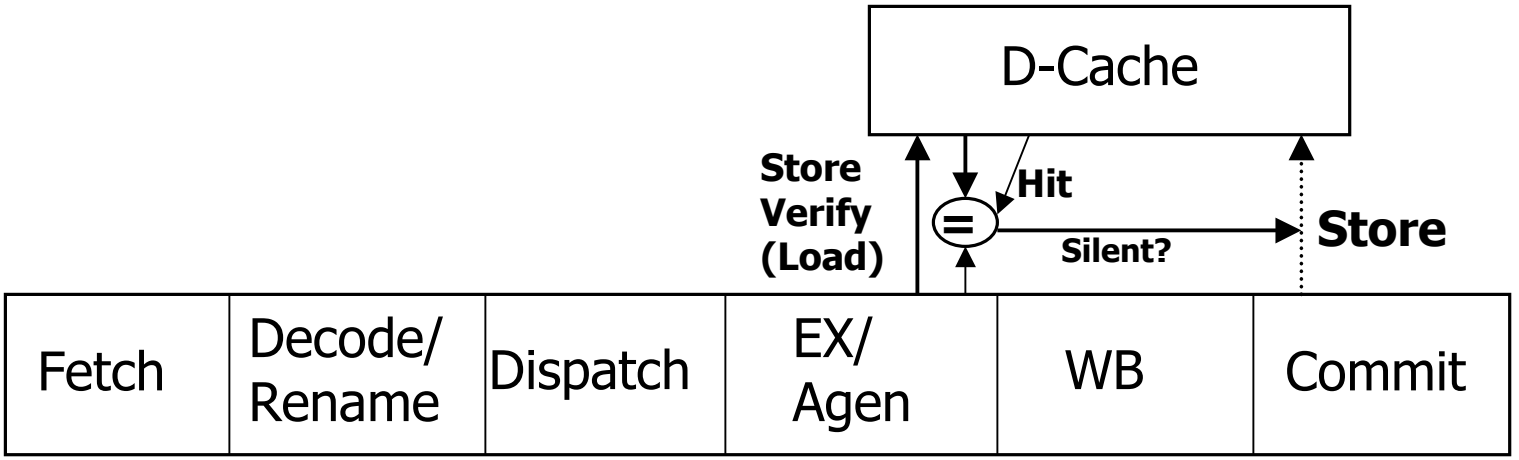
Motivation

- Silent Stores are real and non-trivial
  - 20-60% of dynamic stores are silent
- Multiprocessor benefits:
  - Reduced address and data bus traffic
- Uniprocessor benefits:
  - Reduced writebacks, pressure on write buffers
  - Write port utilization, etc.

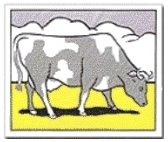


Standard Store Verifies

- Issue a store verify (SV) for every store



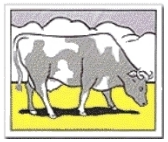
- "Standard" Store Verifies are expensive
  - Load, compare, (store) overhead for every store
    - Increase cache port utilization
    - Can block loads that may be on critical path



Is There a Better Way?

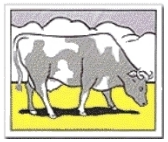
- Predict which stores are likely to be silent and only store verify those
  - Subject of ongoing research
- Find lower cost mechanisms for verifying stores
  - Exploit OoO core μ-arch features
  - Exploit core reliability features for deep-submicron technology trends

Silent Stores for Free:
Reducing the cost of store verification



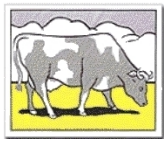
Outline

- OoO core enabled Free Silent Store Squashing (FSSS) Mechanisms
  - Read port stealing
  - Temporal & spatial locality in the load/store queue (LSQ)
- FSSS in ECC cache architectures
  - Data cache protection methods
  - ECC-L1-D$ FSSS
- Trading FSSS for physical bandwidth
- Conclusions



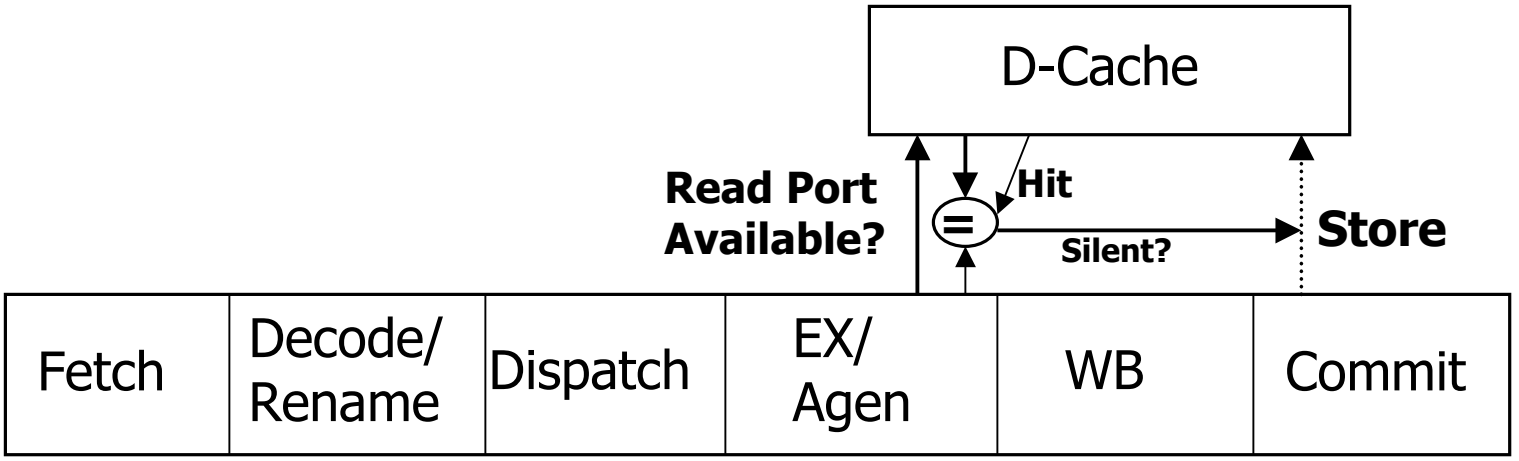
Performance--Machine Model

- SimpleScalar PISA w/realistic memory system
- 8 issue; 64 entry RUU; 64K entry Gshare
- 64KB each L1 I/D cache; 512KB unified L2
- 32 entry load/store queue
- Two fully-pipelined memory access ports
- 32B L1-L2 interface, single cycle occupancy
- Write-through-allocate L1, write-back L2
  - 2 Write buffers, 32B write-combining

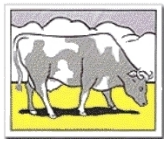


Read Port Stealing

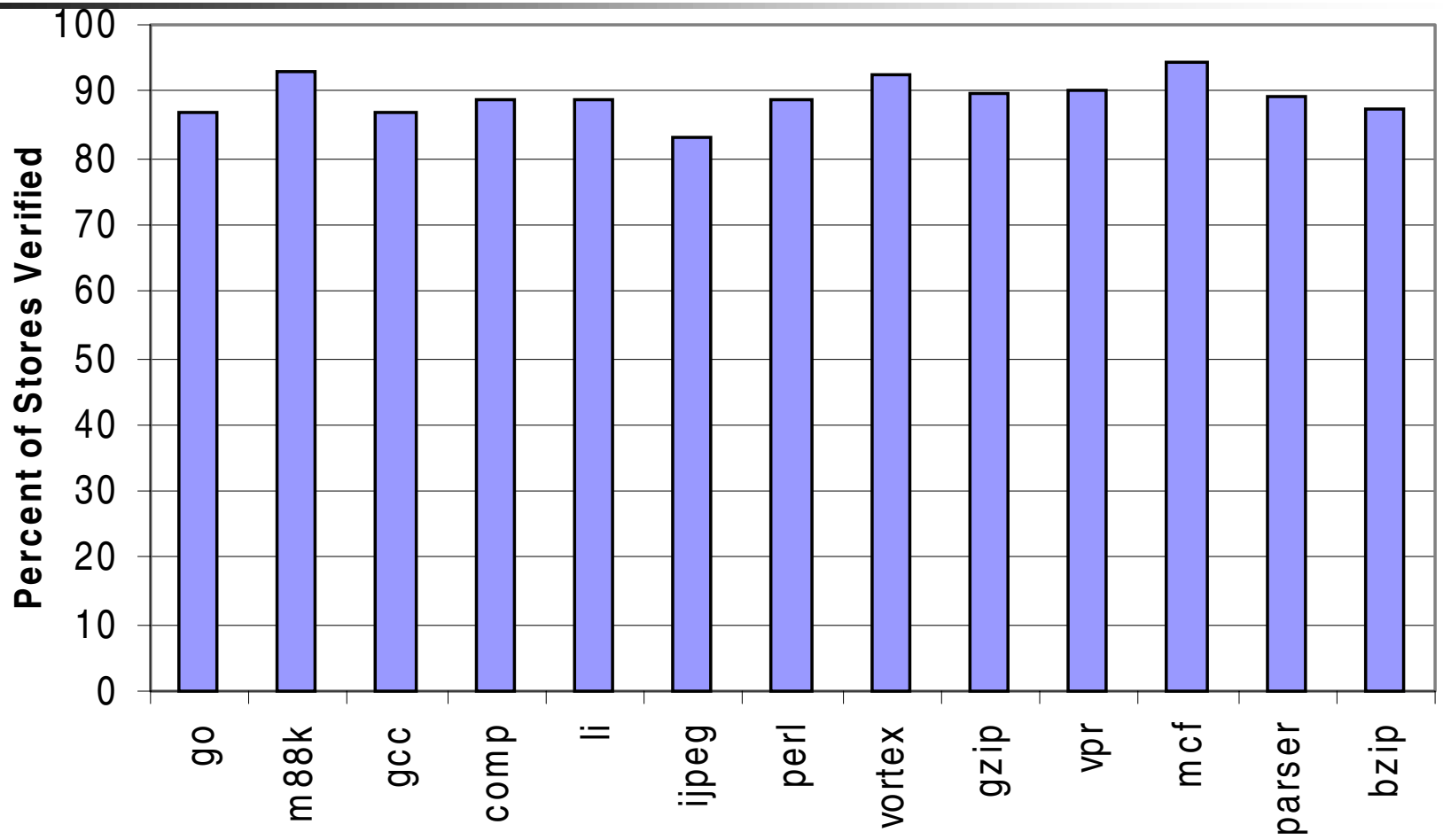
- Only issue a store verify if a cache port is available (schedule ready loads/stores first)
- If a store reaches the head of the ROB before it can be verified, assume it is non-silent



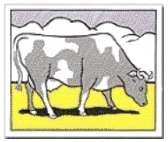
- Similar to standard SV, but does not delay ready loads/stores and does not capture all silent stores



Read Port Stealing--Opportunities

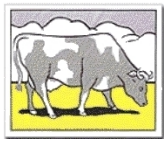


Captures minimum of 84% of store verify opportunities



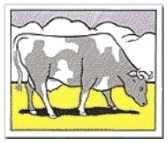
LSQs

- OoO cores implement LSQs to track in-memory dependences for improved performance
  - store forwarding
  - consistency model violations
- LSQs provide temporal and spatial context for a memory operation
  - Surrounds an operation with other references local to it in dynamic program order



LSQ Temporal Locality

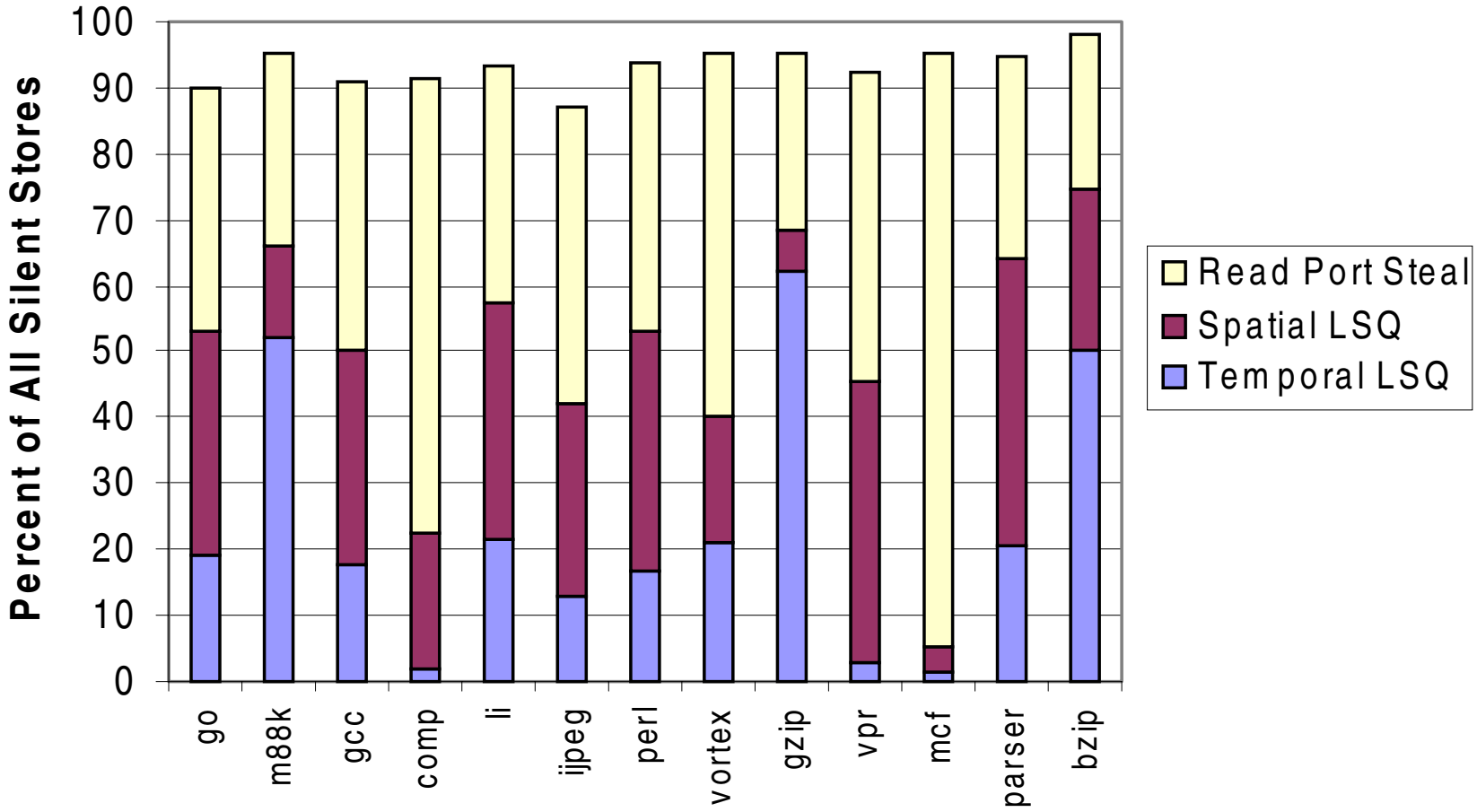
- We can exploit temporal locality (same address aliases) in the LSQ to verify stores
  - WAW: Can forward store to load, why not store to store?
  - WAR: Load allocates data from the cache, use it to squash a subsequent store
  - RAW: In many μ-arch, cache port scheduled before aliasing to an entry in the LSQ is known, use the port to store verify



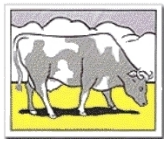
LSQ Spatial Locality

- Obtaining a wide datapath to L1-D$ is possible due to on-chip caches
- Assume a memory reference can provide an entire cacheline of data
  - Exploit spatial locality to issued memory references
    - WAR: Load allocates an entire line
    - WAW: Use read port stealing to allocate
    - RAW: Load allocates an entire line

LSQ Squashing--Silent Stores Captured

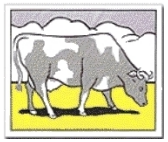


Over 90% of silent stores captured;
Greater than 40% in most cases using locality in LSQ



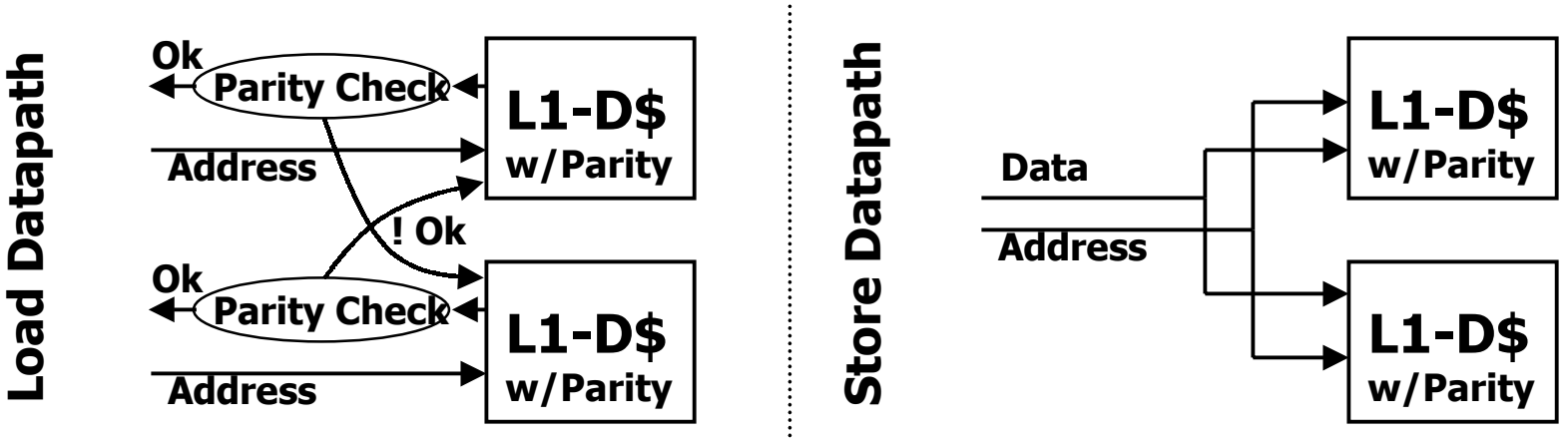
Memory Storage Soft Errors

- Detecting and correcting soft errors is becoming more important
  - Deep-submicron manufacturing
  - Uptime & system reliability concerns
- Many methods exist for ECC:
  - Rely on redundancy for detection/correction
    - Coding: Keep extra bits that allow both **detection** & **correction**
    - Explicit copies: Keep multiple copies with extra bits for **detection**, correct by loading the copy

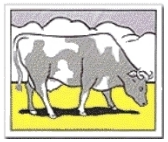


Redundant Data ECC for L1-D$

- Duplication of parity protected L1-D$

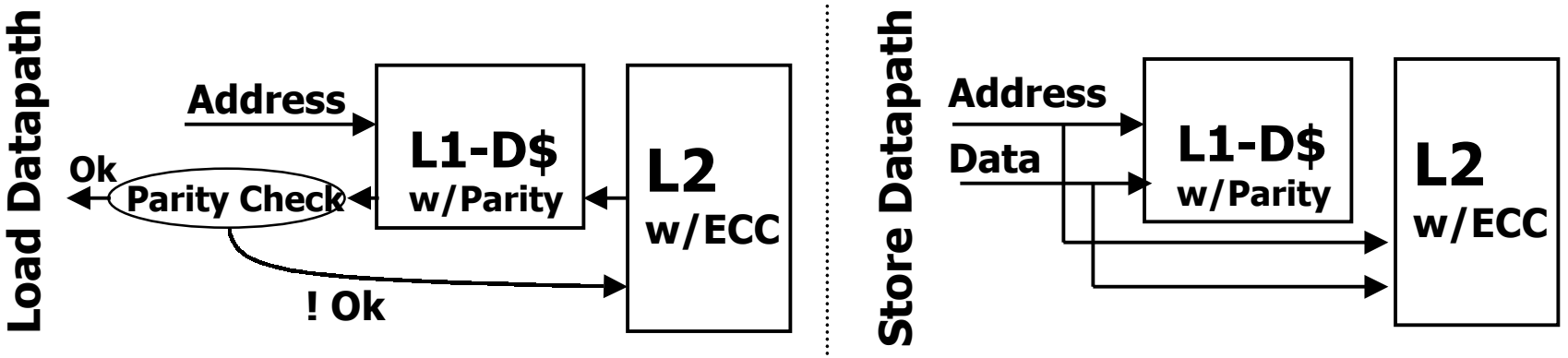


- High overhead--100% over L1-D$ with parity
- 2x read bandwidth vs. write bandwidth
- Leads to configurations with higher load throughput

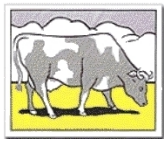


Redundant Data ECC for L1-D$

- Write-through parity protected L1-D$ with inclusive (ECC code protected) L2

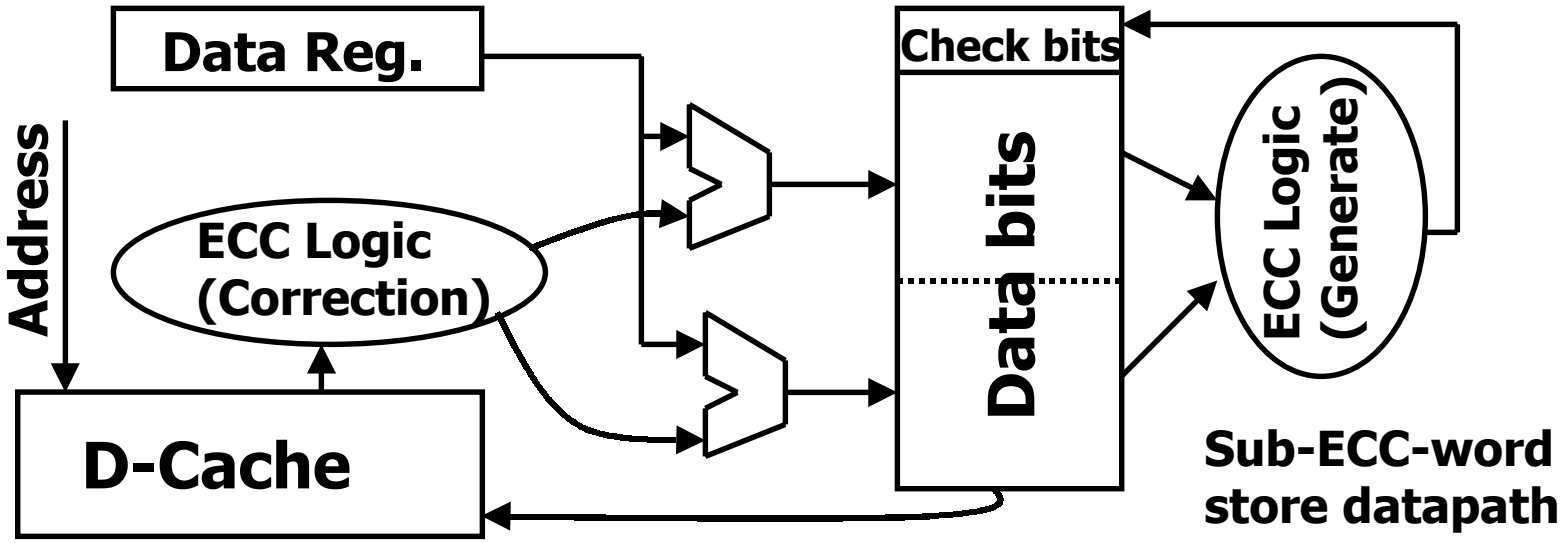


- Write-through creates high demand on the L1-L2 interface
- Can use previous FSSS techniques to reduce stores (and hence write-throughs)

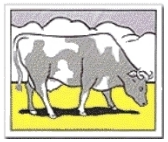


Coding ECC for L1-D$

- Protect the L1-D$ with ECC directly
- ECC-data words relatively large to reduce overhead (ex: 64-bit in 21264, RS64-III)

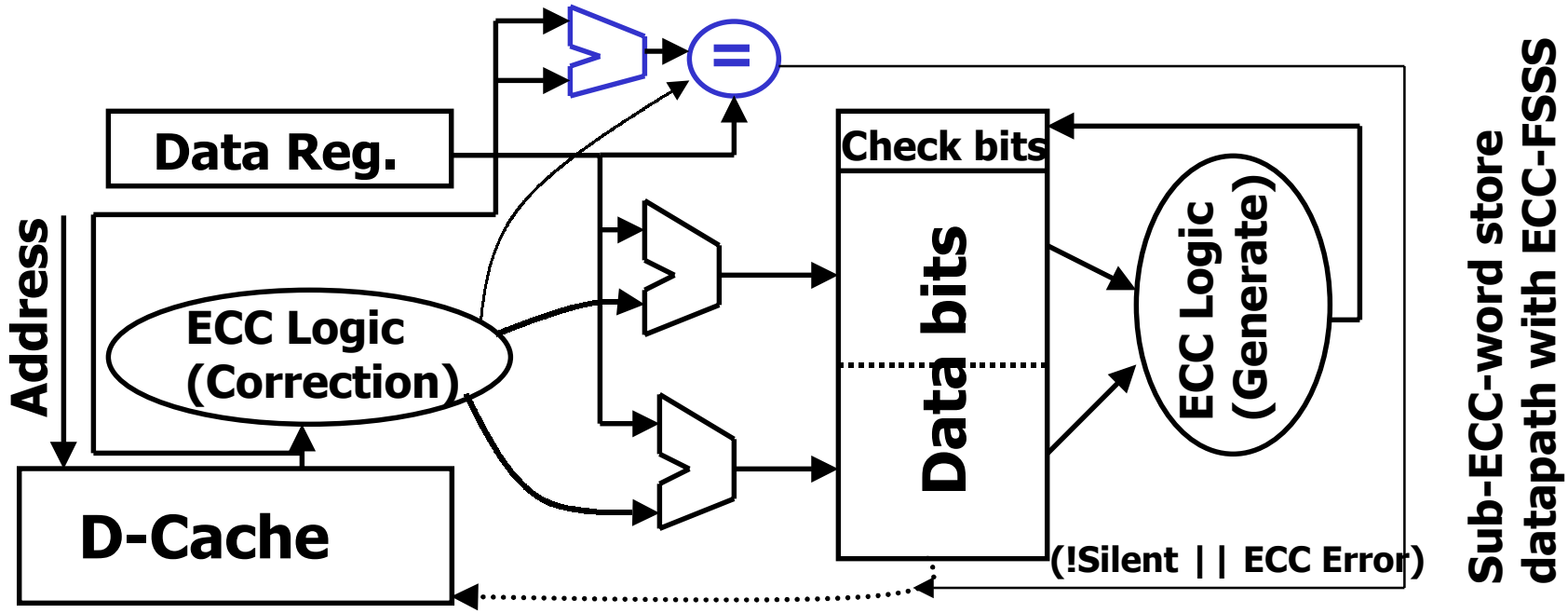


Sub-ECC-word stores consist of four operations: Read original ECC-word, Merge, ECC-gen, Write

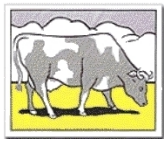


ECC L1 Free Silent Store Squash

- If sub-ECC-word stores are read-modify-writes, why not squash?

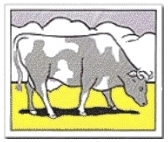


Store verify in parallel with correction & check bit generation gives ECC-FSSS



Effectiveness of ECC-L1 FSSS

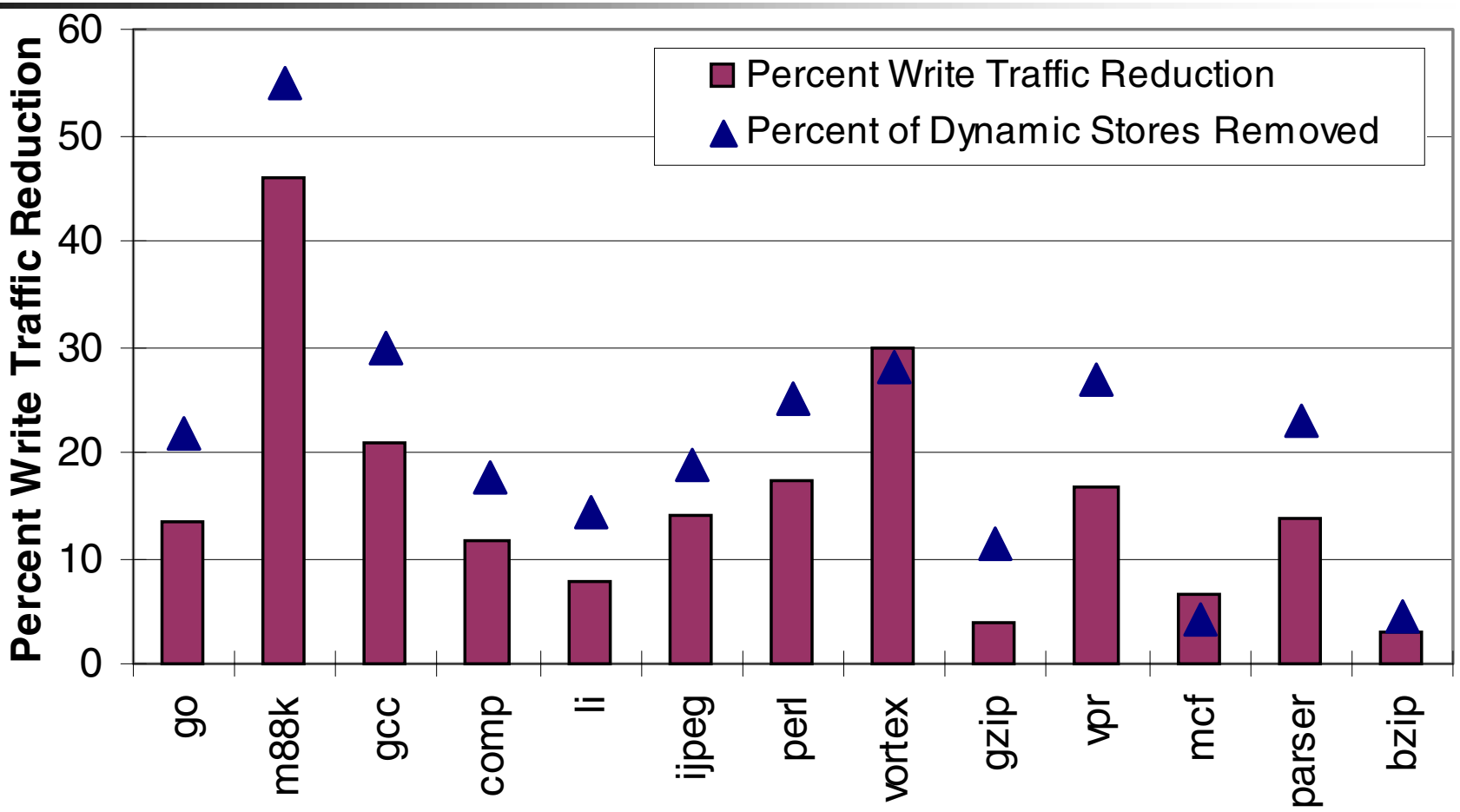
- Can detect 100% of sub-ECC-word L1-D$ hits
  - store-byte (8b), store-half (16b), store-word (32b) in 64b-ECC-data-word μ-arches
- Can also capture **many** more which might not be so obvious
  - IBM RS64-III (Pulsar) has maximal 32b integer stores in 32b mode (common for user programs)
    - All of these can be captured with ECC-FSSS



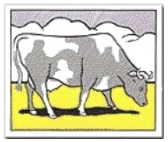
Increasing Write-Through Bandwidth via FSSS

- We expect squashing silent stores to reduce pressure on the L1-L2 interface
- Can we implement a narrower/slower L1-L2 **physical** interface and exploit FSSS for greater **effective** interface bandwidth?
  - Potentially reduce power consumption
  - Ease circuit & physical design

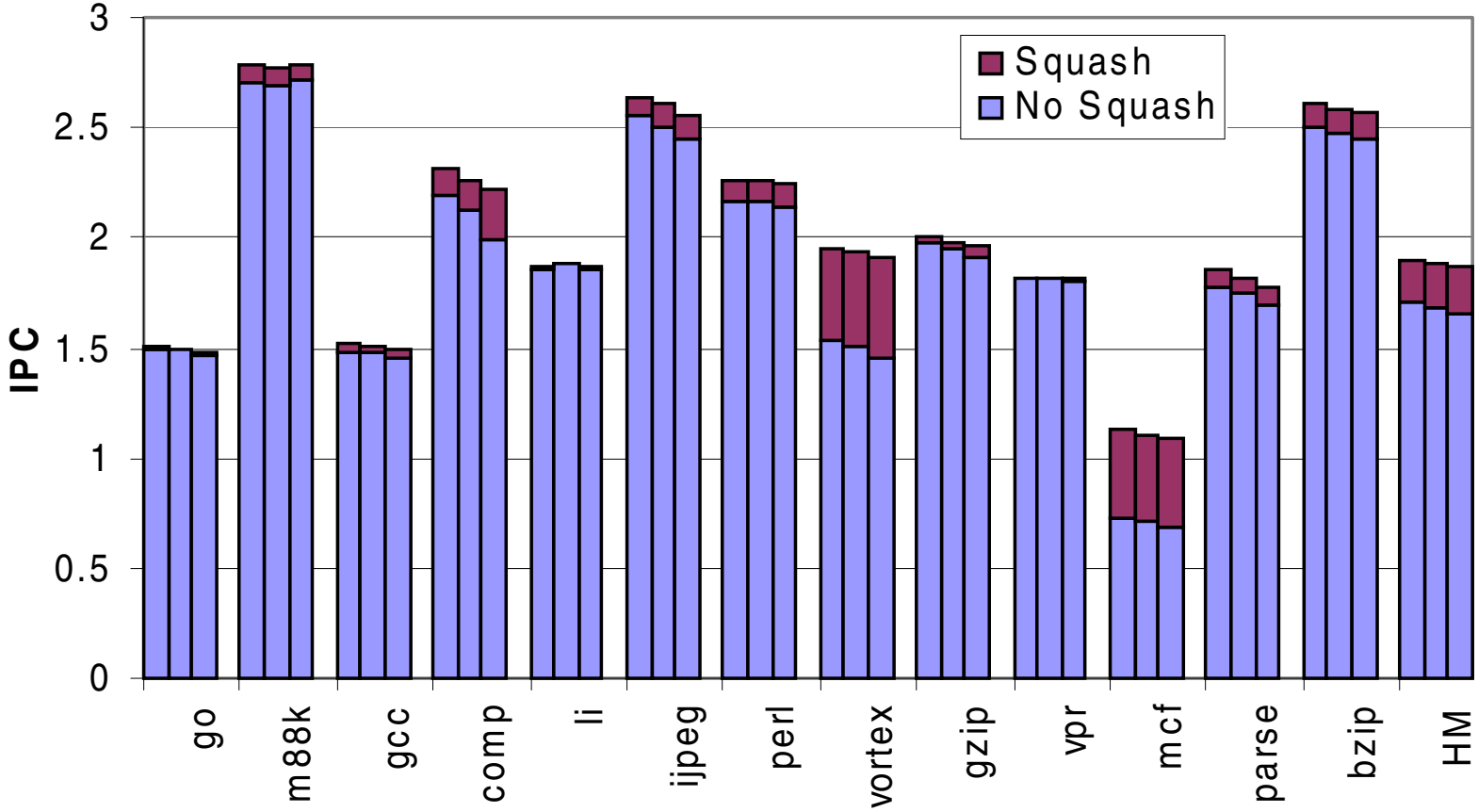
Increasing Write-Through BW--Write-Through Reduction



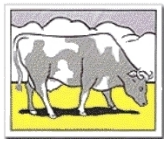
15% average write-through traffic reduction



Increasing Write-Through BW--IPC

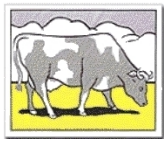


75% lower physical BW+FSSS yields 9% IPC improvement over fast physical interface without FSSS



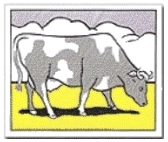
Conclusions

- Standard store verifies are expensive
- Three methods of squashing silent stores for reduced cost
  - Using read port stealing
  - Exploiting temporal and spatial locality in the LSQ
  - Using ECC logic in the L1 data cache
- These methods verify a large fraction of silent stores for non-trivial speedups
- Trade implementation of silent store squashing for higher physical BW between L1 & L2

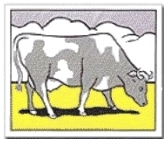


Current and Future Work

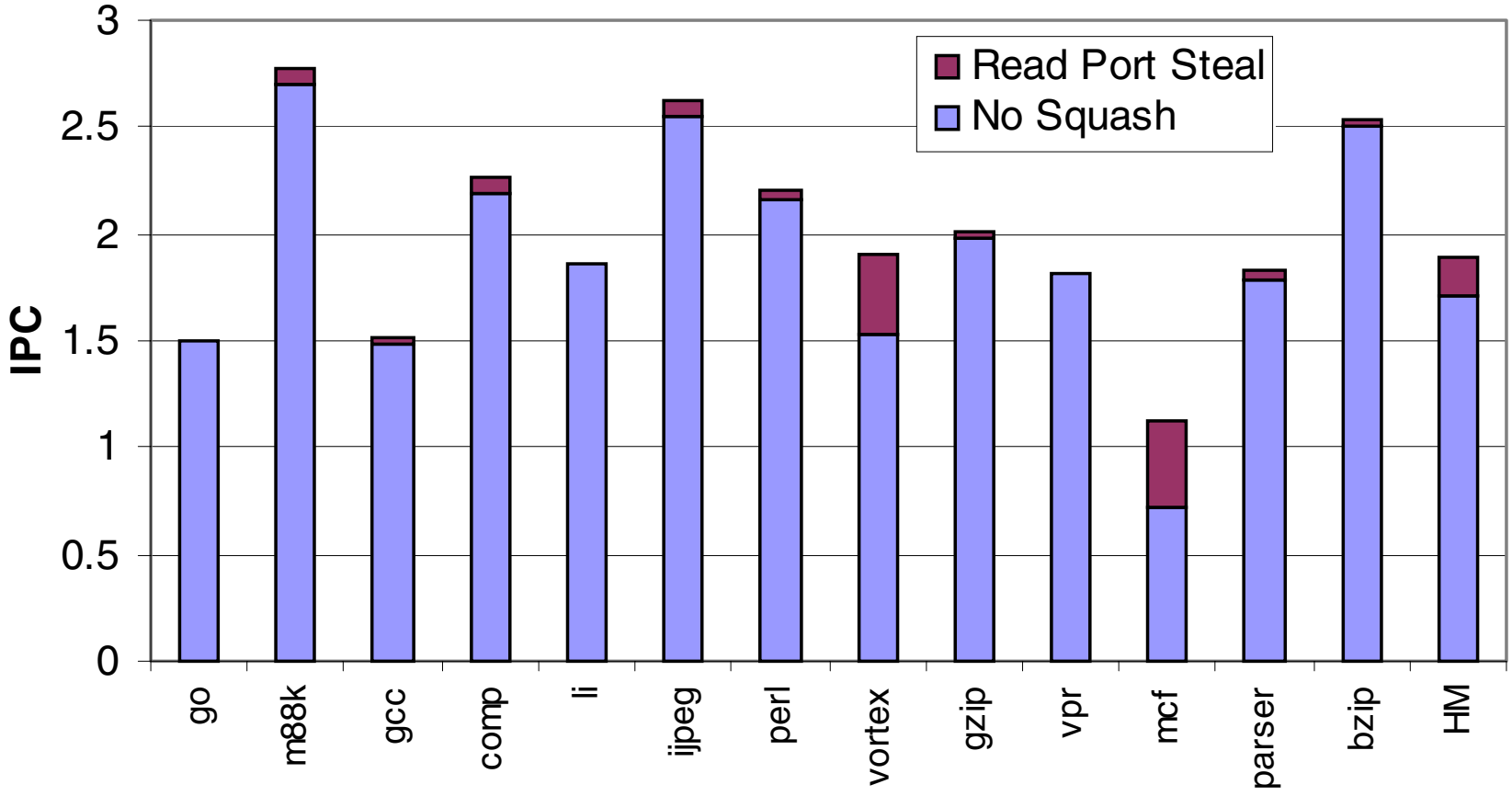
- Silent stores in MPs, as well as program structure and message passing store value locality [Lepak & Lipasti, ISCA-2k]
- Characterizing & Critical silent stores [Bell et. al PACT-2k]
- Silence confidence mechanism(s)
- Exploiting predictable stores in MP systems
- Applying all types of store value locality in different system paradigms
- . . .



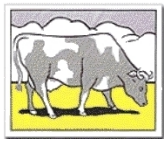
Backup Slides



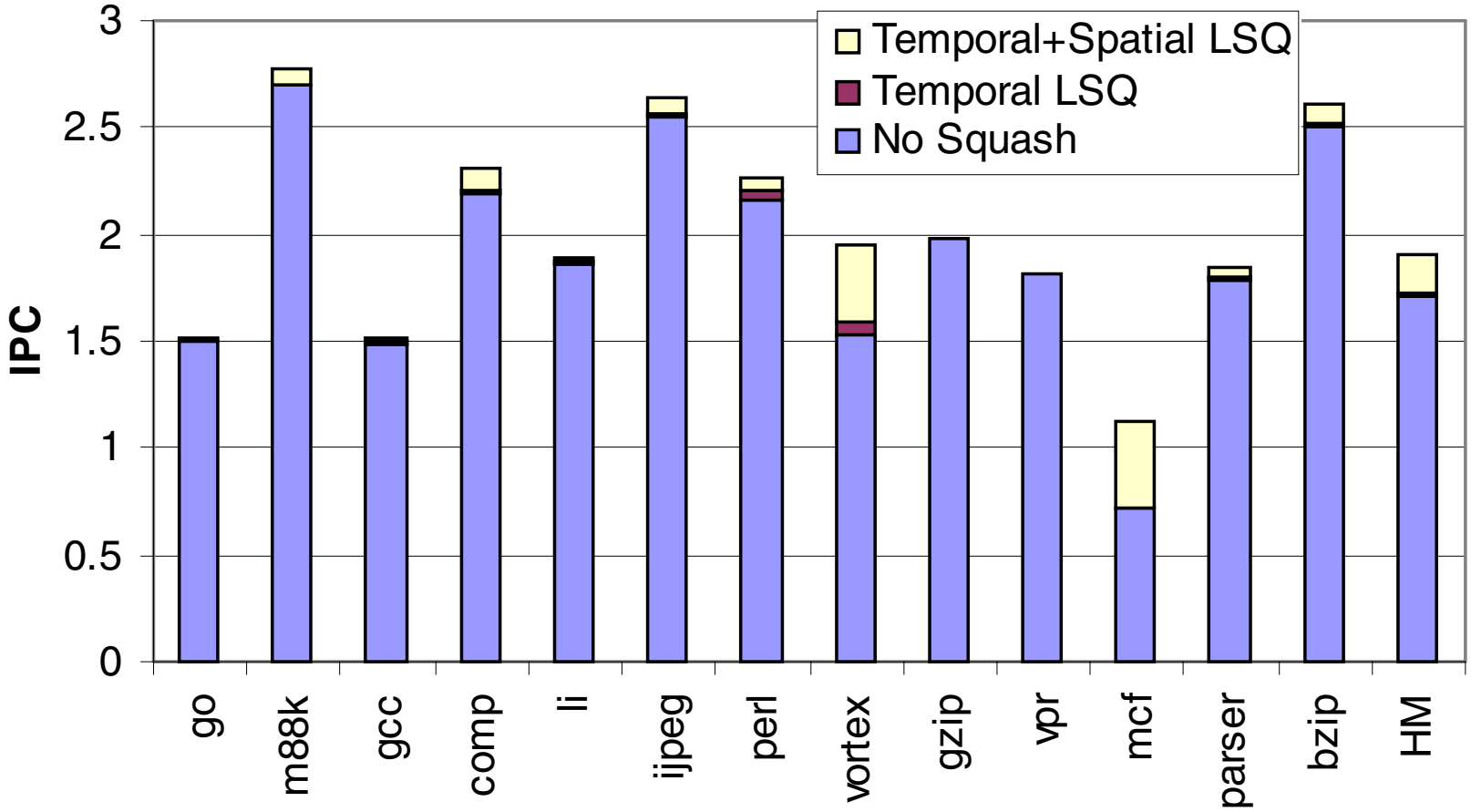
Read Port Stealing--IPC



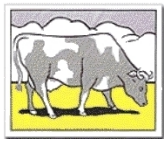
HM improvement of 10%, 0-56% range across benchmarks



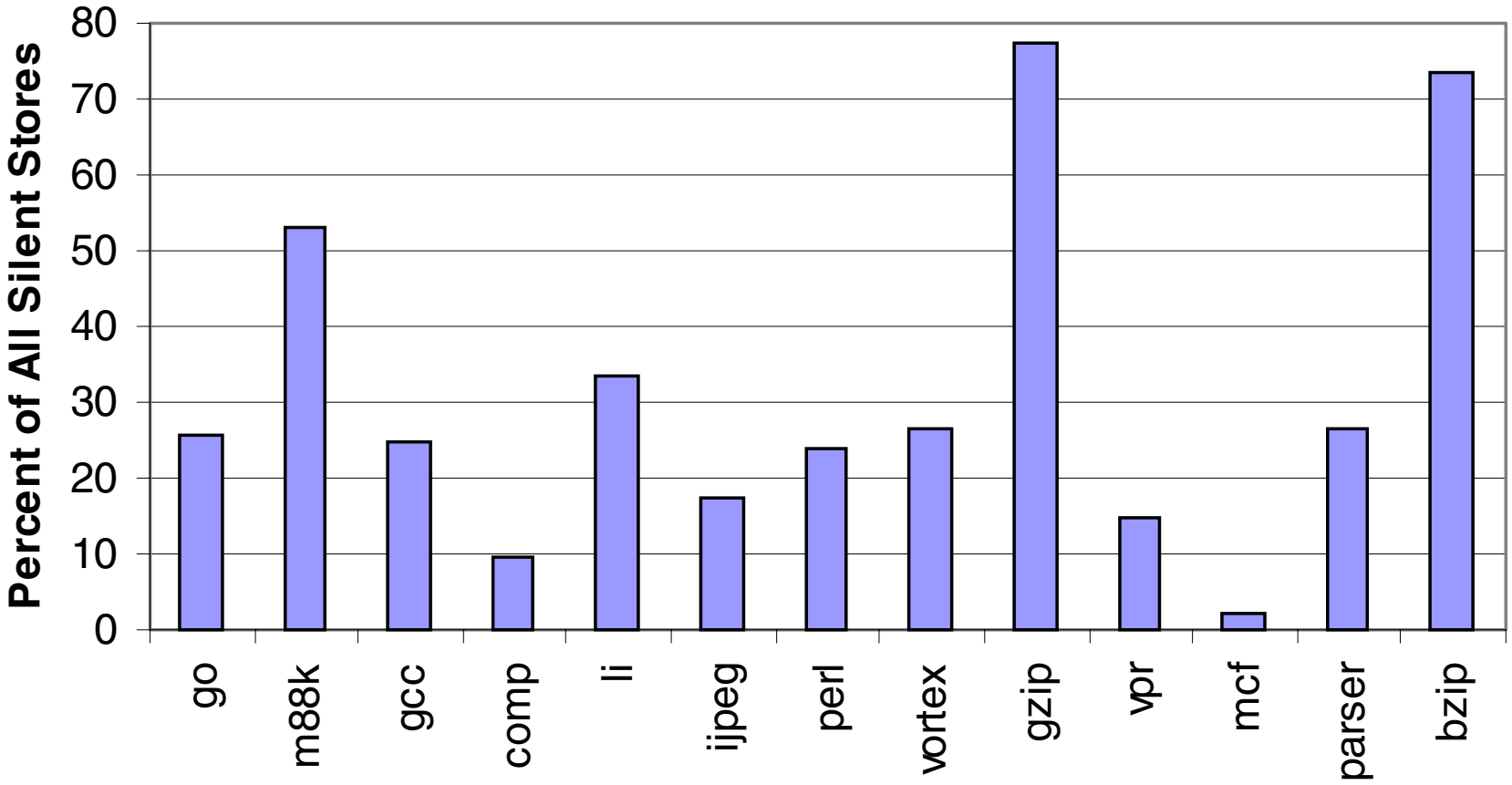
LSQ Squashing--IPC



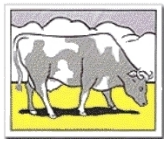
HM improvement of 11%, 0-56% range across benchmarks



LSQ Temporal--Silent Stores Captured

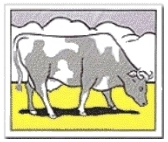


Captures an average of 30% of silent stores across benchmarks

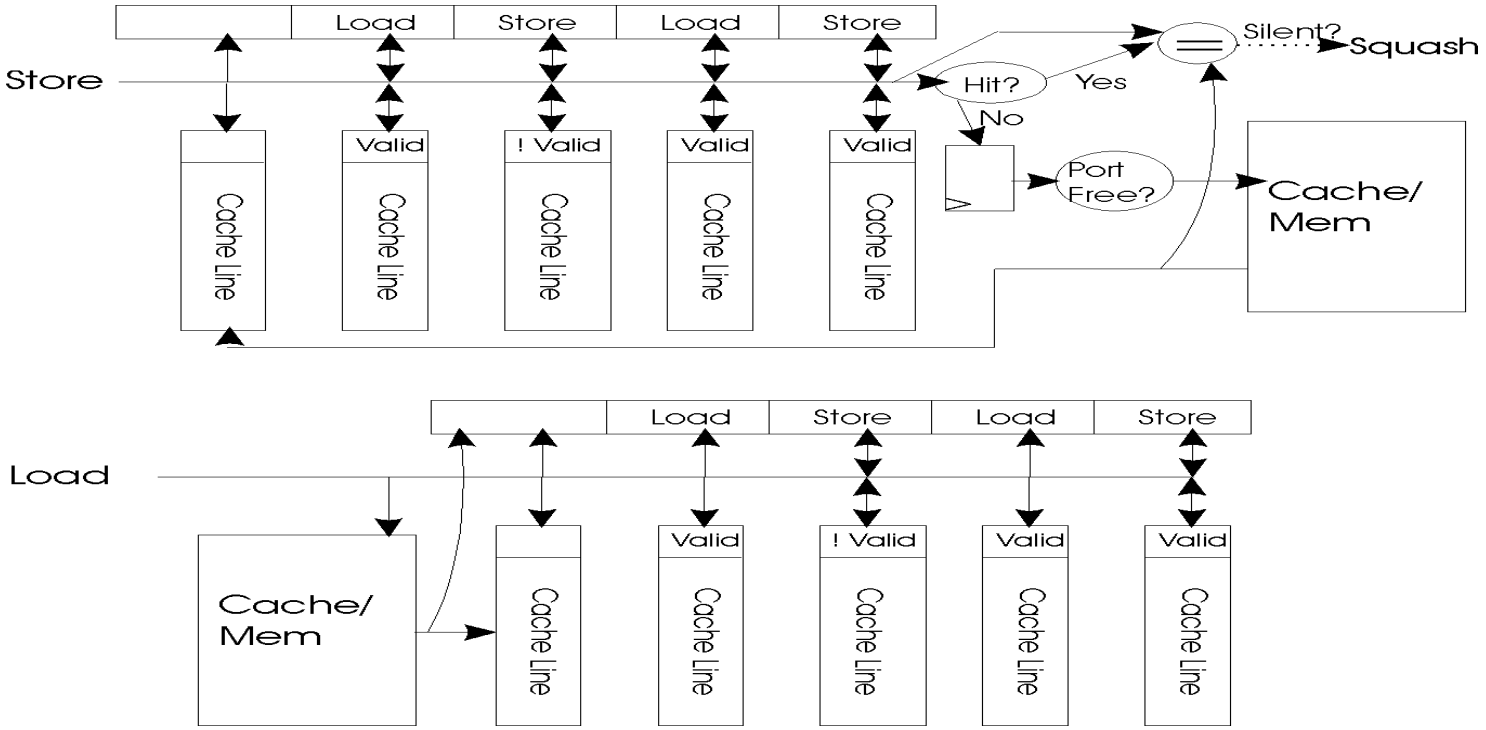


FSSS Method Comparison

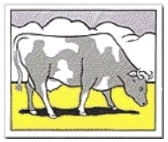
- Read Port Stealing and LSQ squashing provide similar performance results
  - However, LSQ squashing reduces the percent of store verifies issued to the memory system by 50%
- Temporal LSQ squashing is not effective in isolation for this machine
  - May be useful to reduce sharing
- ECC squashing is **truly** free



LSQ Cache Design

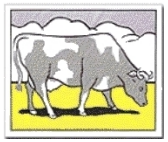


- Assume FIFO LSQ cache operated in lock-step with LSQ
  - Avoids explicit tags, replacement policy considerations
  - MPs: Flush on memory barriers (WC)
  - MPs: Use existing LSQ logic for SC to invalidate (e.g. R10K)

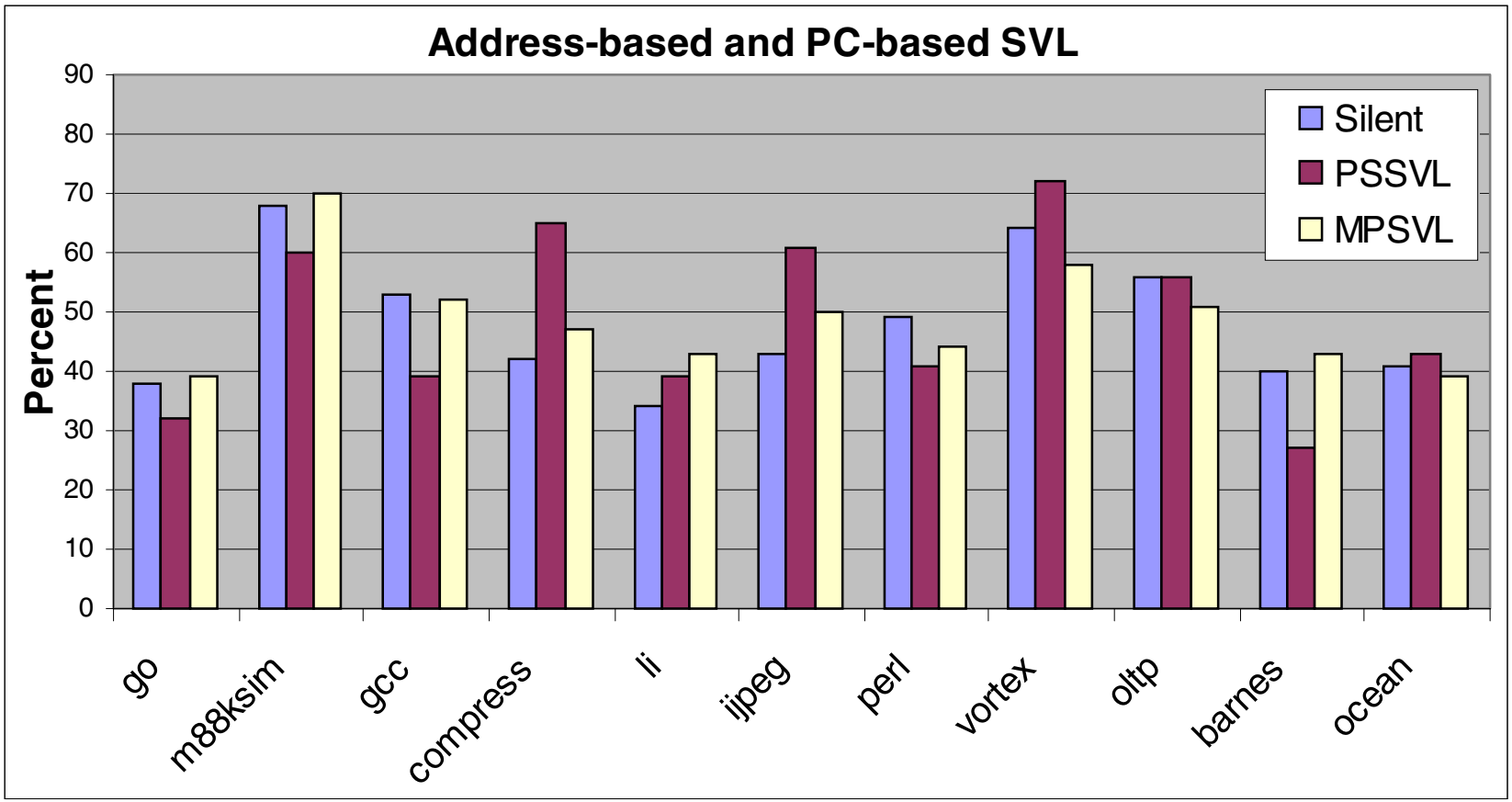


Terminology

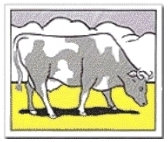
- ***Program Structure Store Value Locality (PSSVL):*** The value locality exhibited by a given static store (can write to many addresses)
- ***Message Passing Store Value Locality (MPSVL):*** The value locality exhibited for a specific memory location (can be written by many PCs)
- ***Stochastically Silent Store:*** A store value which is trivially predictable by any well known method



MPSVL and PSSVL

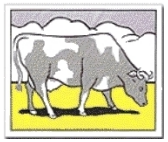


Percentage of stochastically silent (PSSVL, MPSVL) stores is non-trivial
27%-72% for PSSVL, 39%-70% for MPSVL



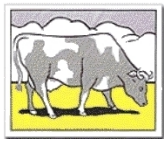
Multiprocessor Sharing

- Measurable reduction in true/false sharing for simple update silent squashing (UFS)
- Substantial reductions by squashing update silent store hits and misses (UFS-P) and stochastically silent stores (SFS)
- Squashing store misses (UFS-P) can be substantially better than simple UFS
  - Motivates silence confidence mechanism for store misses



Multiprocessor Traffic

- Measurable reduction in invalidate traffic for simple update silent store squashing (UFS)—more effective than Exclusive state
  - Substantial reduction for UFS-P and Stochastic False Sharing (SFS)
- Writeback data traffic reduction by squashing update silent store hits and misses (UFS-P)
  - 5%-82% in *oltp*
  - 16%-17% in *ocean*
  - 5%-16% in *barnes*



Multiprocessor Sharing