

ADVANCED PARALLEL COMPUTING 2017

LECTURE 10 - ADVANCED TOPICS I: SVM, MTA, NUCA

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

SHARED VIRTUAL MEMORY (SVM)

(PAGE-BASED) SHARED VIRTUAL MEMORY (SVM)

Main memory is a fully associative cache of virtual address space

Use page faults to handle coherence

Span up coherence domains across physically distributed nodes

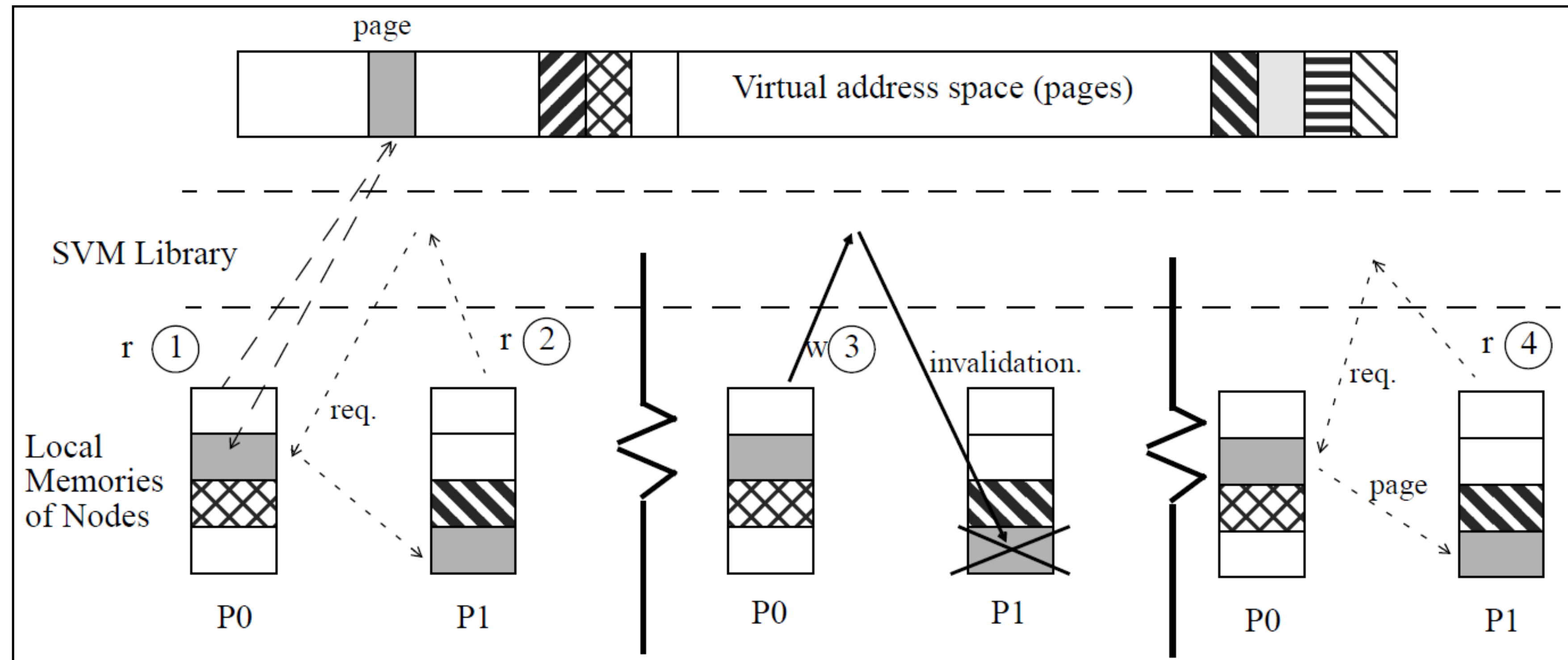
Multi-level cache coherence: local caches do not have to be kept coherent across nodes

No special hardware support required (uniprocessor support sufficient)

Basic protocol functions as before:

1. Finding the source of state information
2. Finding the appropriate copy/copies
3. Communicating with the copies

SVM EXAMPLE



Culler et al, Parallel Computer Architecture, MK 1999

Independent memory management units (MMUs)

Different physical addresses for one virtual address

Directory or home node can hold information (state/copies)

MAIN PROBLEMS

High overheads of protocol invocation and processing

- Handled in SW on a general-purpose processor

- Page faults take time to cause an interrupt/trap to switch to OS

- Communication done by message exchange

- Incoming requests interrupt the processor and pollute caches, slowing down the currently running thread

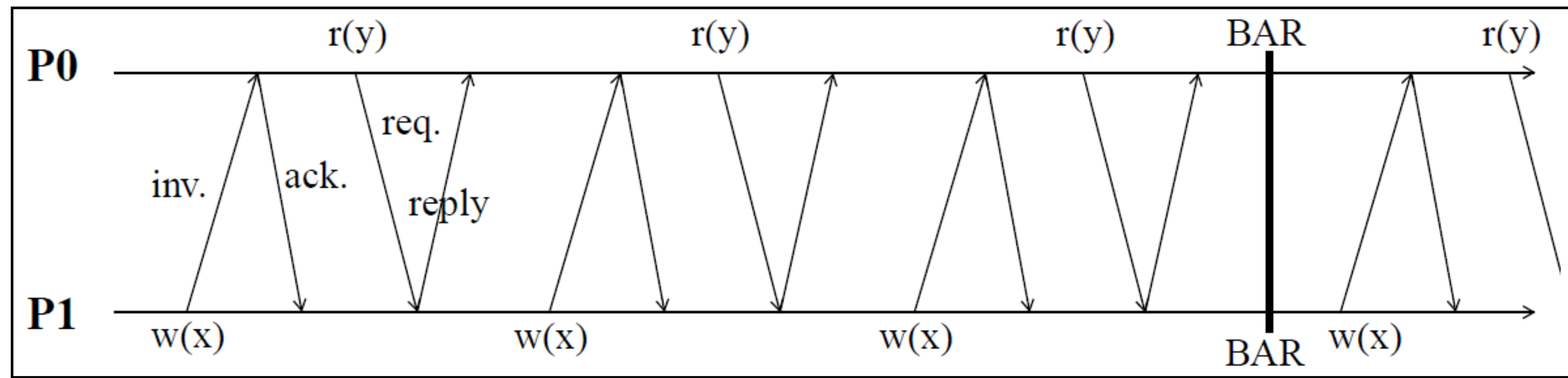
Large granularity of coherence and communication

- For low spatial locality, causes fragmentation in communication and thus useless data transfers

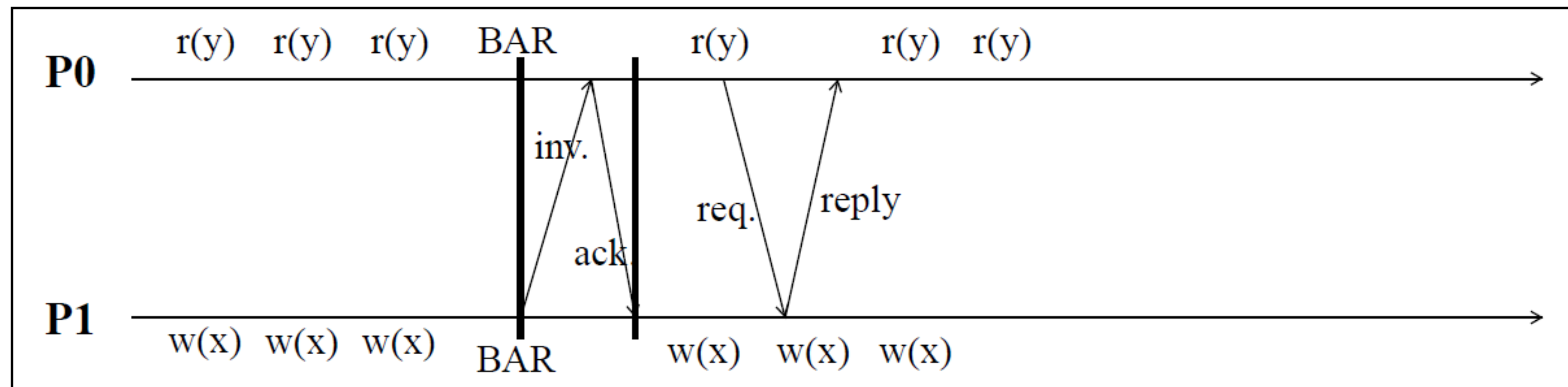
- False sharing leads to frequent page migrations

SVM CONSISTENCY

Sequential consistency



Under relaxed consistency



SVM RELAXED CONSISTENCY - DIFFERENT GOAL

HW-coherent machines: goal is to reduce latency

Avoid stalling the processor to wait for acknowledgements (completion), but invalidations are sent out as early as possible

Although release consistency does not guarantee the effects to be visible before synchronization, in fact they usually will be

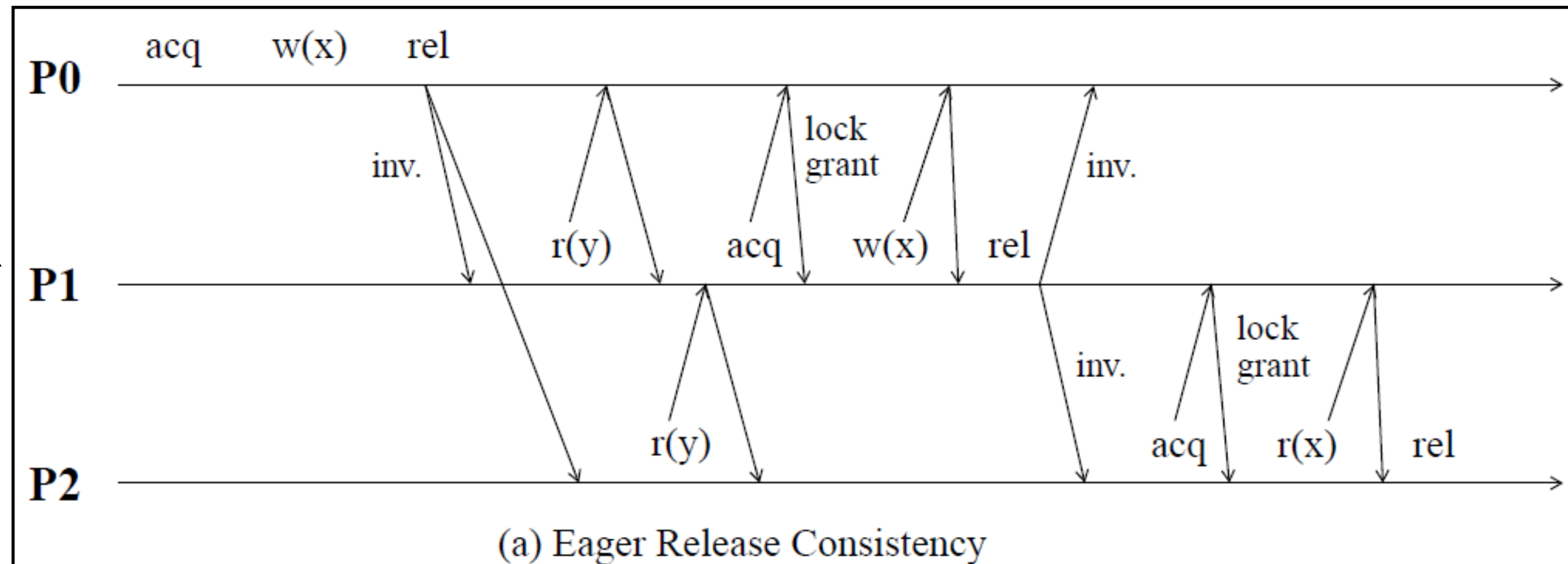
SW-coherent machines: goal is to reduce the amount of false sharing

Invalidations are not propagated until synchronization

Can also be used for HW as well (see delayed consistency, Dubois et al., 1991)

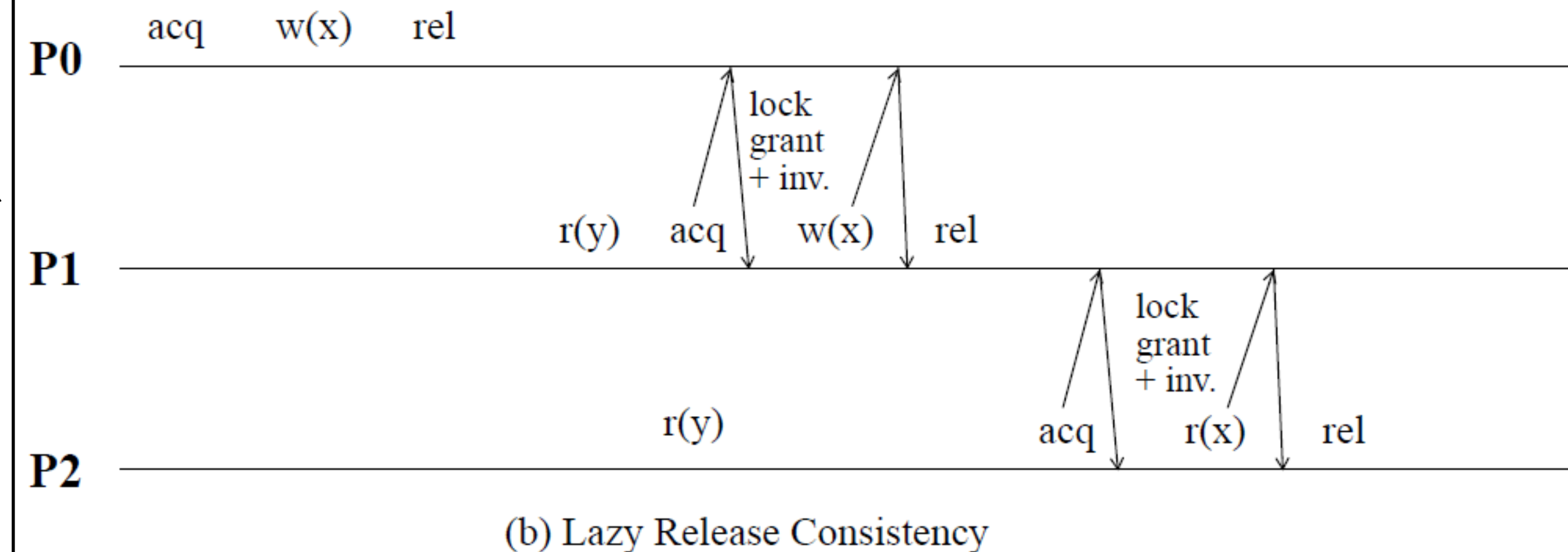
SVM: EAGER VS. LAZY RELEASE CONSISTENCY

Culler et al, Parallel Computer Architecture, MK 1999



x and y are on the same page

ERC is pessimistic, sends out more invalidations than necessary (in this example)



HW LRC would send out invalidations after the release

SVM CONSISTENCY IS DIFFERENT TO HW

SW protocols do not satisfy all consistency requirements

Writes are not guaranteed to be propagated unless synchronization takes place

Different processors may see writes through different synchronization chains (in principle also applies to HW)

SW-LRC main question: how to determine which releases happened in which order before an acquire (synchronization chains)

ERC != LRC!

P0
lock L1;
ptr = non_null_ptr_val;
unlock L1;

P1

while (ptr == null) {};
lock L1;
a = ptr;
unlock L1;

ERC: works!
LRC: starvation!

SVM: MULTIPLE WRITER PROBLEM

Delaying write notices

- Mitigate the effects of false sharing

- Multiple writer problem persists (multiple processors writing to one block)

Key issue: more than one writer allowed

- No ownership required before writing to it

1. Page diff: make internal copy on page write fault, later determine differences, then apply differences

- For LRC: garbage collection required

- Storage requirements

- Use a home node, where diffs are sent to

2. Accelerate home-based, multiple writer protocols by avoiding diffs:

- Use network interfaces that establish mappings between pages on different nodes

- Propagate writes immediately in hardware -> write visibility (+)

- HW support required (-), increased traffic (-)

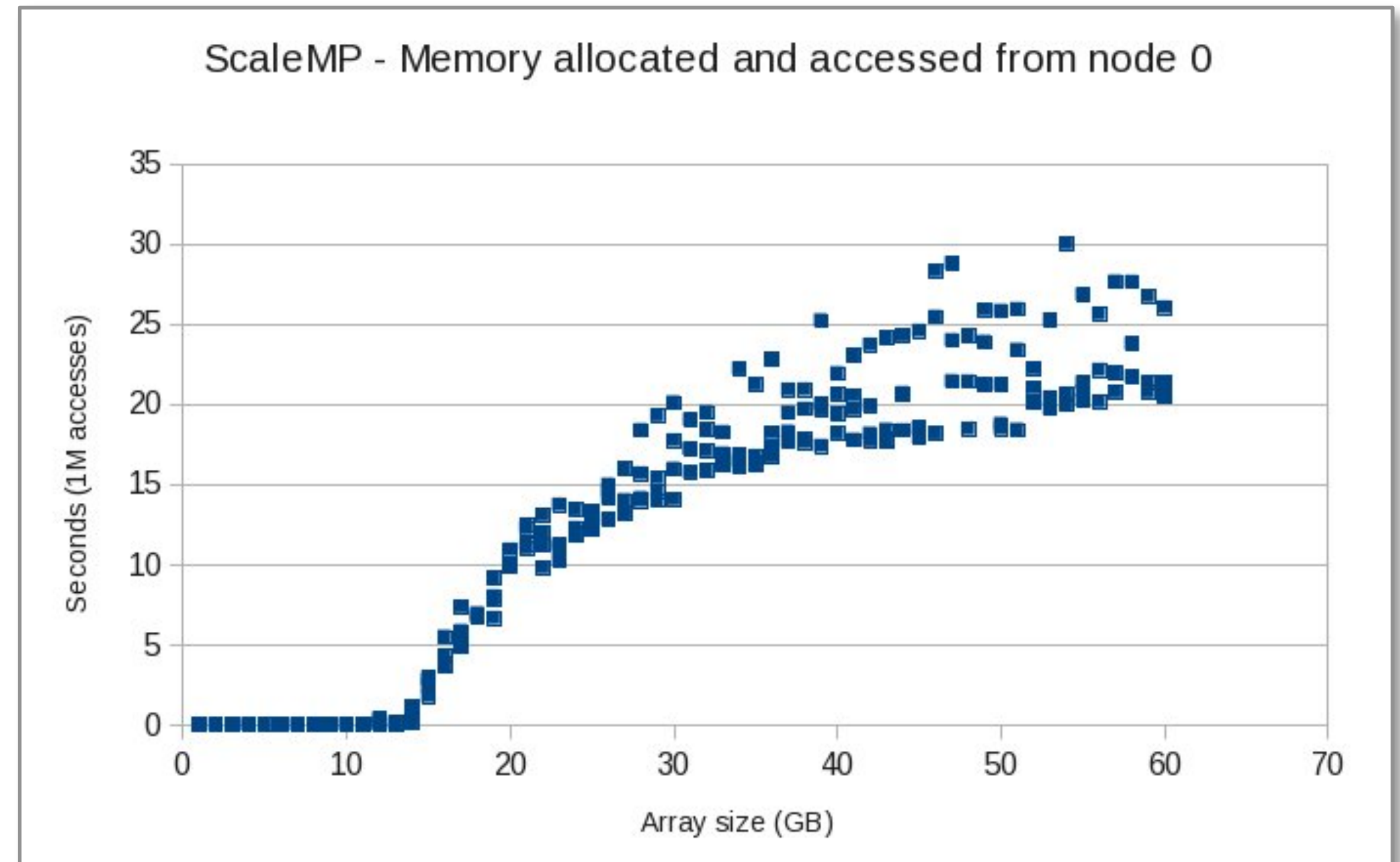
SVM EXAMPLE: SCALEMP

[Schmidl et al., Cluster 2010], ScaleMP performs well if:

- Few synchronizations
- Low number of memory allocations
- Suitable thread placement
- Suitable data placement

Experiment

- 8GB array on node 0, pre-touched, binding threads to different nodes, 1M accesses with 8kb stride
- About 25us latency penalty per access



Concluding remark: any SVM reverts to page-swapping in the case of short local memory

MULTI-THREADING ARCHITECTURES (MTA)

Don't minimize, tolerate!

LATENCY TOLERANCE: INTRODUCTION

Processor-Memory-Gap/Memory Wall: latency of memory access grows by more than 5x in 10years (in terms of processor cycles)

Additional sources for latency

- Multiprocessors: coherence actions require plenty of time

- NUMA systems: each hop adds about 50ns (doubling the memory access latency)

Caches reduce latency if enough locality is present

- Otherwise: see impact of coherence actions

Goal up to now:

- Reduce frequency of long-latency events

- Maintain high, scalable bandwidth

- Provide a convenient programming model

LATENCY TOLERANCE: INTRODUCTION

Latency reduction/minimization: minimize latency, however reduction to zero not possible

Latency tolerance/hiding: hide the (remaining) latency by overlapping it with computation or other useful tasks

Key: parallelism (independent tasks required)

Examples:

- Process switch upon high-latency disk access

- Message passing: non-blocking operations

- Write buffers hide most of a store miss latency

- Load latency can be hidden using multiple outstanding read operations (remember that ILP is limited)

- Speculative reads (prefetching)

HARDWARE MULTI-THREADING

Maybe the most versatile technique for latency tolerance

- No special software analysis or support required

- Invoked dynamically: handles unpredictable situations like cache misses

- Suitable for any long-latency event as long as it can be detected at run time (e.g., synchronization, instruction fetch, ...)

- No impact on the memory consistency model

Cons:

- Might require a certain number of threads to reach peak performance

- Requires substantial changes to processor architecture

HARDWARE MULTI-THREADING - HW REQUIREMENTS

For N-way multithreading (N threads can be active)

State replication (N times)

Program counter

Processor state: registers, re-order buffer, branch target register

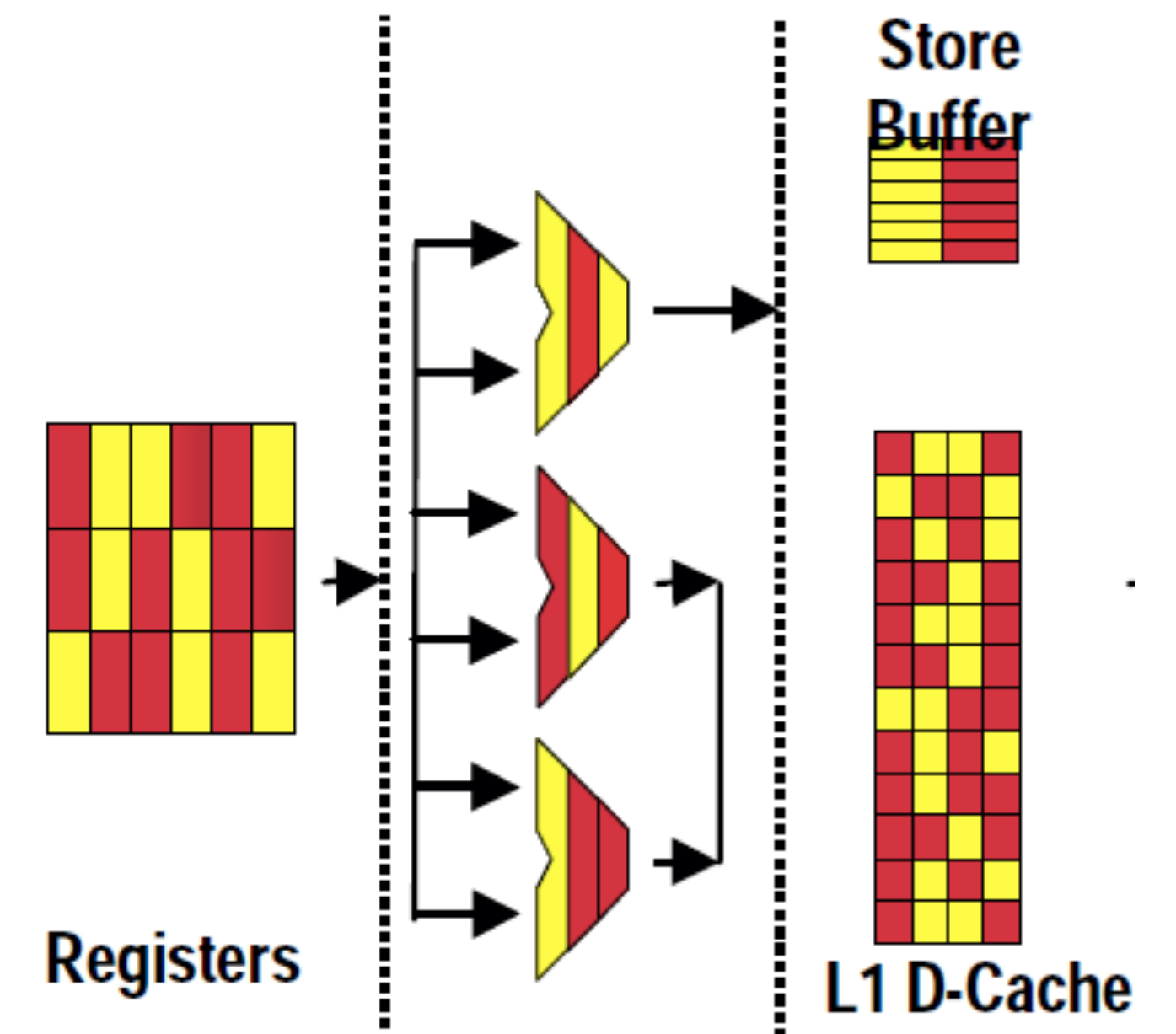
Register file

Saving/restoring context (N context sets)

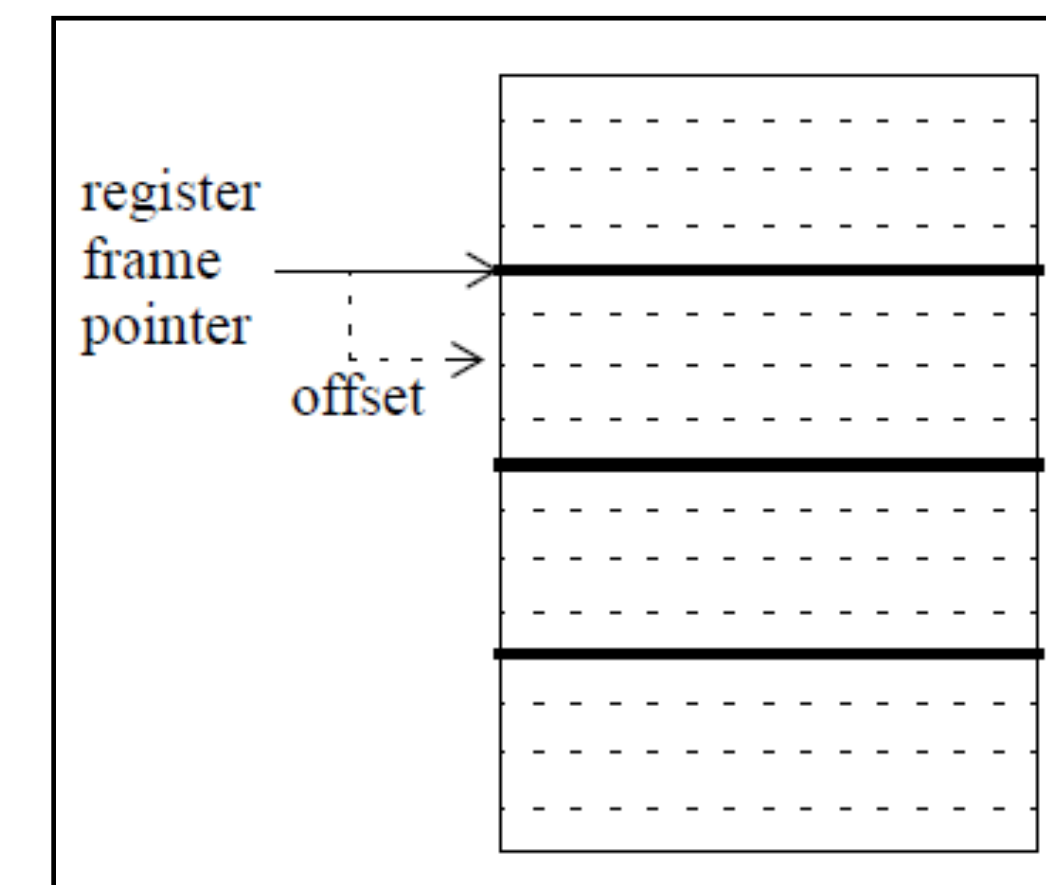
Saving/restoring latency determines switching overhead

Tag caches/TLBs (if present)

Note on VIPT (virtually indexed, physically tagged)



Intel Technology Journal Q1, 2002. Vol. 6 Issue 1



HARDWARE MULTI-THREADING

$$utilization = \frac{t_{busy}}{t_{busy} + t_{switching} + t_{idle}}$$

Blocked MT

Switch context only when a thread is blocked or stalled

Long-latency events

Context switch can last several cycles

Many implementation options

Context saving/restoring

Tagging

Pipeline is multi-threaded?

Low HW costs, good single-thread performance

Interleaved MT

Switch context every processor cycle

Choose an instruction from a different active thread every cycle

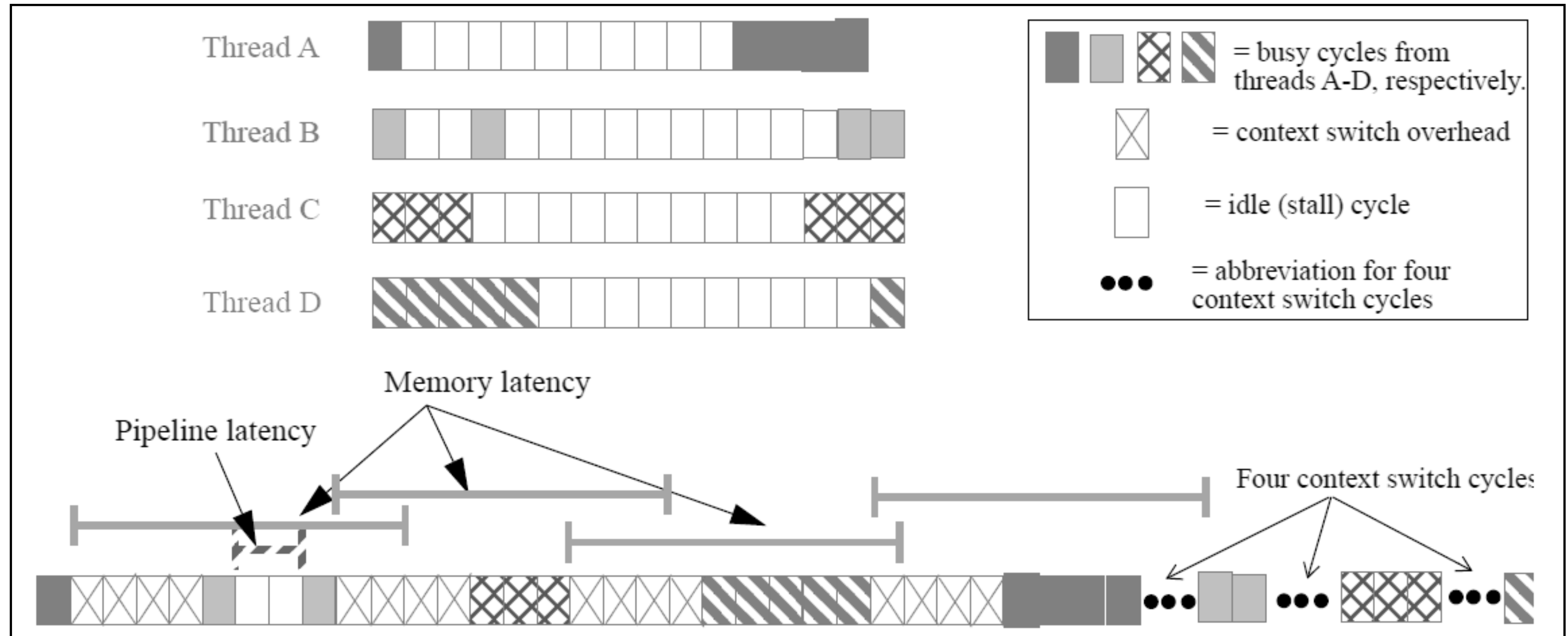
Minimal context switching costs required

Context saving/restoring not an option => tagging

Pipeline has to be multi-threaded

High HW costs, but simplified processor design, bad single-thread performance

BLOCKED MULTI-THREADING

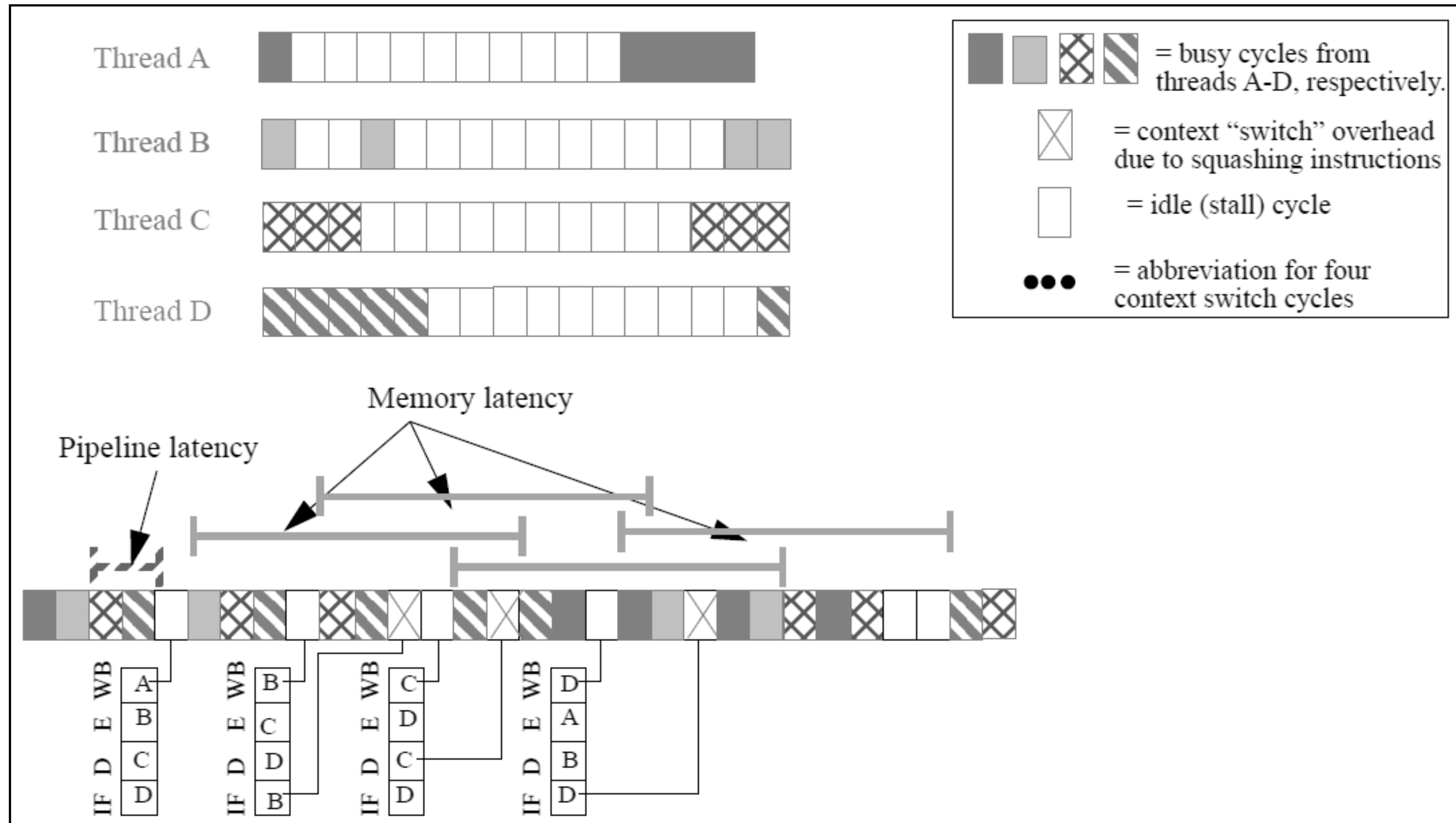


Thread A: cache miss with 10 cycles delay

Context switch: 4 cycles

Culler et al, Parallel Computer Architecture, MK 1999

INTERLEAVED MULTI-THREADING



SMT: COMBINING MULTI-THREADING WITH MULTI-ISSUE

Problems of traditional multiprocessors

ILP not sufficient to fill all issue slots within a cycle

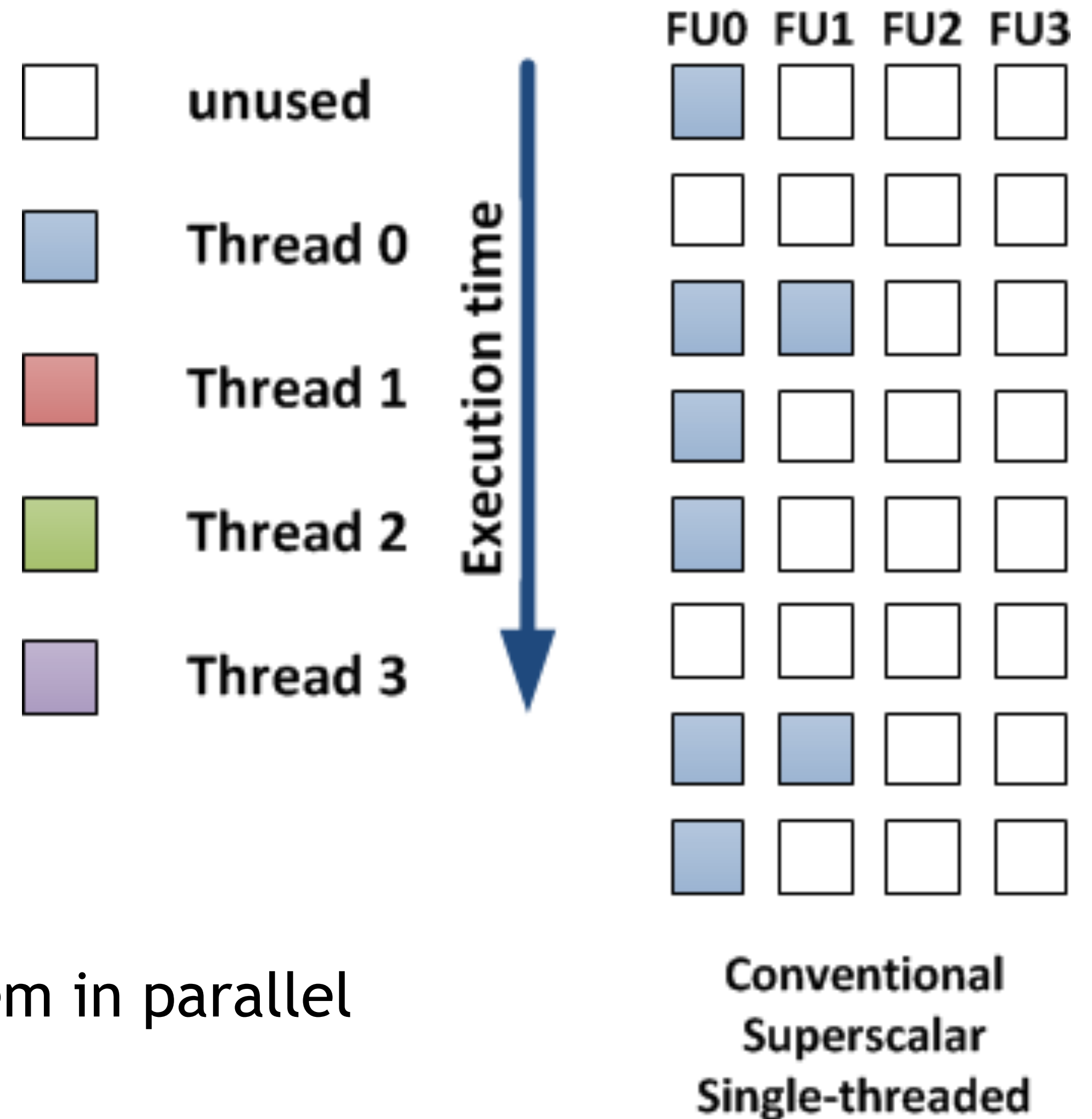
Many empty slots due to long-latency instructions and dependencies

Horizontal/vertical waste

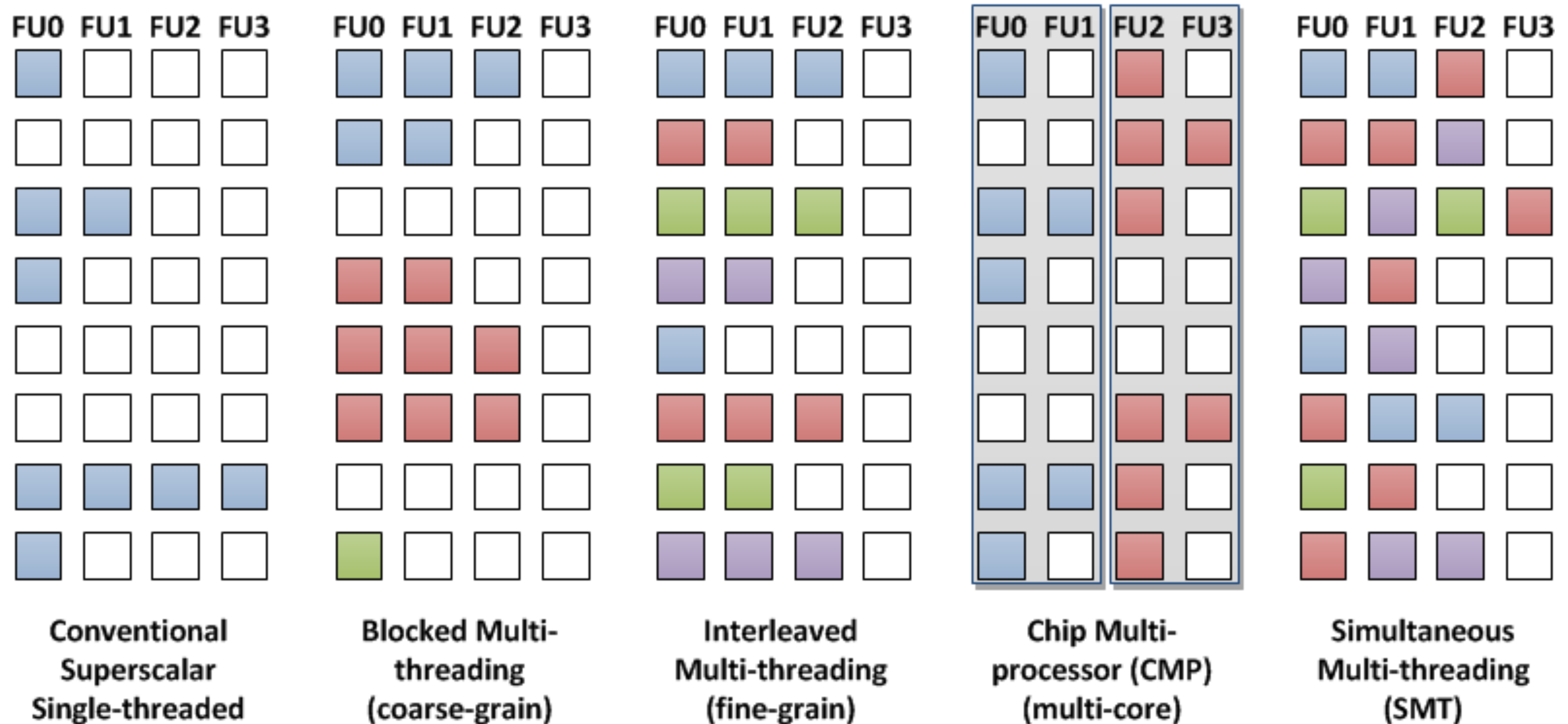
Simultaneous Multi-Threading (SMT)

Allow to fill issue slots from any thread

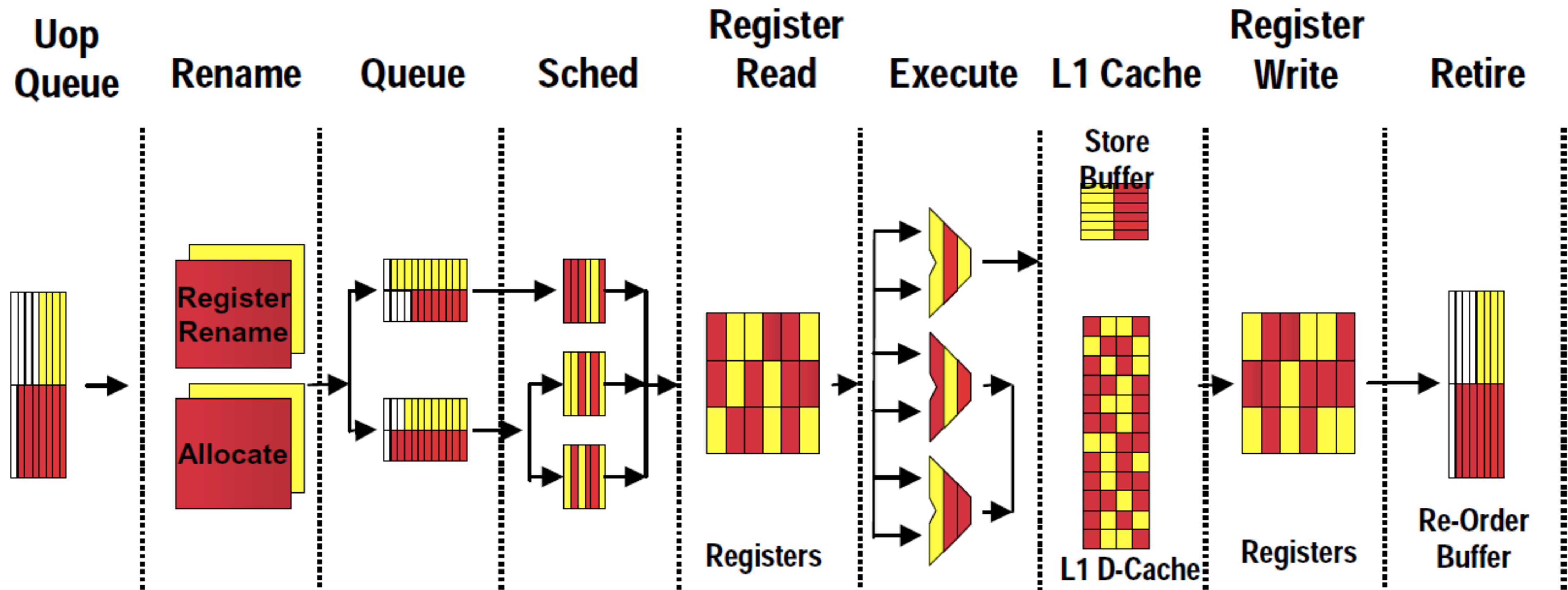
Use superscalar architecture to process them in parallel



SMT: COMBINING MULTI-THREADING WITH MULTI-ISSUE



INTEL'S HYPERTHREADING (P4, NEHALEM)



Intel Technology Journal Q1, 2002. Vol. 6 Issue 1

ANOTHER EXAMPLE: CRAY XMT

500 MHz single 64-bit Threadstorm processor

128 threads/processor

AMD Socket 940 compatible

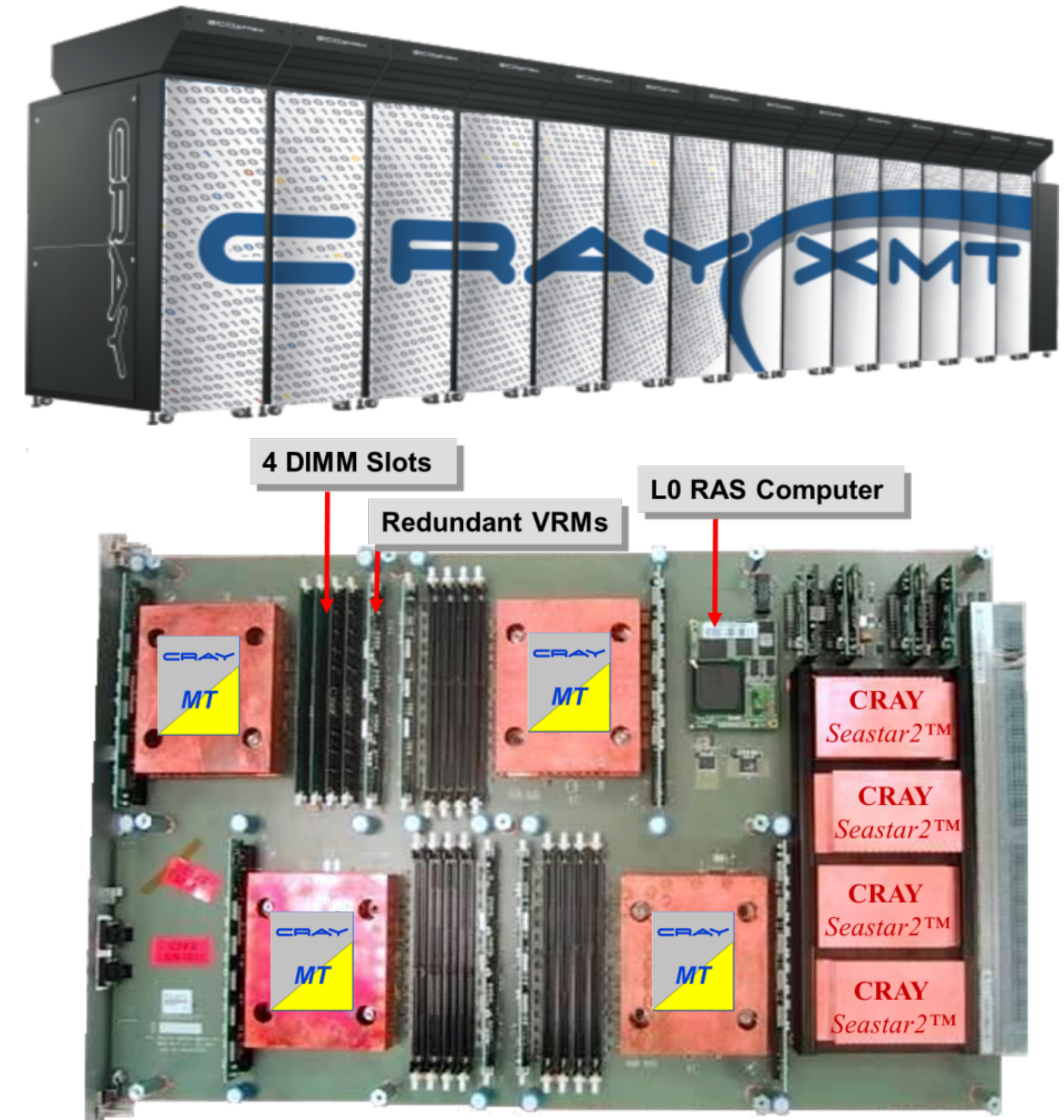
Max 8k processors/system

Same system/interconnect as message passing Cray system (SeaStar2)

Outstanding memory ops:

128P: 180

512P: 144



John Feo, Cray Inc., Eldorado presentation

REMINDER: LATENCY TOLERANCE TECHNIQUES

Property	Relaxed Consistency Models	Prefetching	Multi-Threading	Block Data Transfer
Types of latency tolerated	Write (blocking read processors) Read and write (dynamically scheduled processors)	Write Read	Write Read Synchronization	Write Read
Software requirements	Labeling synchronization operations	Predictability	Explicit extra concurrency	Identifying and orchestrating block transfers
Extra hardware support	Little	Little	Substantial	Not in processor, but in memory system
Supported in commercial systems?	Yes	Yes	Yes	(Yes)

NON-UNIFORM CACHE ARCHITECTURES (NUCA)

CACHE TRENDS

Core count is growing, demand for cache size is growing

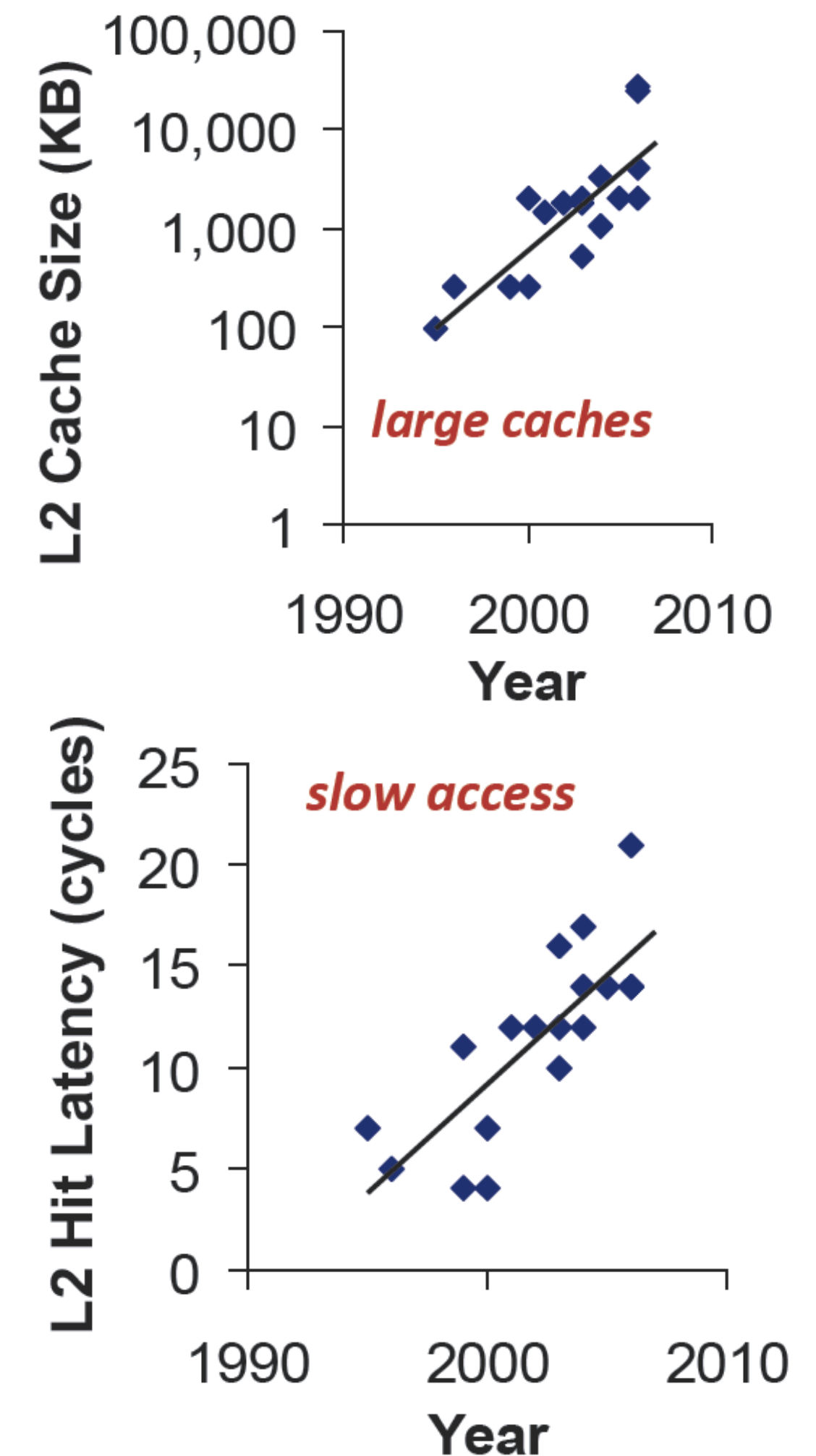
GBs of main memory, but only MBs of cache: $<0.1\%$ can be cached

On-chip communication delay remains constant across technologies

Wire delays dominate access latency of far-away cache slices

Large caches are slow caches

Cache design trade-offs: private vs. shared



Wenisch, EECS 570, U. Michigan

BASIC IDEA

Balance cache slice access with network latency

NUCA: non-uniform cache architecture by physically distributing cache slices across the die

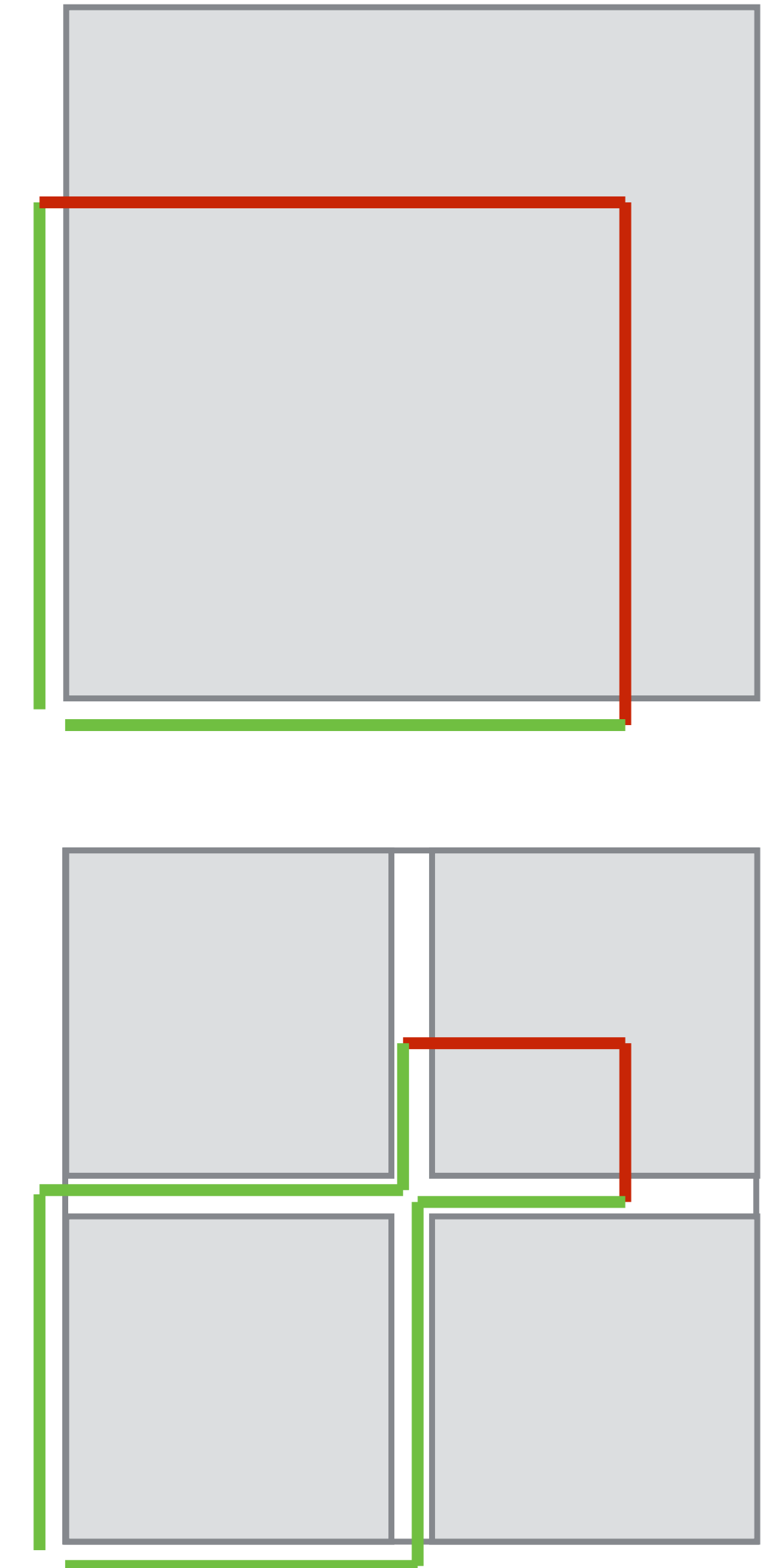
Range of latencies, several times higher in worst case

Goal: move cache blocks closer to the core

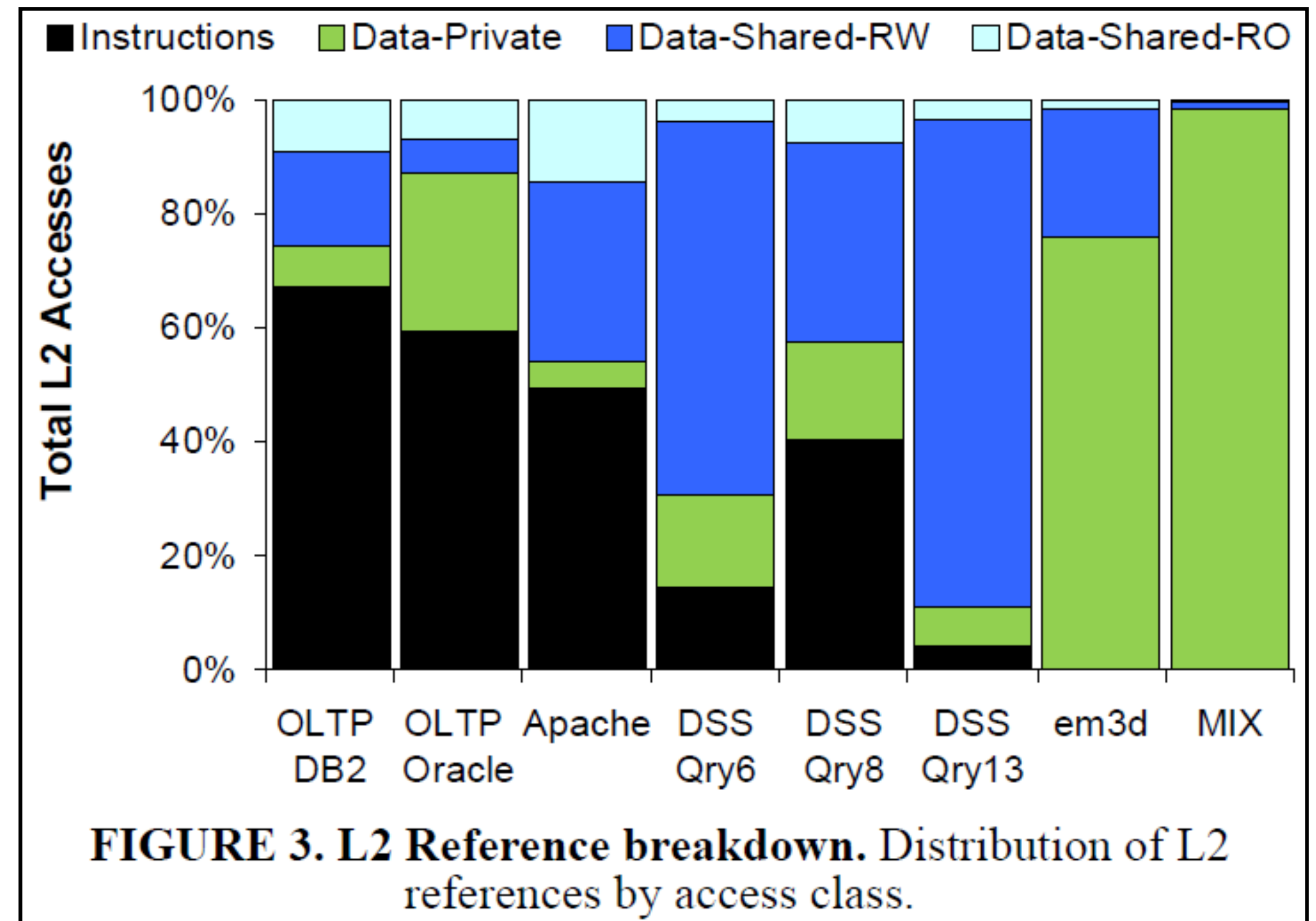
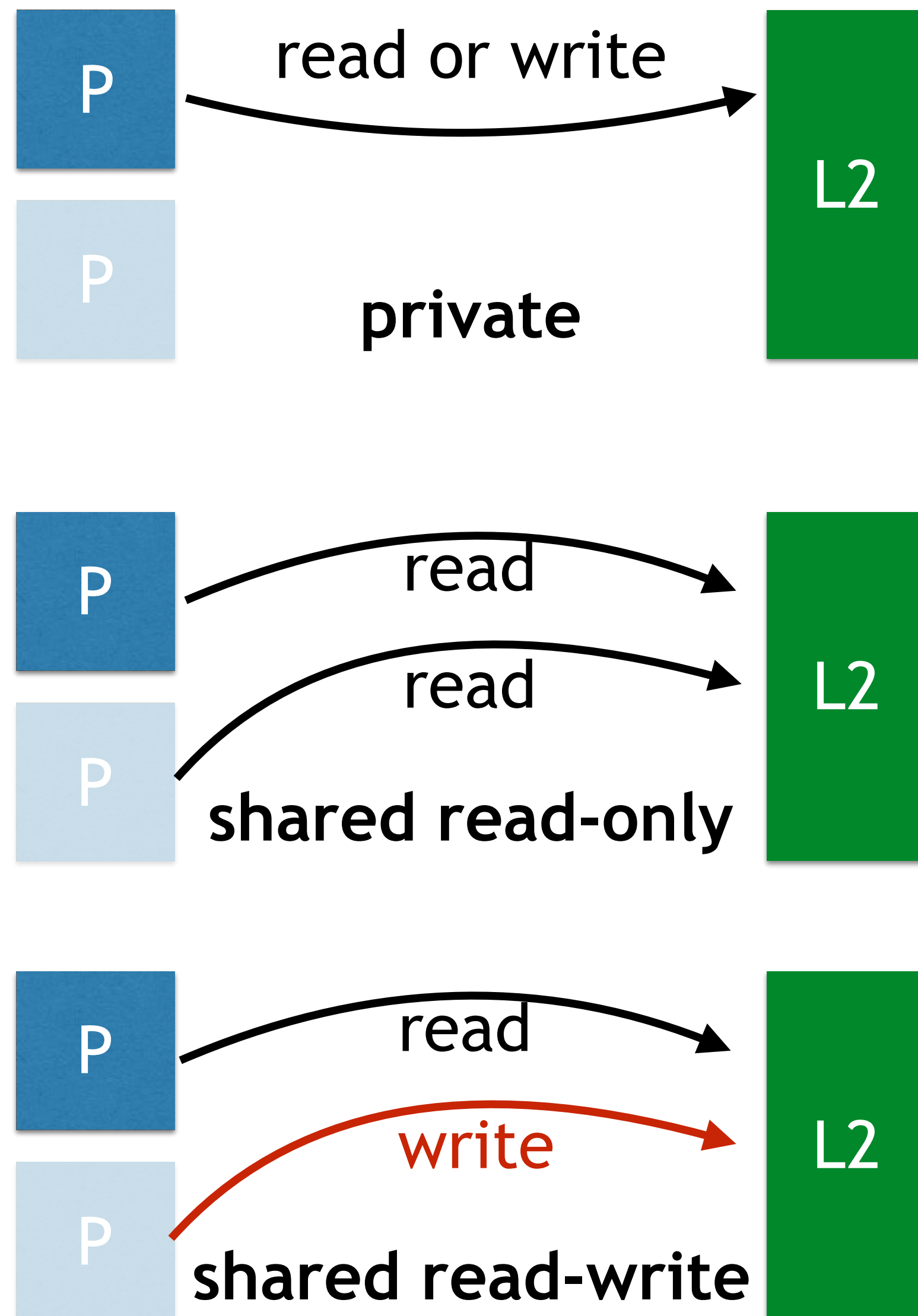
Previously: cache block determines location

Now: look-up mechanism required to find cache blocks

Main question: which cache block to put where?

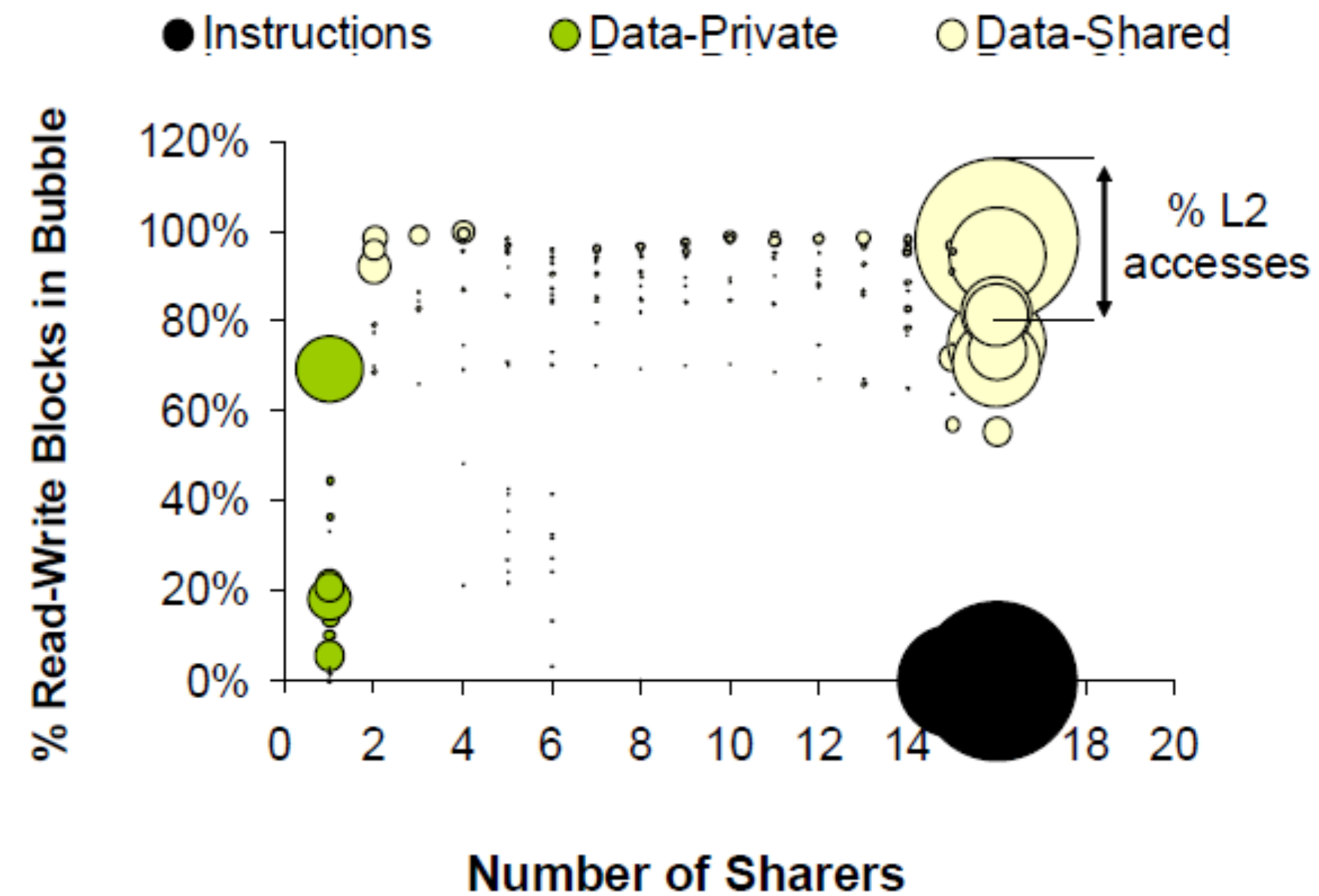


BASIC IDEA: CLASSIFICATION OF ACCESSES

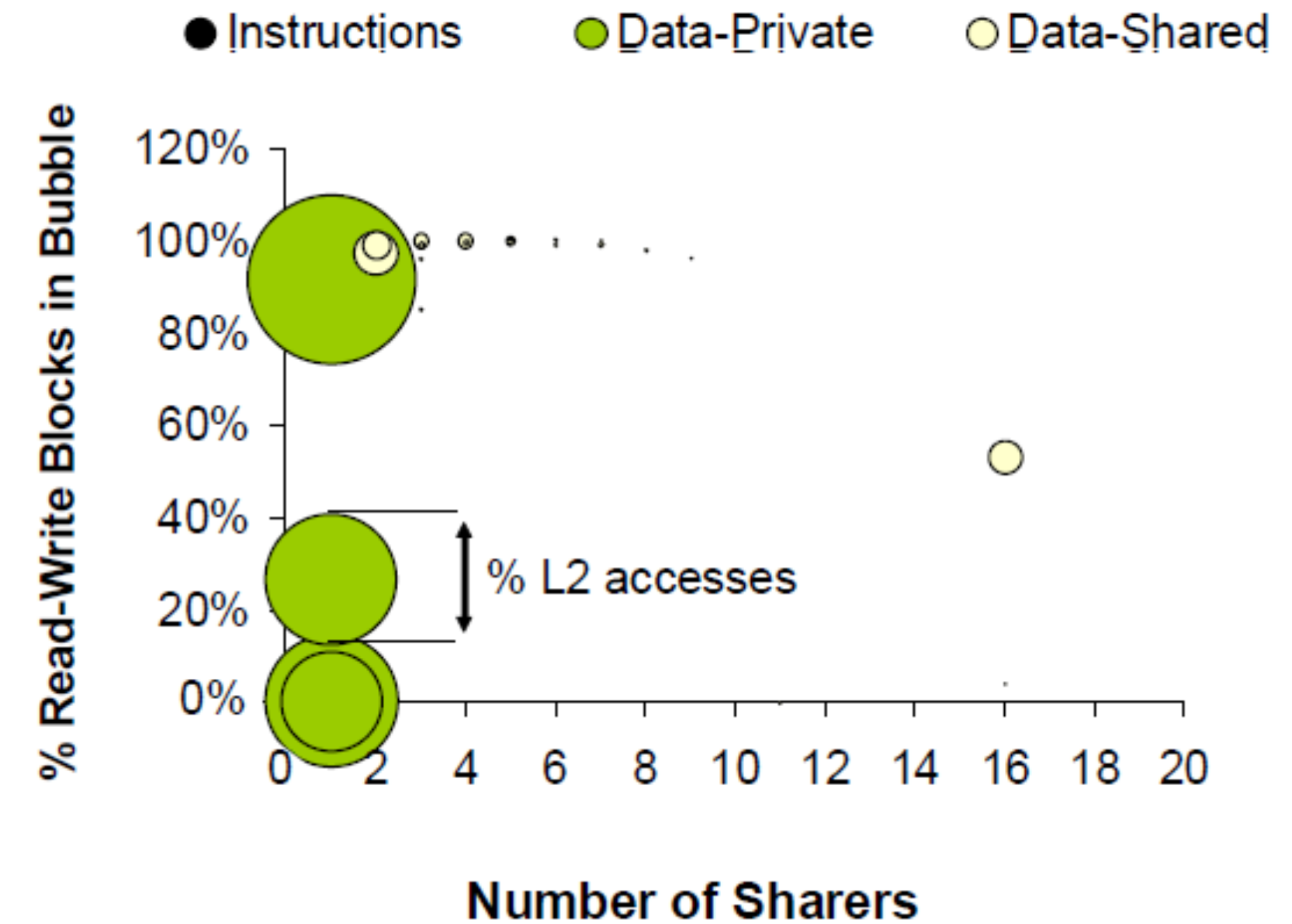


Hardavellas et al., ISCA 2009

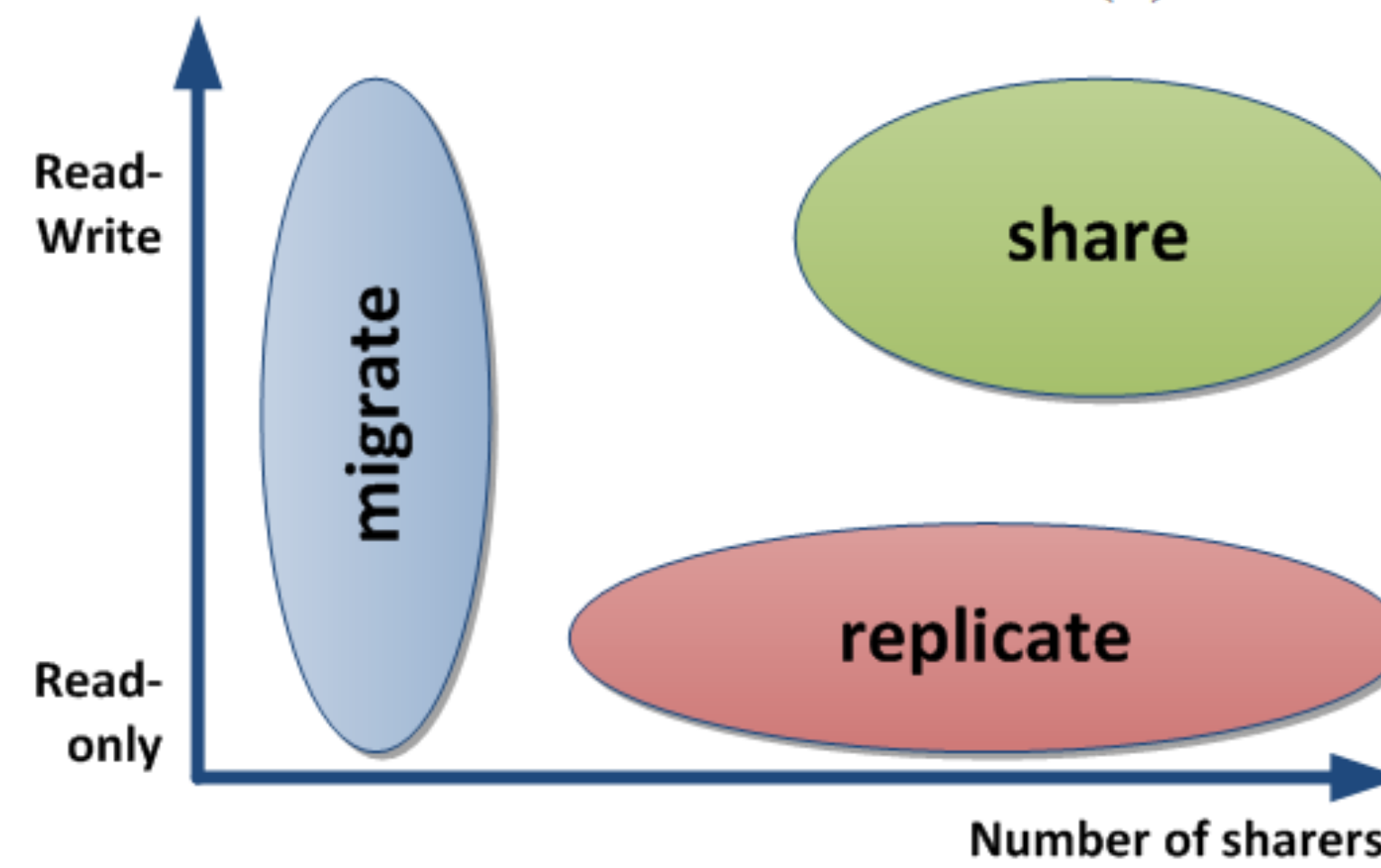
CLASSIFYING CACHE BLOCKS



(a) Server Workloads



(b) Scientific and Multi-programmed Workloads



CLASSIFYING CACHE BLOCKS

According to (typical) patterns (data and instructions)

1. Private data accessed by only one core (trivial) => Migrate?
2. Shared data universally shared by all cores, mostly read-write
=> Migrate or replicate?
3. Instructions are universally shared and read-only => Replicate?

For migration/replication: overhead > benefits

On average, a core accesses one block only once or twice before another core writes to it (invalidation)

Exception: instructions (read-only, no coherence)

Reactive NUCA: placement using standard address interleaving,
block's address determines location, unique locations (no replication)

REACTIVE NUCA DESIGN

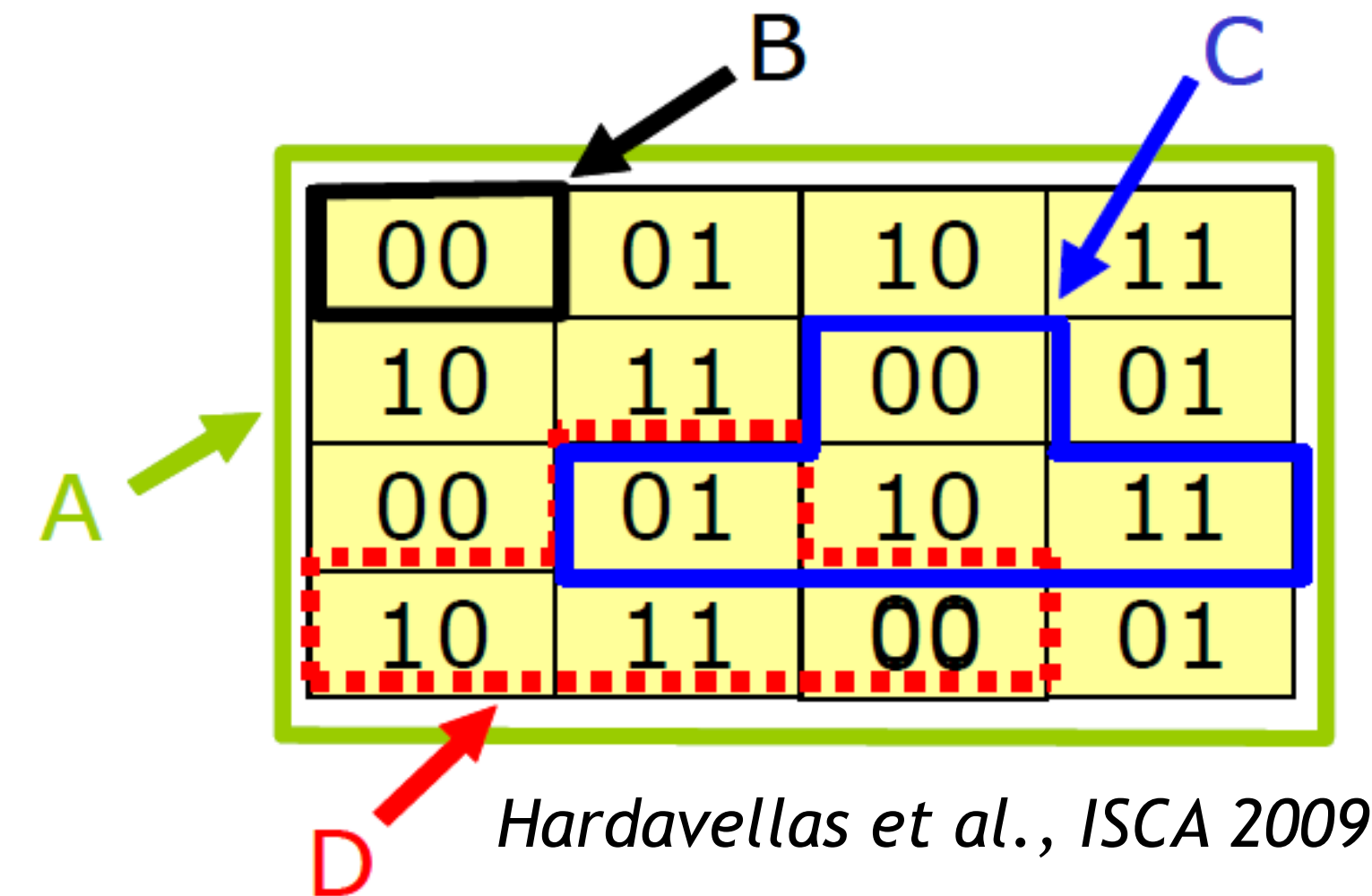
Conceptually: overlapping clusters of fixed-center clusters, power-of-2 sizes

Placement: cluster is selected based on the classification

Private data: size-1 clusters (B)

Shared data: size-16 clusters (A)

Instructions: size-4 clusters, replicated across the chip (C,D)



Hardavellas et al., ISCA 2009

Classification: done by OS at page granularity

Communicated to cores through TLB

Extend page table entries by private/shared bit

Reclassification: page marked as poisoned, blocks are invalidated and subsequent accesses get a new cluster ID

PERFORMANCE

P = private

fast access

A = adaptive selective
replication

Beckmann et al, MICRO
2006

S = shared

large capacity

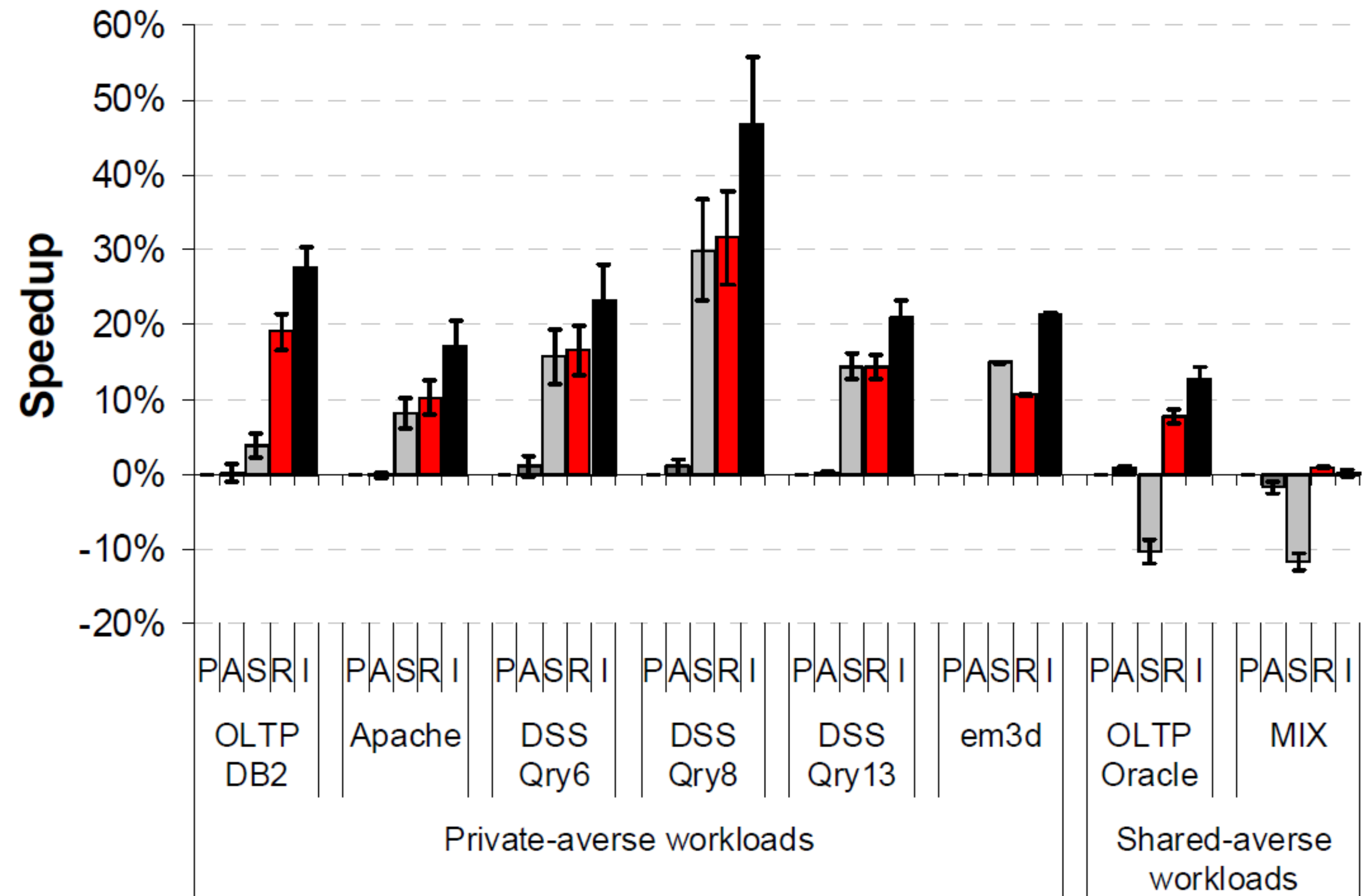
R = reactive NUCA

I = ideal

fast access & large
capacity

MIX: based on SPEC2000

DSS: decision support
system (TPC-H)



SUMMARY

SUMMARY

SVM & applied consistency models

- Shared memory over distributed resources

- Software-only solution

- Commercial: ScaleMP, Symmetric Computing; Research: Munin, Treadmarks, ...

Tolerating latency by multi-threading

- MTA are maybe the most versatile approach to tolerate latency

- Can be combined with latency reduction techniques

- various x86 (limited multi-threading); GPUs, Cray XMT (massive multi-threading)

NUCA

- Shared use of distributed cache slices

- Reactive NUCA addresses placement issue

- Migration/Replication of cache blocks

- Most recent multi-core designs make use of shared LLCs