ADVANCED PARALLEL COMPUTING
LECTURE 10 - APPLIED CONSISTENCY

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

RECAP: CONSISTENCY MODELS

Motivation: improved performance by (hopefully) increased memory parallelism and less processor idling

Relaxed consistency models: relax ordering guarantees (W2R, W2W, R2W, R2R), relax visibility guarantees (atomicity), relax transitivity (W->R/W->R)

Safety net: either synchronizing memory ops, or dedicated fences/ membars

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

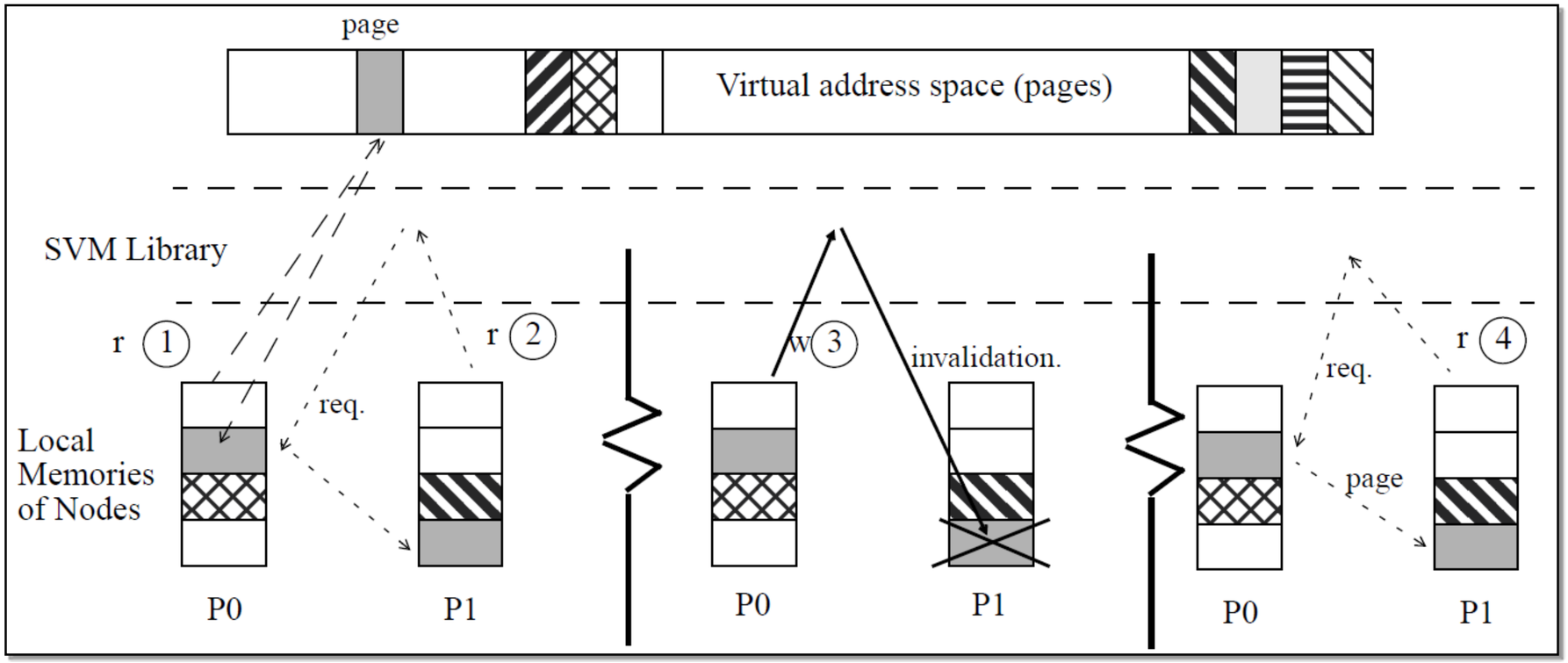
- Multi-level cache coherence: local caches do not have to be kept coherent across nodes

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

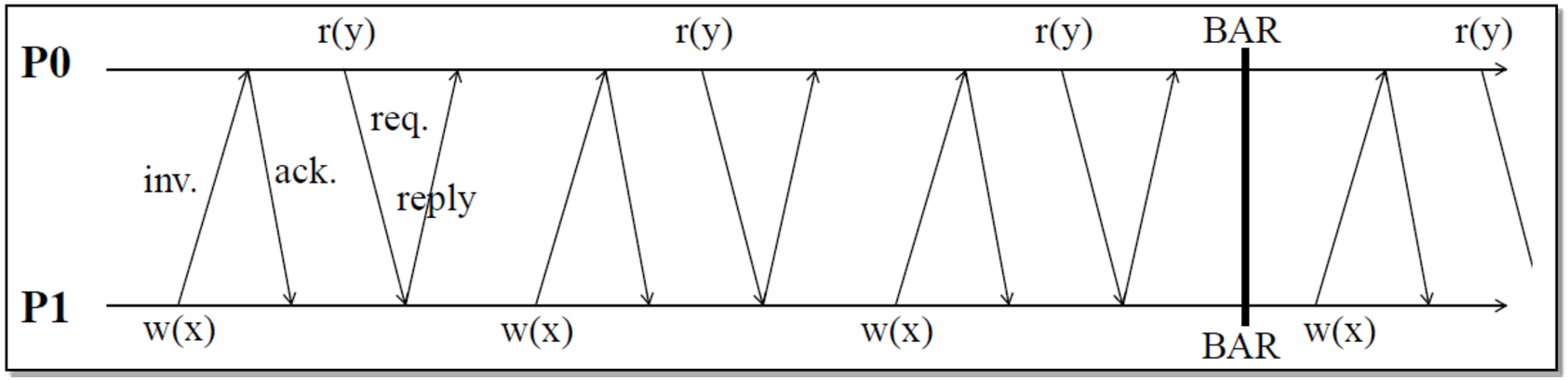
Directory or home node can hold information (state/copies)

MAIN PROBLEMS

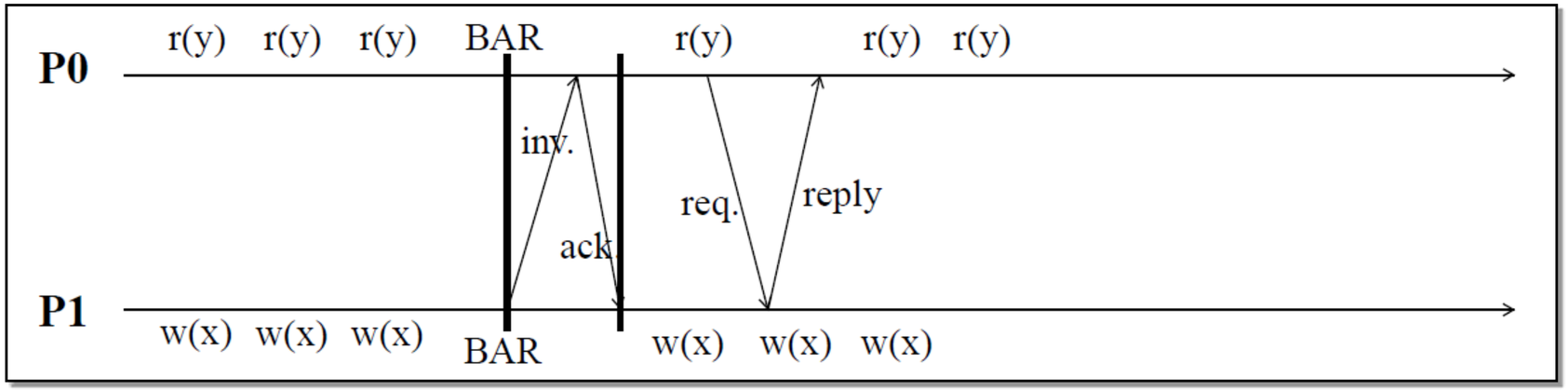
- High overheads of protocol invocation and processing
  - Handled in SW on a general-purpose processor
  - Page faults take time to cause an interrupt/trap to switch to OS
  - Communication done by message exchange
  - Incoming requests interrupt the processor and pollute caches, slowing down the currently 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



Culler et al, Parallel Computer Architecture, MK 1999

SVM RELAXED CONSISTENCY – DIFFERENT GOAL

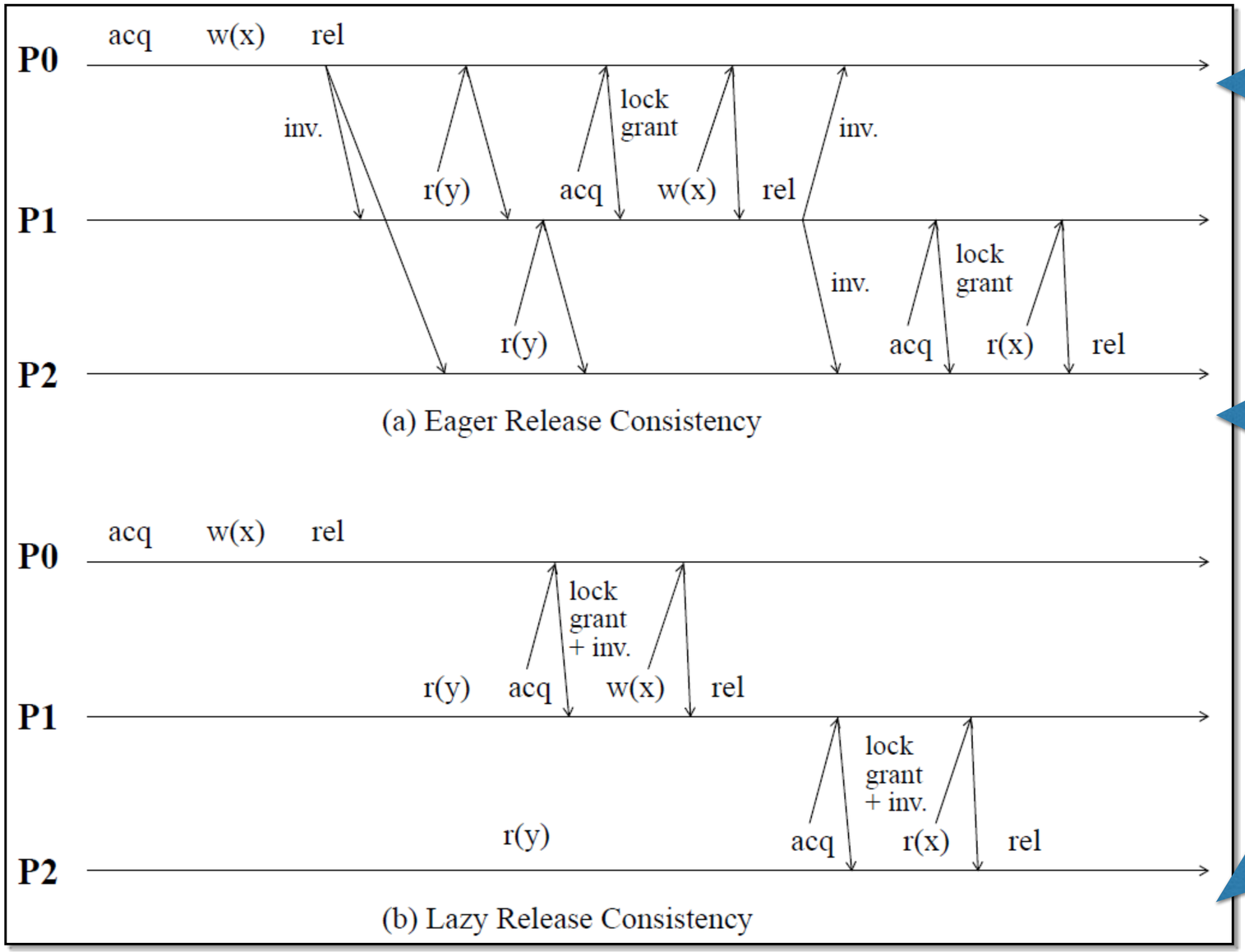
HW-coherent machines: goal is to reduce latency

- 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 sent 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!

ERC: works!
LRC: starvation!

P0

```
lock L1;
ptr = non_null_ptr_val;
unlock L1;
```

P1

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

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 (-)

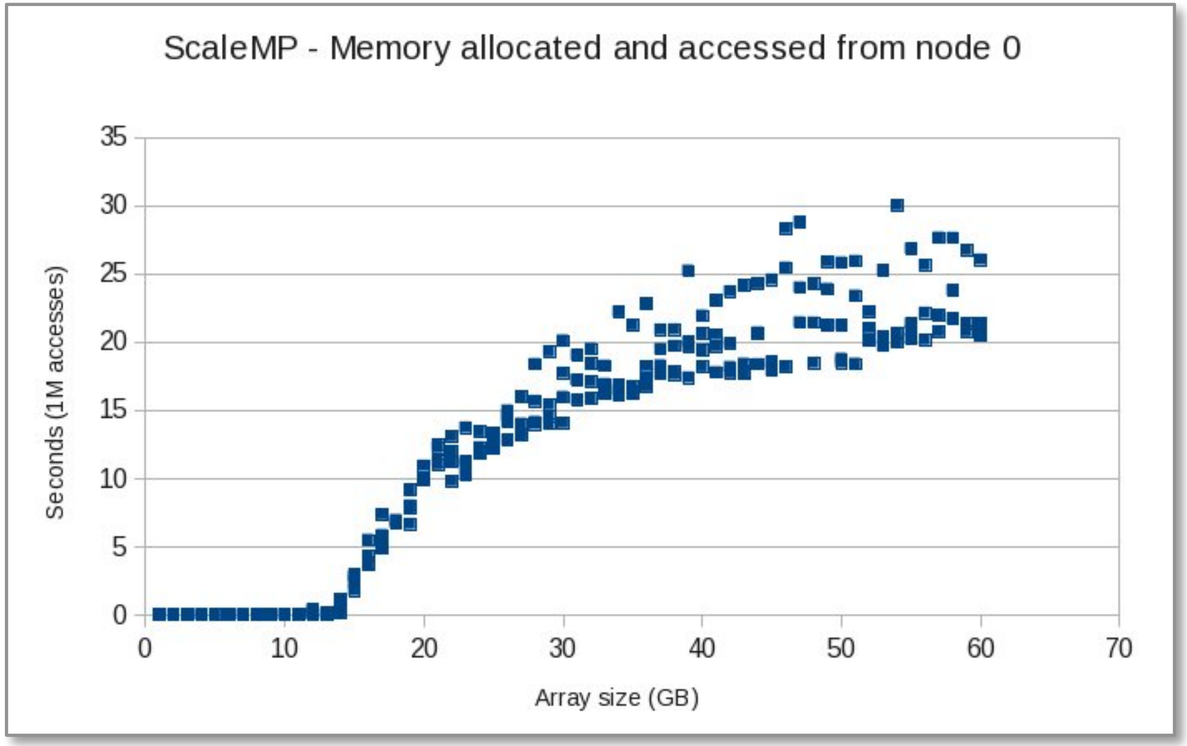
SVM EXAMPLE: SCALEMP

[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

C++11 CONSISTENCY MODEL

DATA-RACE-FREE

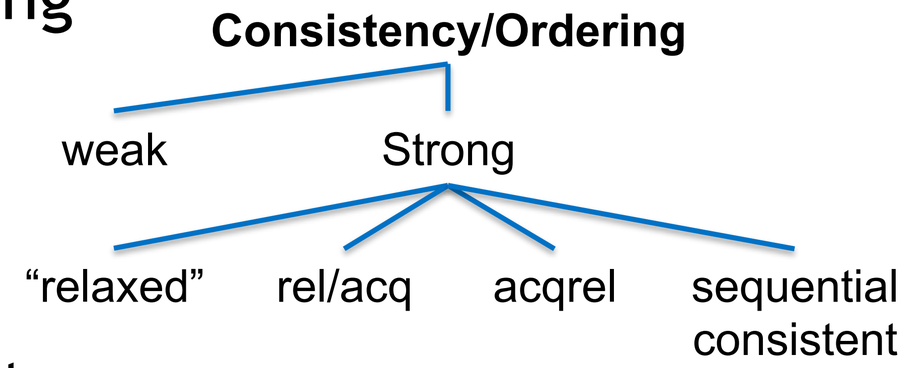
DRF: all data races are protected using strong memory operations

- Weak mem ops = all other

Based on release consistency

- A read-acquire executes before all reads and writes by the same thread that follow it in program order.
- A write-release executes after all reads and writes by the same thread that precede it in program order.
- [Formal definitions courtesy of Herb Sutter]

Transitivity/cumulativity: all history is visible, based on read-from relations



MEMORY ORDERING AT COMPILE TIME

```
int A, B;
void foo()
{
    A = B + 1;
    B = 0;
}
```

`gcc -S -masm=intel -O2 t.c`

Fine for single-threaded programs

Potentially broken for multi-threaded

-O0

```
...
mov eax, DWORD PTR B[rip]
add eax, 1
mov DWORD PTR A[rip], eax
mov DWORD PTR B[rip], 0
...
```

-O2

```
...
mov eax, DWORD PTR B[rip]
mov DWORD PTR B[rip], 0
add eax, 1
mov DWORD PTR A[rip], eax
...
```

```
int A, B;
void foo()
{
    A = B + 1;
    asm volatile("" ::: "memory");
    B = 0;
}
```

-O2

```
...
mov eax, DWORD PTR B[rip]
add eax, 1
mov DWORD PTR A[rip], eax
mov DWORD PTR B[rip], 0
...
```

OUT-OF-THIN-AIR STORES

```
int A, B;
void foo()
{
    if (A)
        B++;
}
```

```
void foo()
{
    register int r = B;     // promoting B to a register
    if (A)
        r++;
    B = r;                  // new memory store
}
```

Previously allowed, now forbidden in C++11

Hans Boehm, http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

MEMORY ORDERING AT RUNTIME

Ambiguous synchronization point

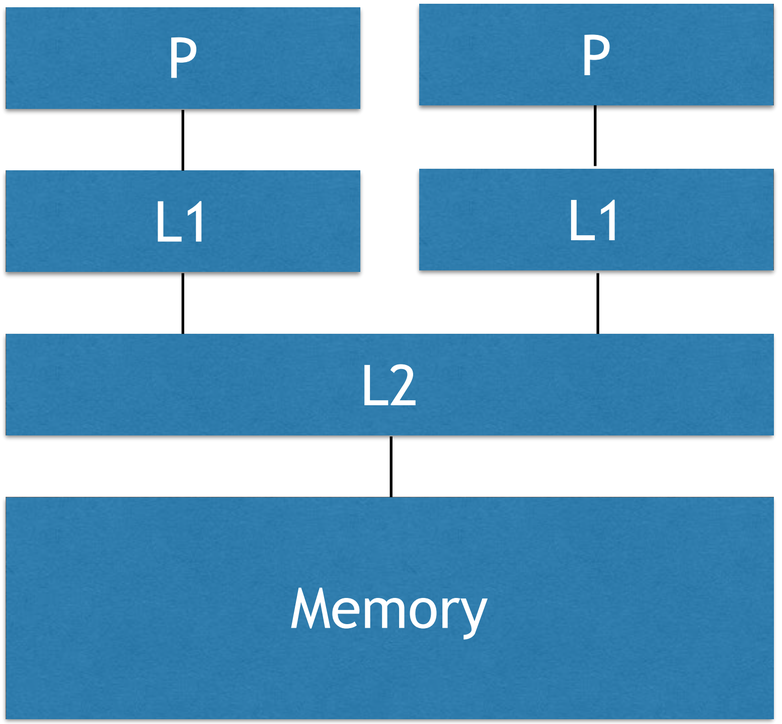
- Banked SRAM/DRAM
- Multiple memory controllers
- Unordered network
- …

Tool: memory barriers (aka fences)

- {R,W}2{R,W}
- E.g., W2R: all stores performed before the barrier are visible & all loads performed after the barrier see the latest visible value
- Often non-local operation

Data dependency barrier

- Weaker form of read barrier
- Resolves: two loads with a data dependency (second load depends on the result of the first)
- Partial ordering, usually local operation



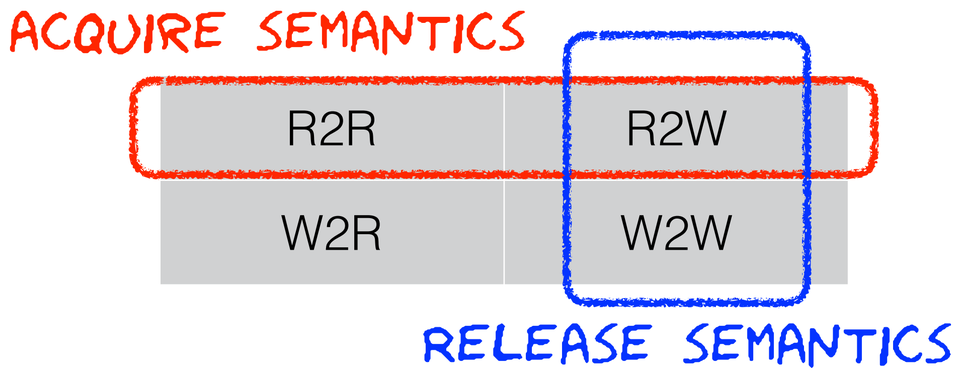
R2R	R2W
W2R	W2W

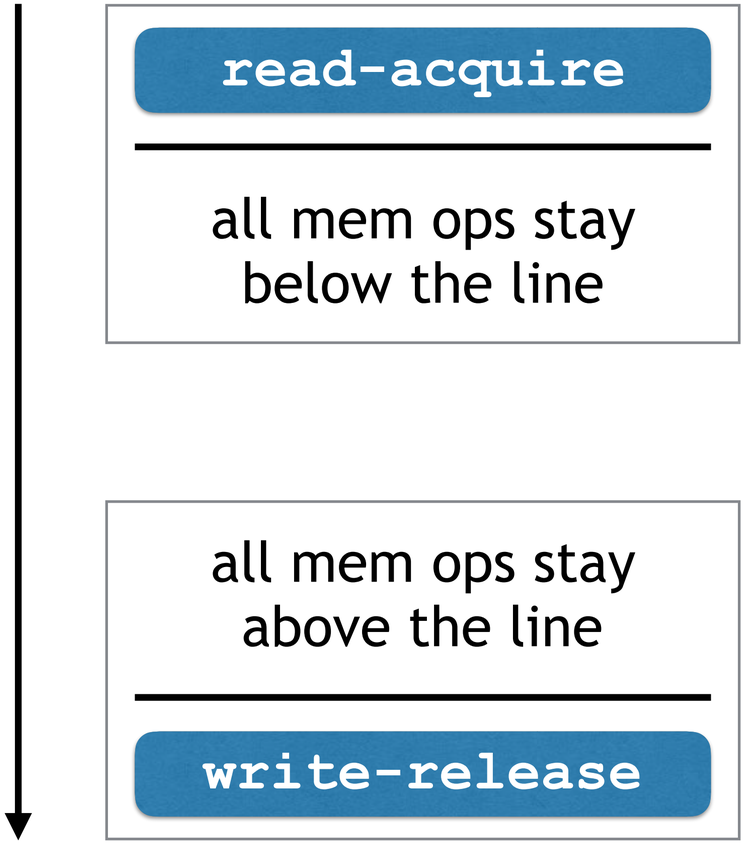
ACQUIRE AND RELEASE SEMANTICS

Formal definitions, courtesy of Herb Sutter:

A read-acquire executes before all reads and writes by the same thread that follow it in program order.

A write-release executes after all reads and writes by the same thread that precede it in program order.





PROCESSOR OVERVIEW

	R2W	R2R	W2W	W2R	Data dependency orders loads?	Conditional atomic	Other atomics	Atomics provide barrier?
x86	no-op	no-op	no-op	mfence, cpuid, locked i.	yes	CAS: cmpxchg	xchg, locked instructions	full
arm	dmb	dmb	dmb-st	dmb	indirection only	LL/SC: ldrex/strex		target only
ppc (XBOX360)	lwsync	lwsync	lwsync	hwsync	indirection only	LL/SC: ldarx/stwcx		target only
alpha	mb	mb	wmb	mb	no	LL/SC: ldx_l/stx_c		target only

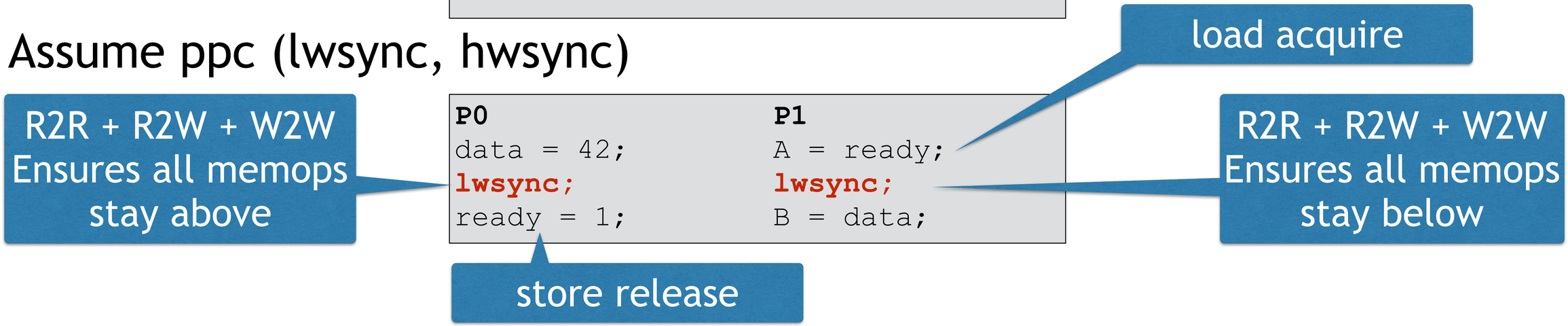
http://gee.cs.oswego.edu/dl/jmm/cookbook.html, some corrections

PLATFORM-SPECIFIC PRODUCER-CONSUMER

```
//global variables
int data = 0;
int ready = 0;
```

```
P0                      P1
data = 42;              A = ready;
ready = 1;              B = data;
```

Assume ppc (lwsync, hwsync)



No problems with write atomicity (assuming plain `int`)

Portability?

PORTABLE C++ (WITH AND WITHOUT FENCES)

```
//global variables
int data = 0;
atomic<int> ready(0);
```

P0	P1
`data = 42;`	`A = ready.load(memory_order_relaxed);`
`atomic_thread_fence(memory_order_release);`	`atomic_thread_fence(memory_order_acquire);`
`ready.store(1, memory_order_relaxed);`	`B = data;`

`memory_order_relaxed` in the example: ensure these ops are atomic, but don't impose any ordering constraints

Without fences

P0	P1
`data = 42;`	`A = ready.load(memory_order_acquire);`
`ready.store(1, memory_order_release);`	`B = data;`

HOW TO IMPLEMENT A MEMORY BARRIER?

Mainly depends on ordering guarantees

Ordered network

- Watermark in the buffer

Unordered network

- Track acknowledgements to identify outstanding transactions

DISCUSSION

Model: data-race-free. simple, easy to reason about

Hardware: complex, different types of memory barriers

- In particular guaranteed visibility can have huge associated costs
- In general: local vs. non-local operations
- Ordered vs. unordered paths (network)

Open issues

- Source object: in terms of code region (memory ops)
- => Object fences (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4522.html)
- Target scope: in terms of hardware resources (physical target)
- => Scoped consistency models (OpenCL, CUDA, HRF)

Memory fences are heavy

- Fences are a nice solution to the transitivity problem, but like a hammer
- Many advantages for passing ownership (message passing is also a consistency model)

SUMMARY

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)

David E. Culler, Jaswinder Pal Singh, Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann,1998

SUMMARY

SVM & applied consistency models

- Shared memory over distributed resources
- Software-only solution
- Commercial: ScaleMP, Symmetric Computing; Research: Munin, Treadmarks, …

C++ and data-race-free

- Sane consistency model (one of the first)
- Portable abstraction, easy reasoning (mostly)
- Moves responsibility of choosing the right memory bar to the compiler

ADDITIONAL MATERIAL

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)

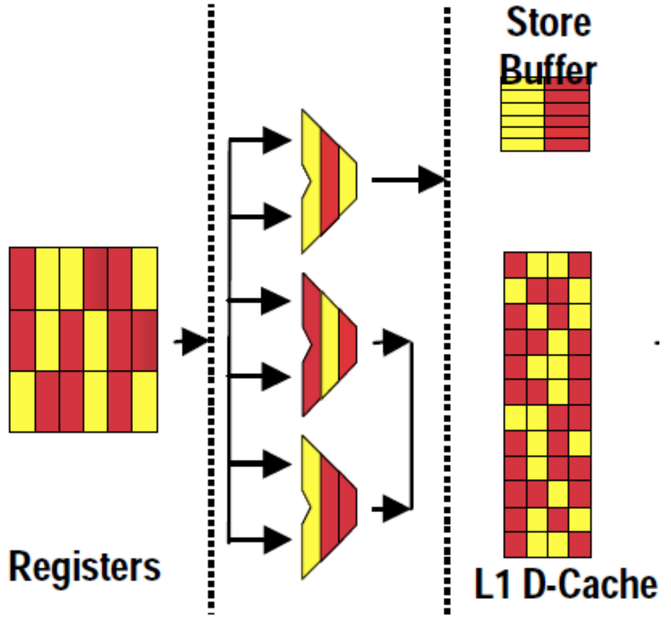
- Program counter
- Processor state: registers, re-order buffer, branch target register
- Register file

Saving/restoring context (N context sets)

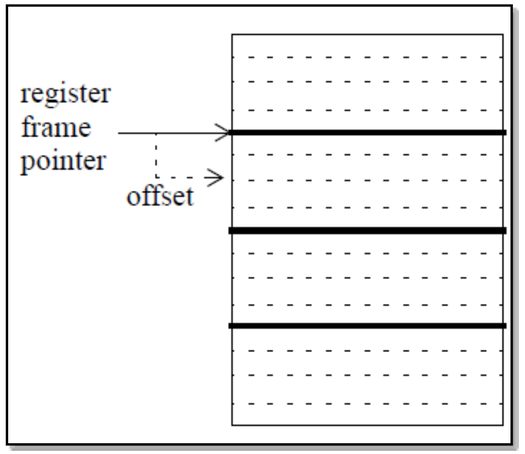
- Saving/restoring latency determines switching overhead

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

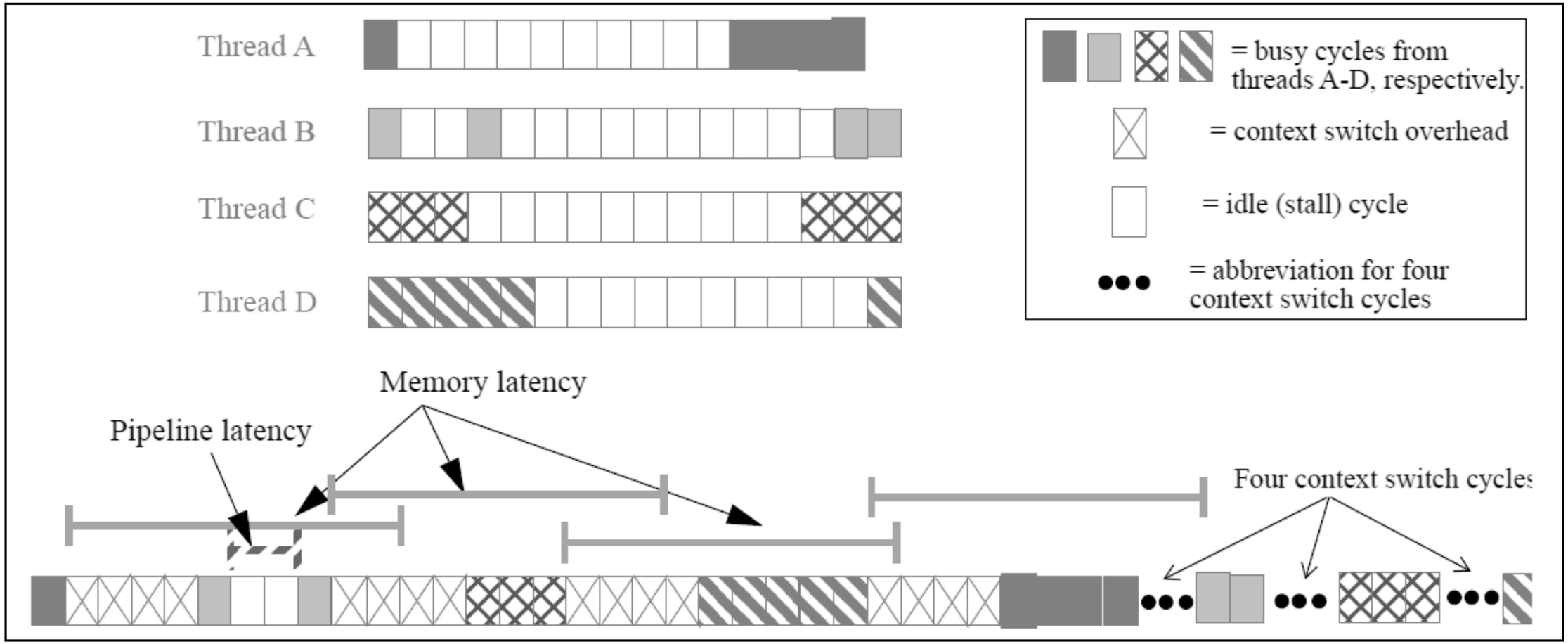
Minimal context switching costs required

- 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

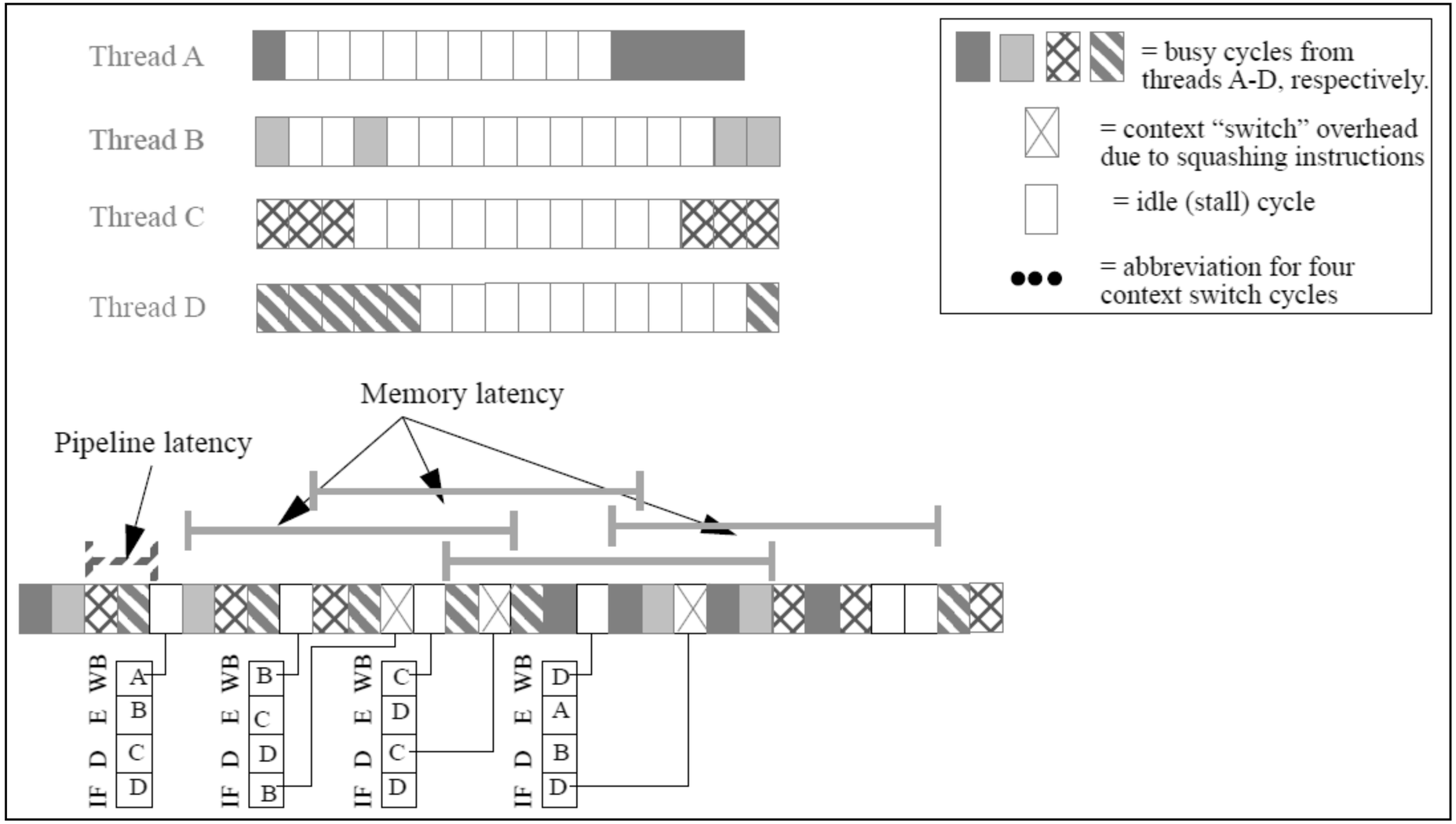


Thread A: cache miss with 10 cycles delay

Context switch: 4 cycles

Culler et al, Parallel Computer Architecture, MK 1999

INTERLEAVED MULTI-THREADING



Culler et al, Parallel Computer Architecture, MK 1999

SMT: COMBINING MULTI-THREADING WITH MULTI-ISSUE

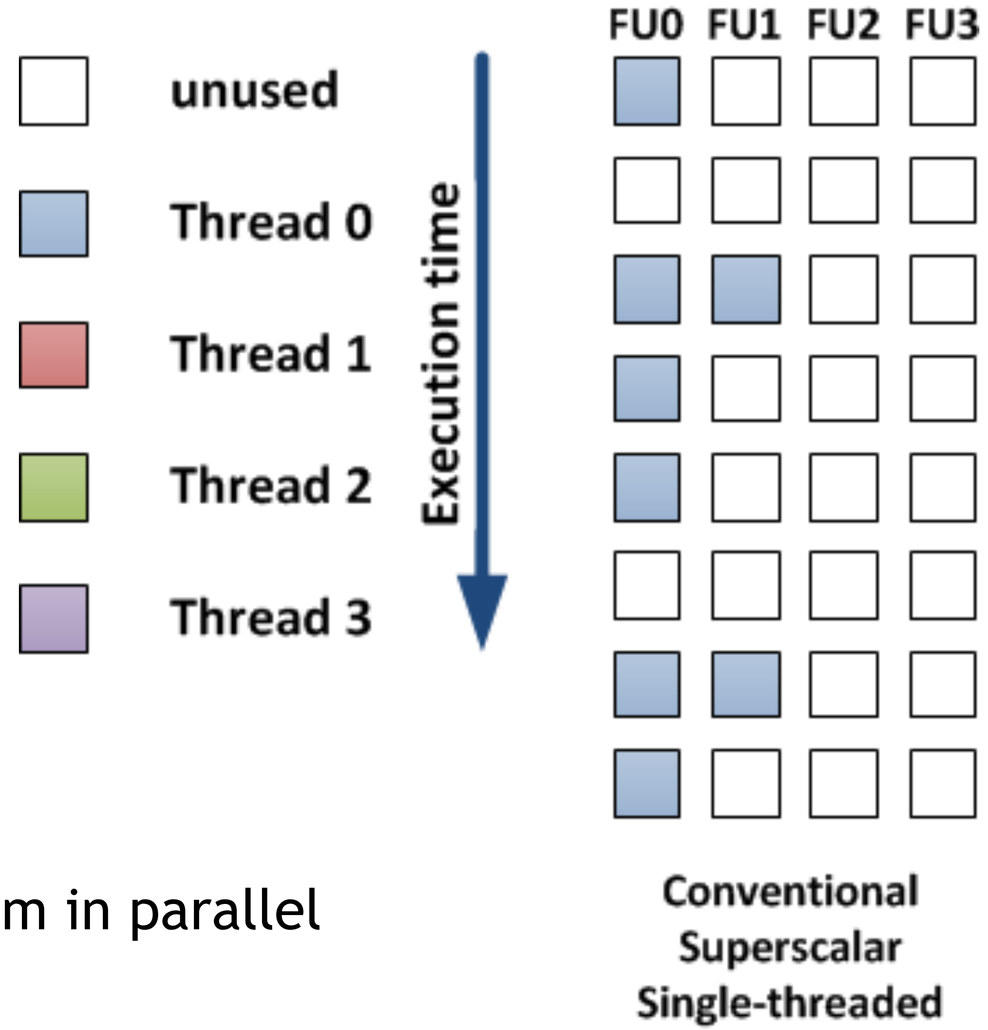
Problems of traditional multiprocessors

- ILP not sufficient to fill all issue slots within a cycle
- Many empty slots due to long-latency instructions and dependencies

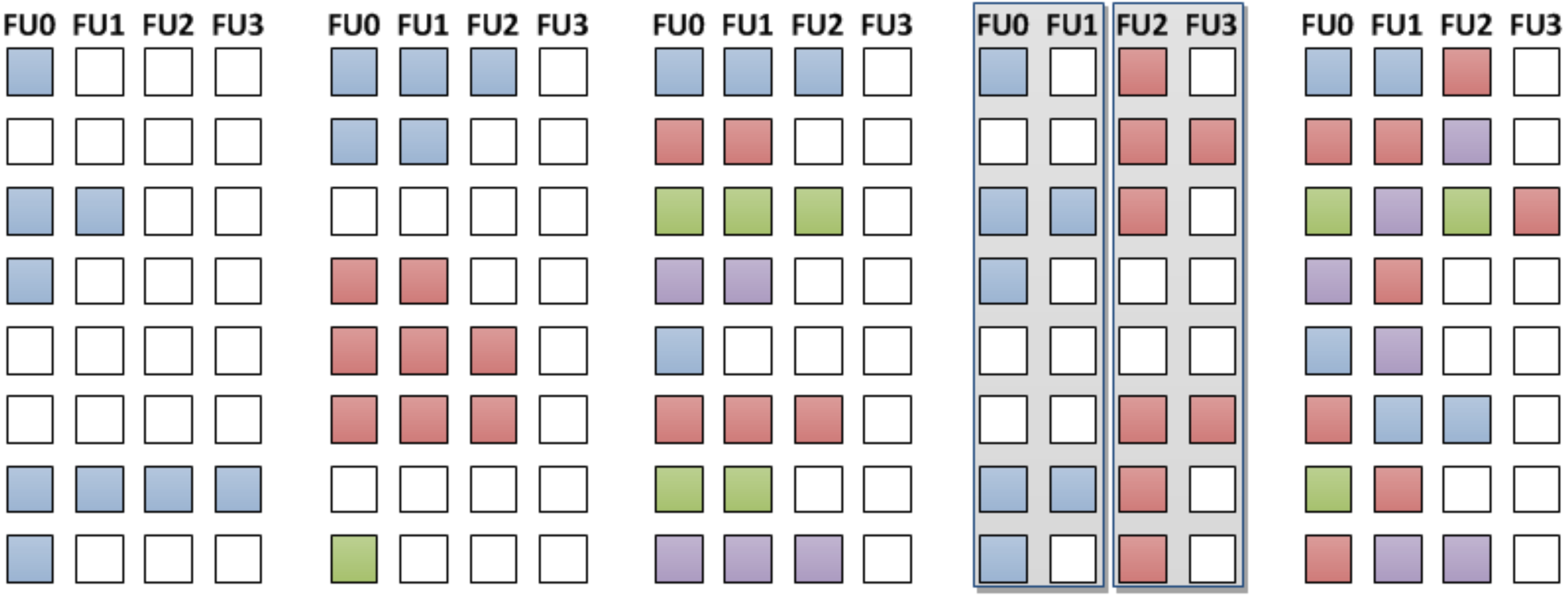
Horizontal/vertical waste

Simultaneous Multi-Threading (SMT)

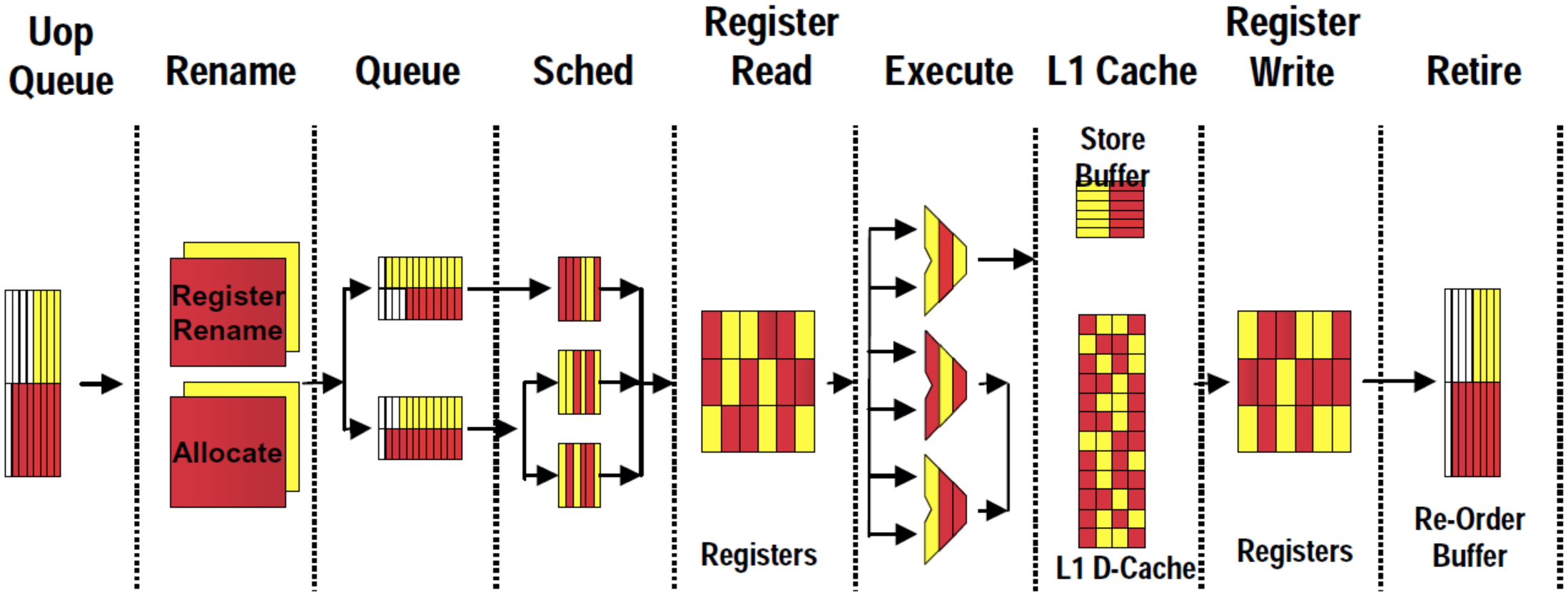
- Allow to fill issue slots from any thread
- Use superscalar architecture to process them in parallel



SMT: COMBINING MULTI-THREADING WITH MULTI-ISSUE



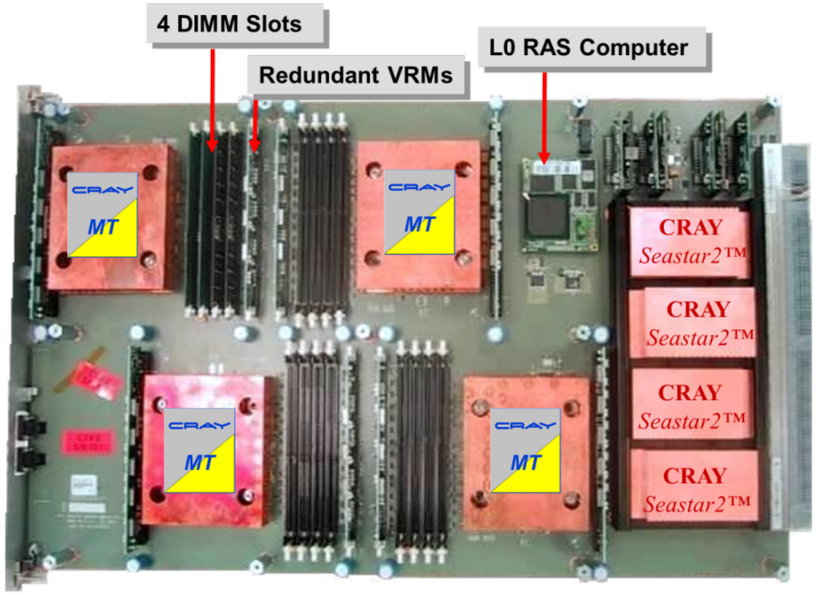
INTEL'S HYPERTHREADING (P4, NEHALEM)



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)

David E. Culler, Jaswinder Pal Singh, Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann,1998

NON-UNIFORM CACHE ARCHITECTURES (NUCA)

CACHE TRENDS

Core count is growing, demand for cache size is growing

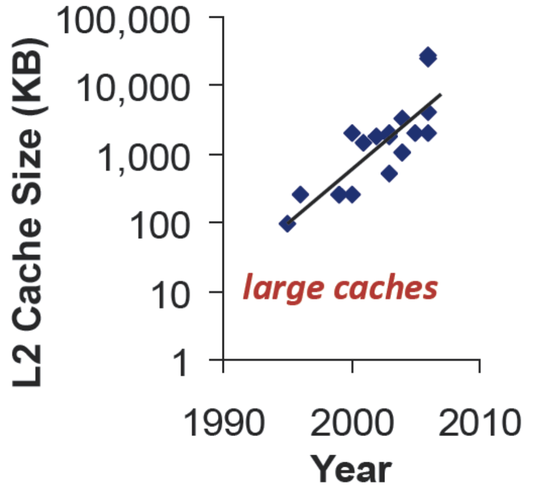
- GBs of main memory, but only MBs of cache: <0.1% can be cached

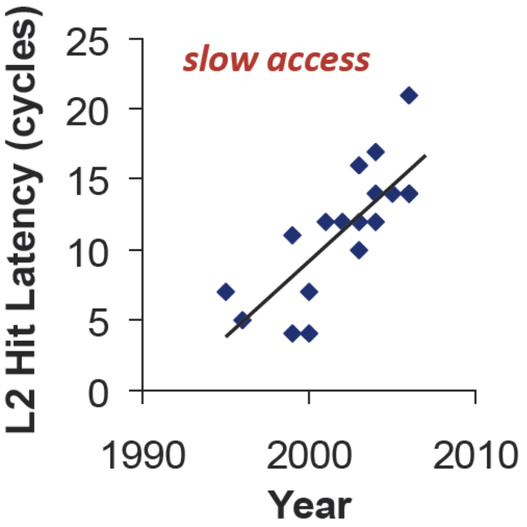
On-chip communication delay remains constant across technologies

- 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

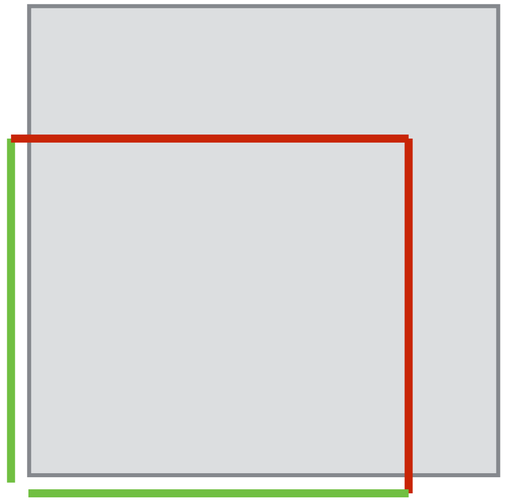
NUCA: non-uniform cache architecture by physically distributing cache slices across the die

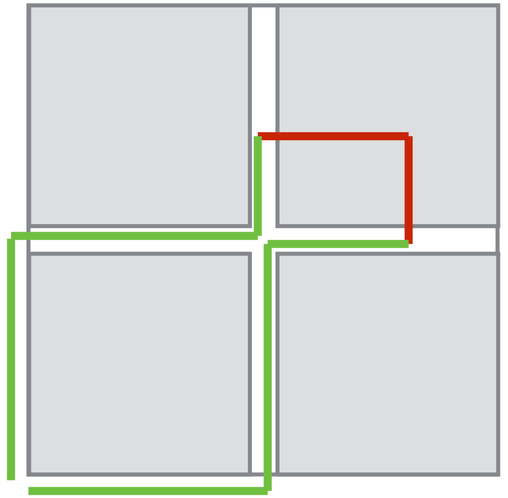
- Range of latencies, several times higher in worst case

Goal: move cache blocks closer to the core

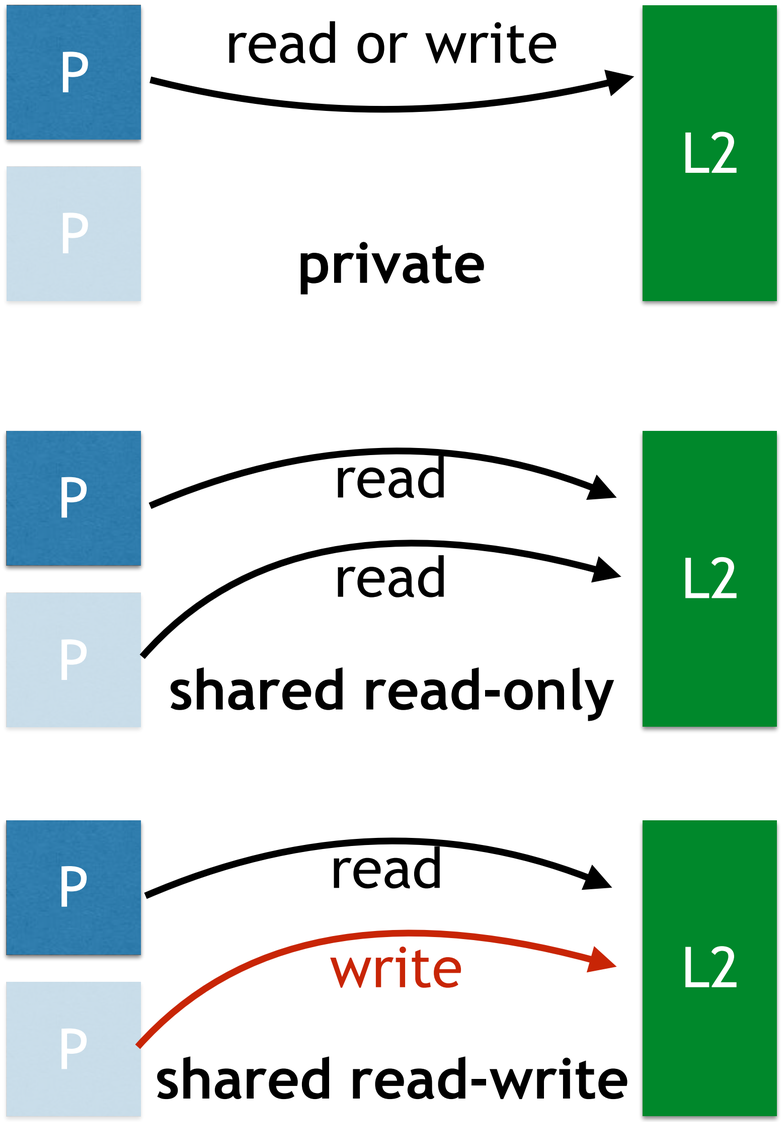
- Previously: cache block determines location
- Now: look-up mechanism required to find cache blocks

Main question: which cache block to put where?





BASIC IDEA: CLASSIFICATION OF ACCESSES



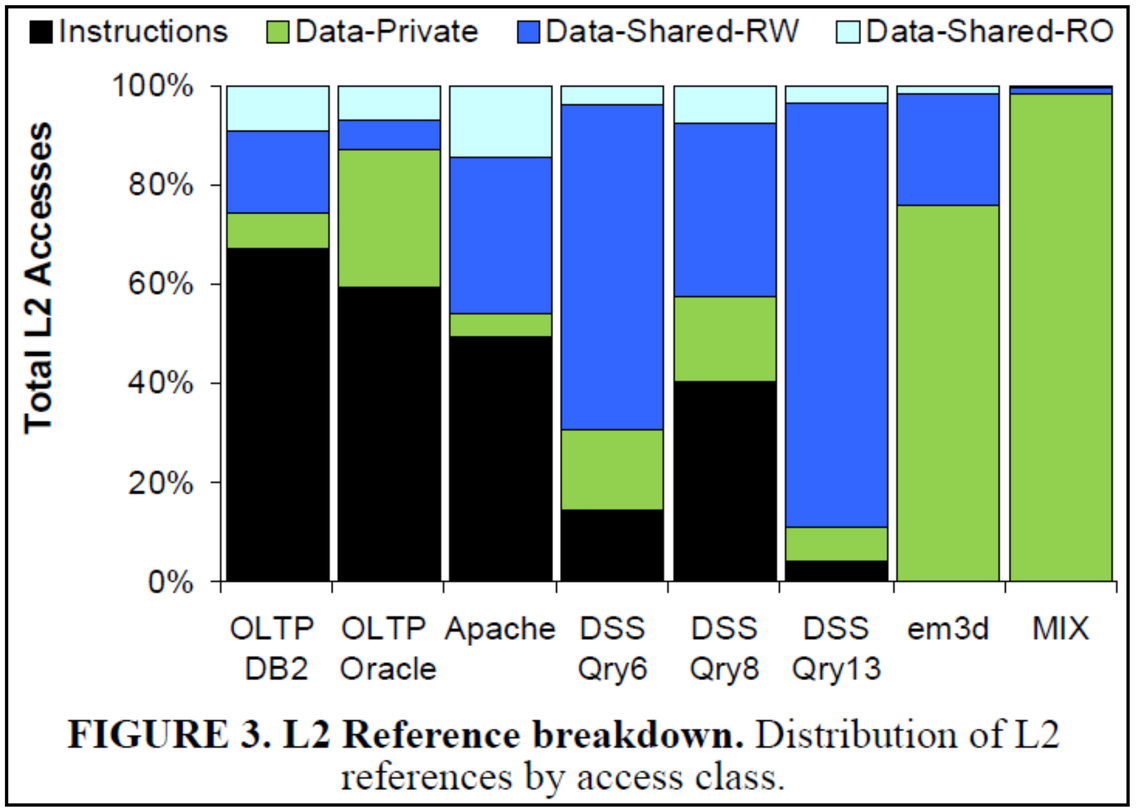
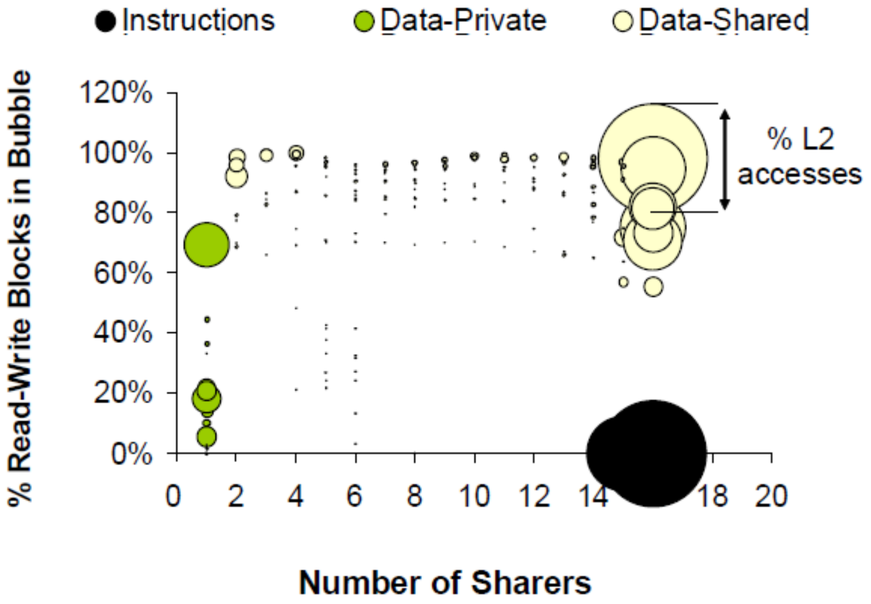


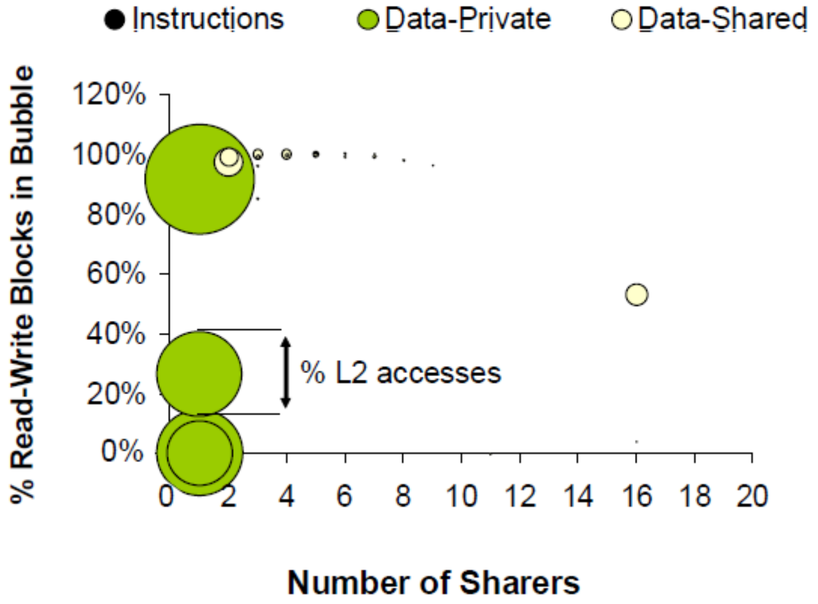
FIGURE 3. L2 Reference breakdown. Distribution of L2 references by access class.

Hardavellas et al., ISCA 2009

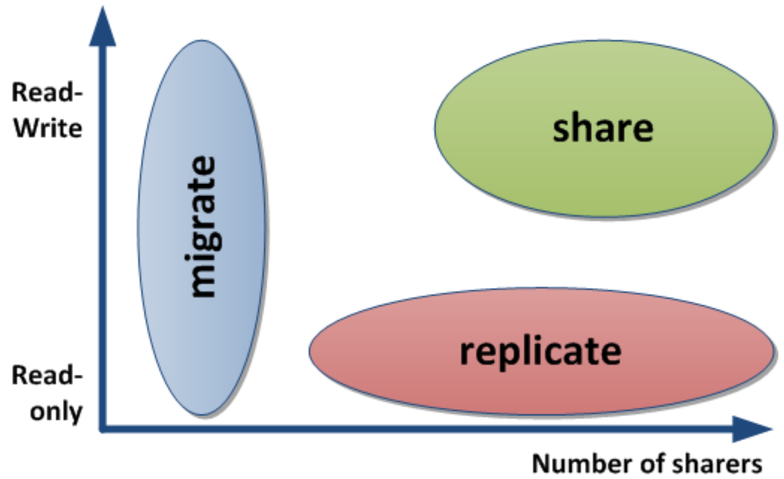
CLASSIFYING CACHE BLOCKS



(a) Server Workloads



(b) Scientific and Multi-programmed Workloads



Hardavellas et al., ISCA 2009

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)

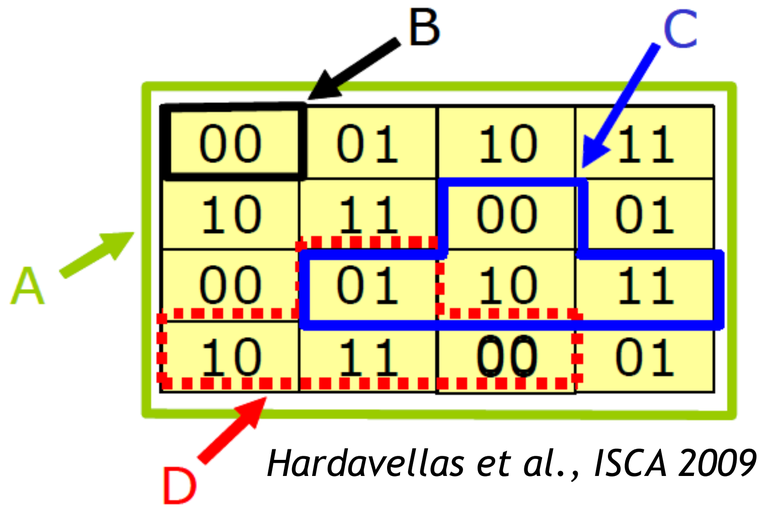
Reactive NUCA: placement using standard address interleaving, block's address determines location, unique locations (no replication)

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)