Corey
: An Operating System for Many Cores

Jhuyeong Jhin

2017. 6. 12

What is the problem?

- Chip manufacturers have adopted multicore architectures

→"Cache-coherent", "shared memory", "multiprocessor" is common in modern PCs

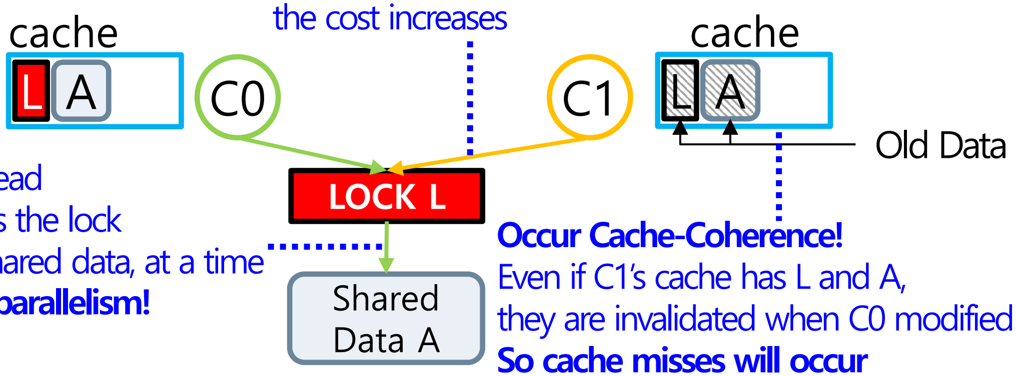
- However, the existing OS services are implemented without optimizing for the increasing number of cores = **lower scalability**

- Since many applications spend time in the kernel, Poorly scaling OS services can be huddle on the application performance

Introduction

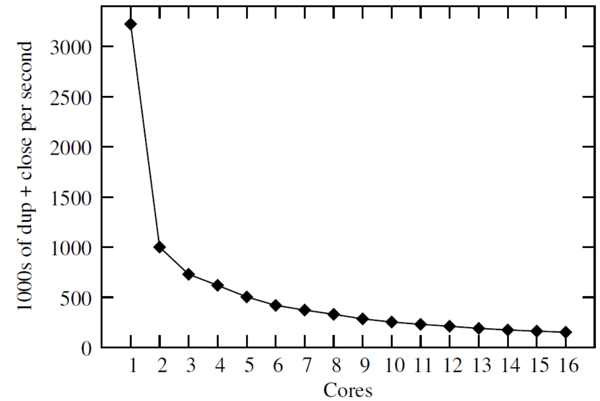
- One source of poor scalability
  : **data structure modified by multiple cores** (ex. File descriptor table)

C1 must wait for C0 to release the lock
→ **Locking cost**
As the # of waiting threads increases,
the cost increases



Only one thread
which aquires the lock
can access shared data, at a time
→ **interfere parallelism!**

Occur Cache-Coherence!
Even if C1's cache has L and A,
they are invalidated when C0 modified them first.
So cache misses will occur
when C1 accesses the L and A.

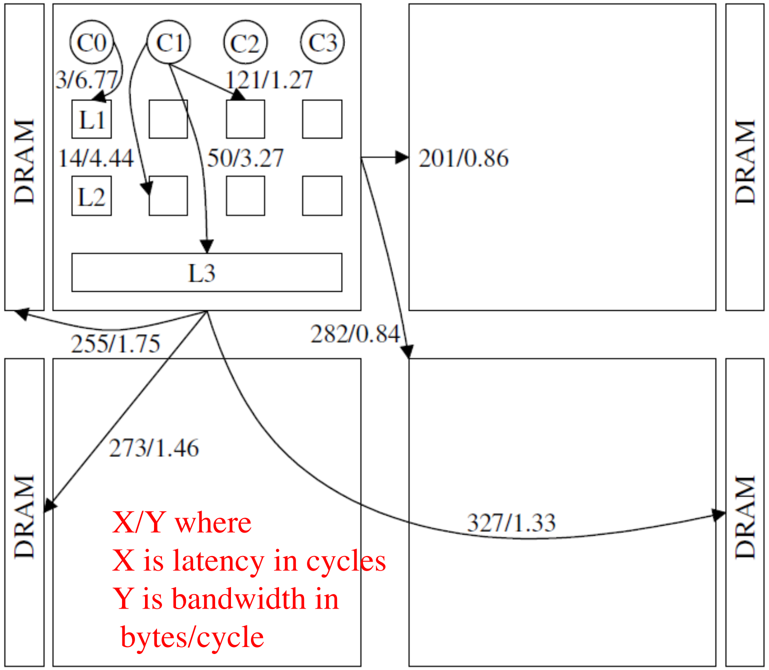


※ linux 2.6.25 version

- **Corey**
  - An operating system for many cores
  - Three new abstractions: *Address Range, Kernel Core, Share*
  - Let application control how the OS shares data btw cores
  - Implement Most higher services as **library OS** (organized like an exokernel)

Multicore Challenge - Hardware Obstacle

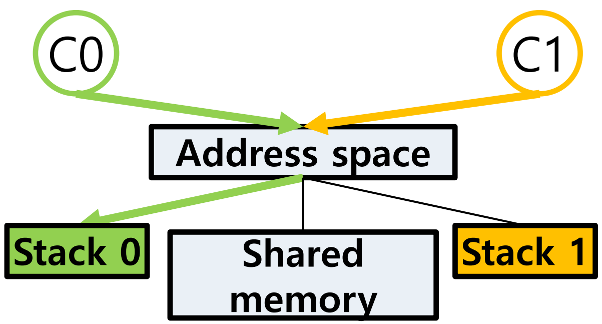
- **e.g. AMD 16-core machine**
  - 4 quad-core Opteron processors
    - L1 and L2 are private
    - L3 are shared by 4 cores on the same chip
  - Square interconnection
    - Access times are non-uniform
    - carry data btw cores and memory
    - Broadcast cache coherence
      to locate and invalidate cache lines
    - point-to-point cache coherence transfers
      of individual cache lines



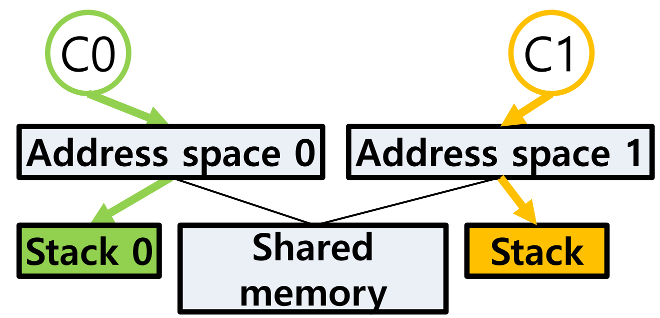
Design - PROBLEM1

- **Most OSs let applications choose btw shared and private**
  - Single **shared** address space for all cores
    - Implemented with multiple threads
  - One **Private** address space per core
    - Implemented with multiple processes with mmap(MAP_SHARED)

CASE 1: At the same time, C0, C1 access to its stack, respectively.



<SHARED>
Because they share a single address space,
C1 waits for C0 to release the lock for the address space.
→ **needless lock cost!**

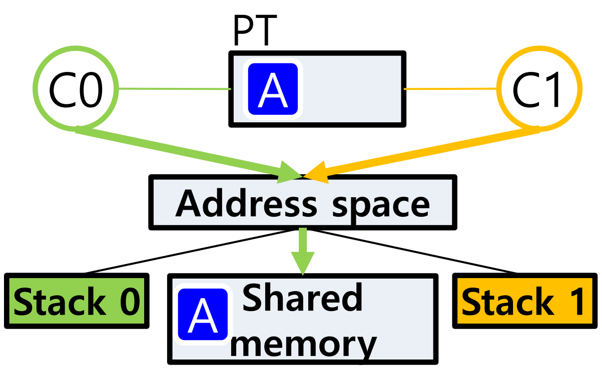


<PRIVATE>
Because they use separate address spaces,
C0, C1 don't need lock for their own address space.

Design - PROBLEM1 (cont.)

- **Most OSs let applications choose btw shared and private**
  - Single **shared** address space for all cores
    - Implemented with multiple threads
  - One **Private** address space per core
    - Implemented with multiple processes with mmap(MAP_SHARED)

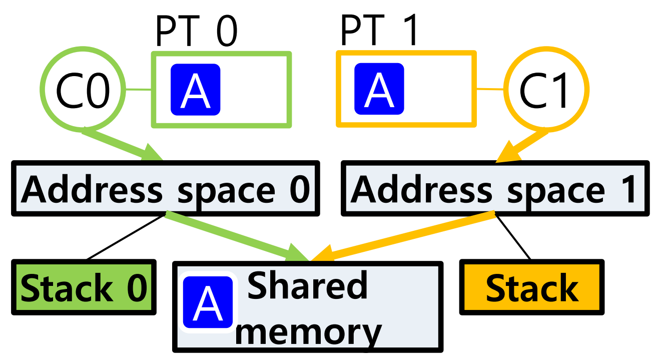
CASE 2: C0 and C1 read A(shared data) in order.



<SHARED>

just one soft page fault when C0 read A.

When C1 read A, the soft page fault does not occur because they share a single address space and page table.



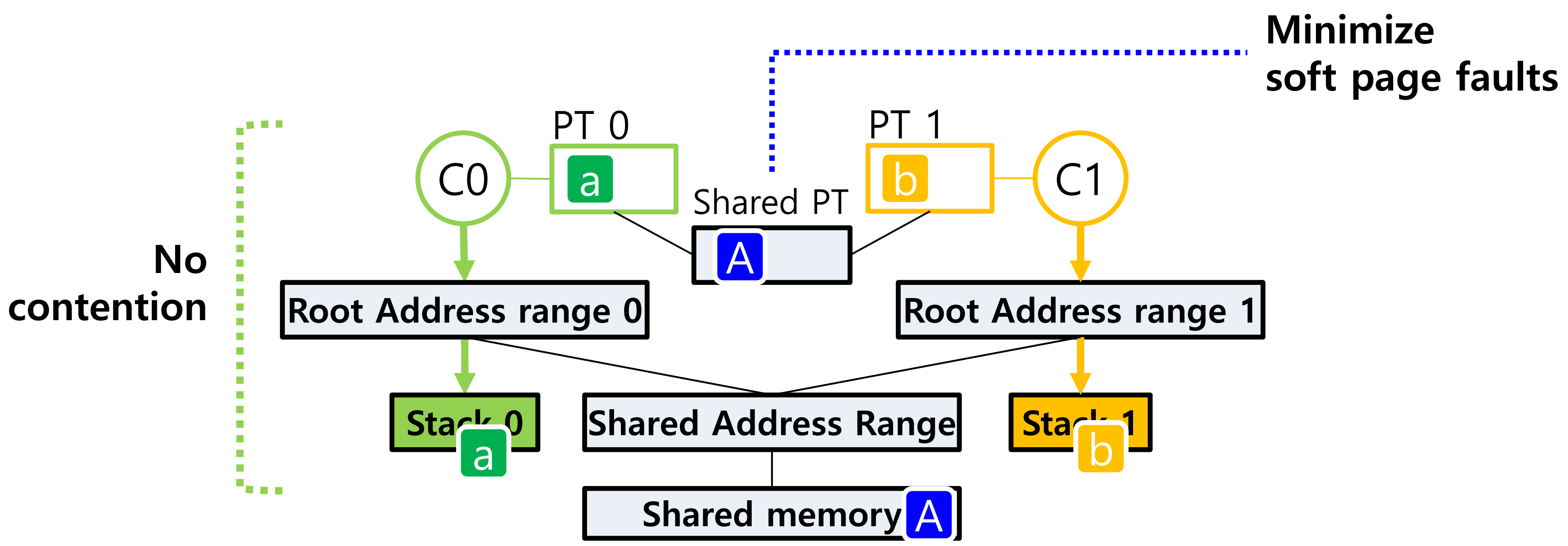
<PRIVATE>

soft page fault for both reads.

because they use separate address spaces and page tables.

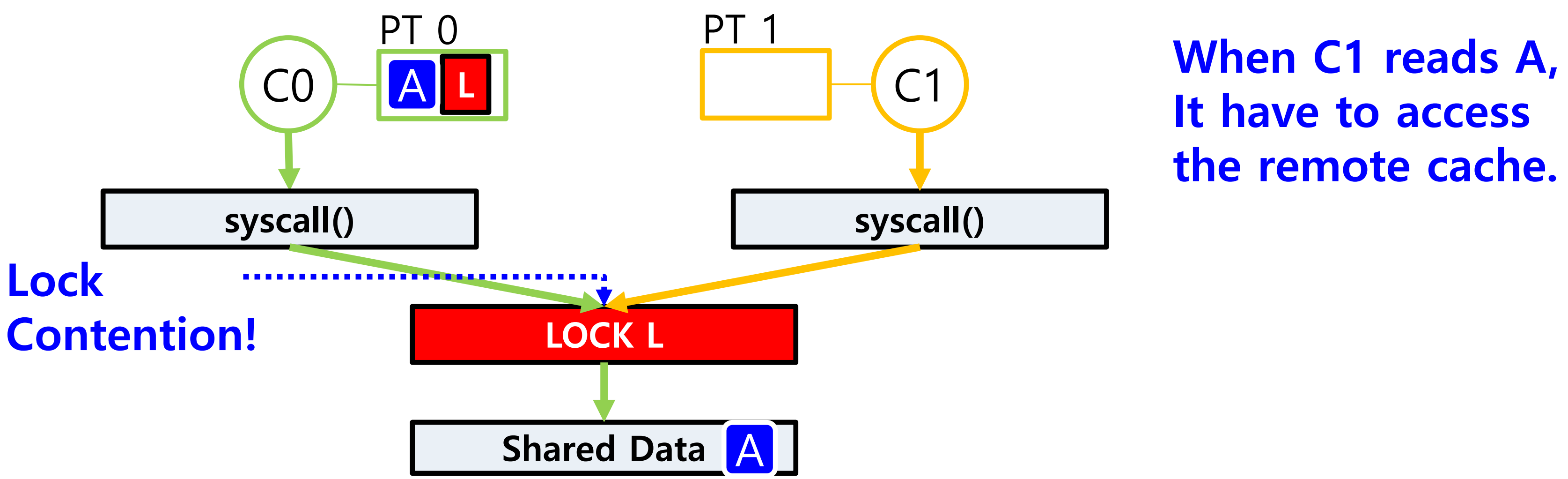
Design - SOLUTION1: Address Range

- **Neither of the previous choices is fully satisfactory**
- **SOLUTION: Address range**
  - kernel-provided abstraction
  - **allow applications decide whether shared/private**
  - Avoid contention for private memory
  - Share PTEs for shared memory
    → Minimize soft page faults
    : when page is in main memory, but not mapped in the process page table



Design - PROBLEM2

- In general, syscall is executed on the invoked core
- When two applications invokes the same syscall, if the syscall needs to access large shared kernel data?
  - One waits for another to release the shared data
  - And also, fetches relevant cache lines from the last core to use the data

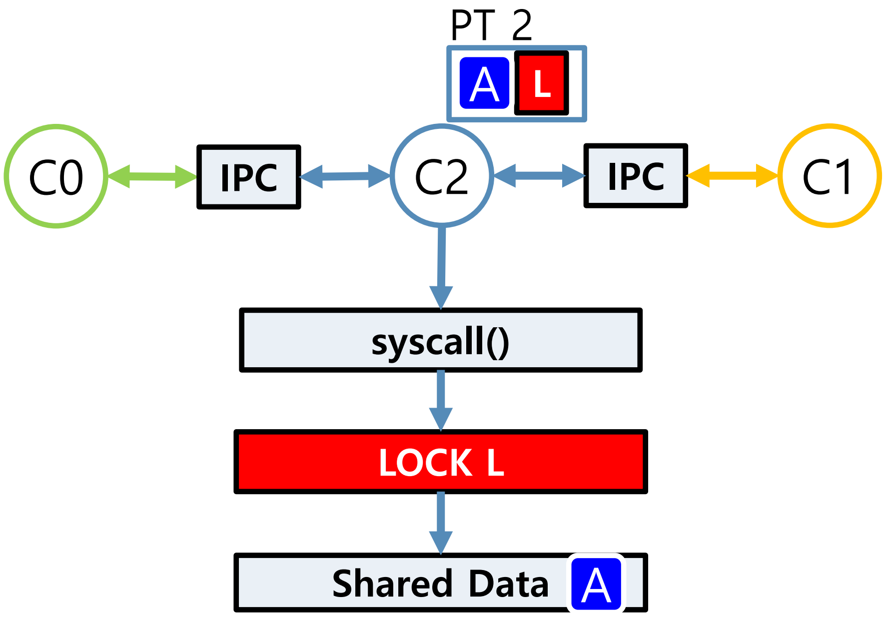


Design - SOLUTION2: Kernel Core

- **Kernel core**
  - **let applications dedicate cores to run specific kernel functions**
  - avoid inter-core contention over the data these functions access

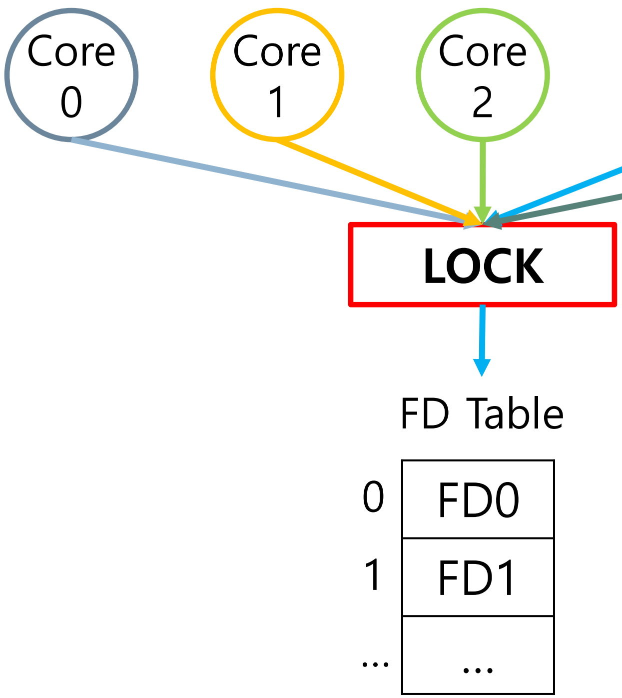
  <CONS> reduce the number of cores available to the function

  <PROS> improve overall performance by reducing access to remote cache



Design - PROBLEM3

- Many kernel operations involve looking up identifiers in table to obtain a pointer to kernel data structure
  - file descriptor entry, process ID, ...
- **If particular identifier used by an application needs only limited scope, the larger scope of lookup results may in the needless contention**
  - example



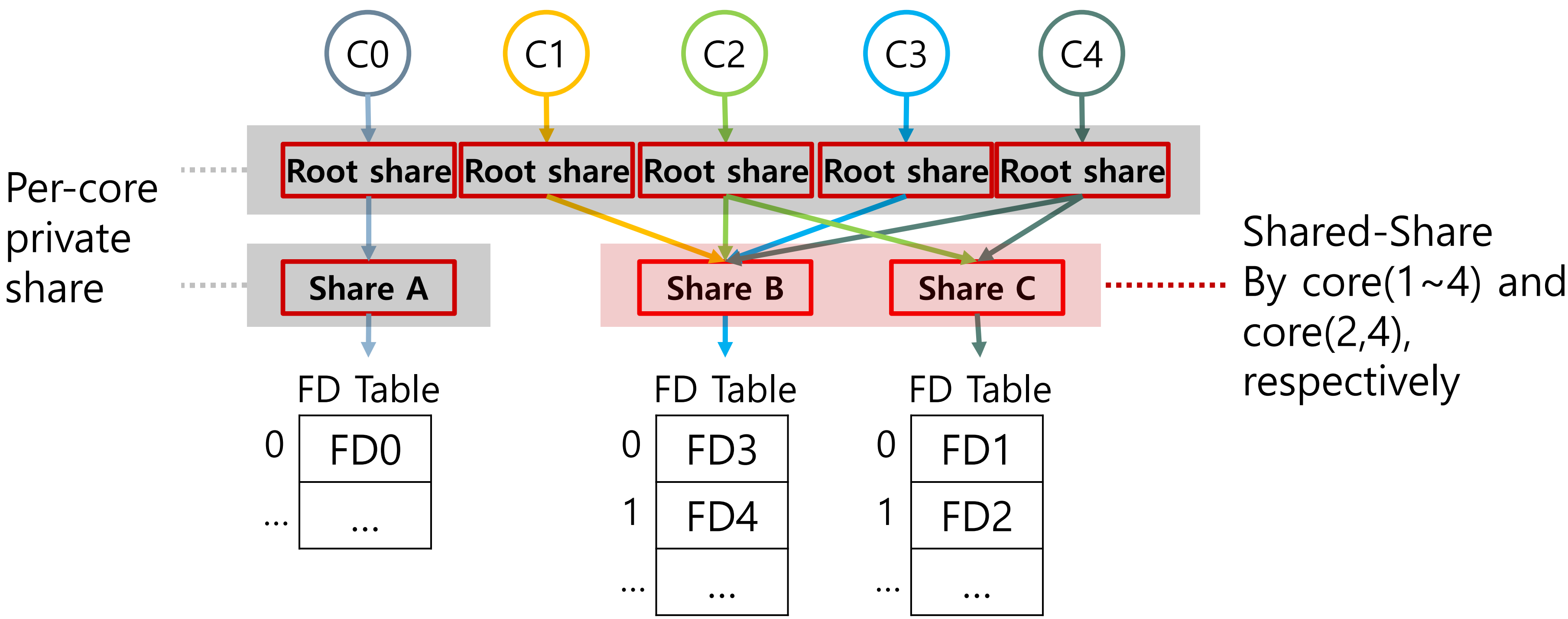
Assume that

- Core 0~4 is executing the threads of a process, respectively.
- The cores acquires the lock of FD Table to access a FD
- Core 0 want to access the FD0
- But currently, Core 3 acquires the lock

Q. If FD0 is only used by a specific function(Core 0)?
A. the needless contention occurs!

Design - SOLUTION3: Share

- **Share**
  - lets application to specify **which object identifiers are visible to other cores**
  - Each thread has a **root share** that is private to the core executing the thread
    - need no lock (private)
  - If two cores want to "share a share",
    they create share ID and add that to private root shares
    (or share reachable from these root shares)



Corey Kernel

- Applications implement domain specific optimization using the corey's system call
  - Overview of corey system call

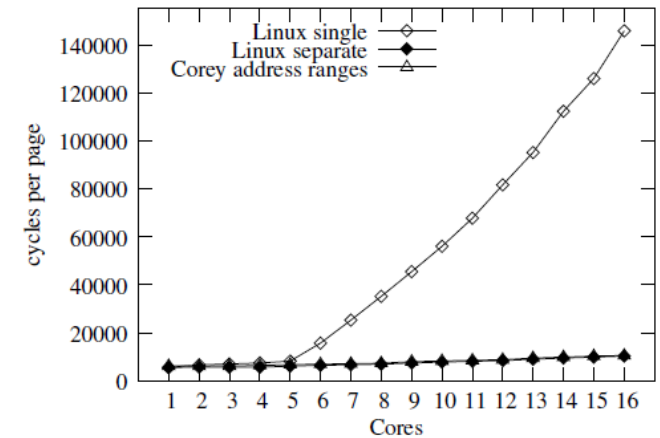
System call	Description
name obj_get_name(obj)	return the name of an object
shareid share_alloc(shareid, name, memid)	allocate a share object
void share_addobj(shareid, obj)	add a reference to a shared object to the specified share
void share_delobj(obj)	remove an object from a share, decrementing its reference count
void self_drop(shareid)	drop current core's reference to a share
segid segment_alloc(shareid, name, memid)	allocate physical memory and return a segment object for it
segid segment_copy(shareid, seg, name, mode)	copy a segment, optionally with copy-on-write or -read
nbytes segment_get_nbytes(seg)	get the size of a segment
void segment_set_nbytes(seg, nbytes)	set the size of a segment
arid ar_alloc(shareid, name, memid)	allocate an address range object
void ar_set_seg(ar, voff, segid, soff, len)	map addresses at voff in ar to a segment's physical pages
void ar_set_ar(ar, voff, ar1, aoff, len)	map addresses at voff in ar to address range ar1

- The corey's system calls allocate and manage the five types of low-level objects : share, segment, address range, pcore, device

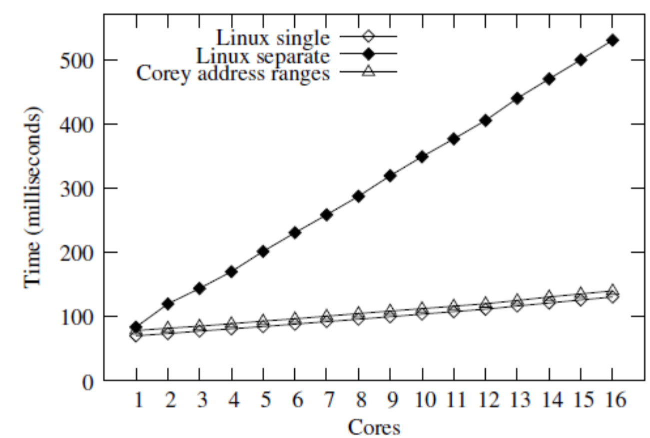
Evaluation

- **Linux (comparing): Debian Linux with kernel 2.6.25**

- **Two micro benchmarks**
  - Memclone
    - each core allocate its own 100MB array and modify each page of the array
  - Mempass
    - allocates a single 100MB shared buffer on one of the cores
    - it touches all page of the buffer, and lets the next core do the same
    - repeat these until every core touches every page.



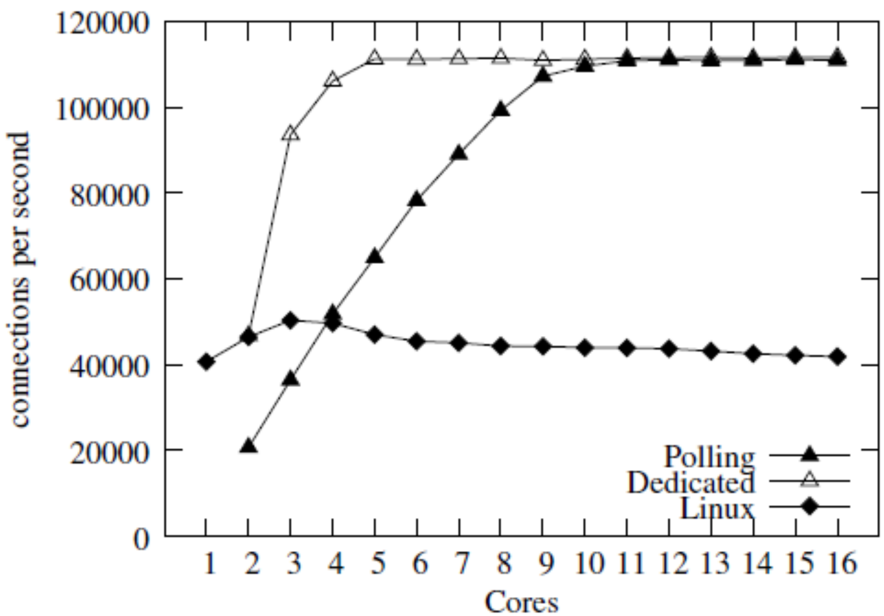
(a) memclone



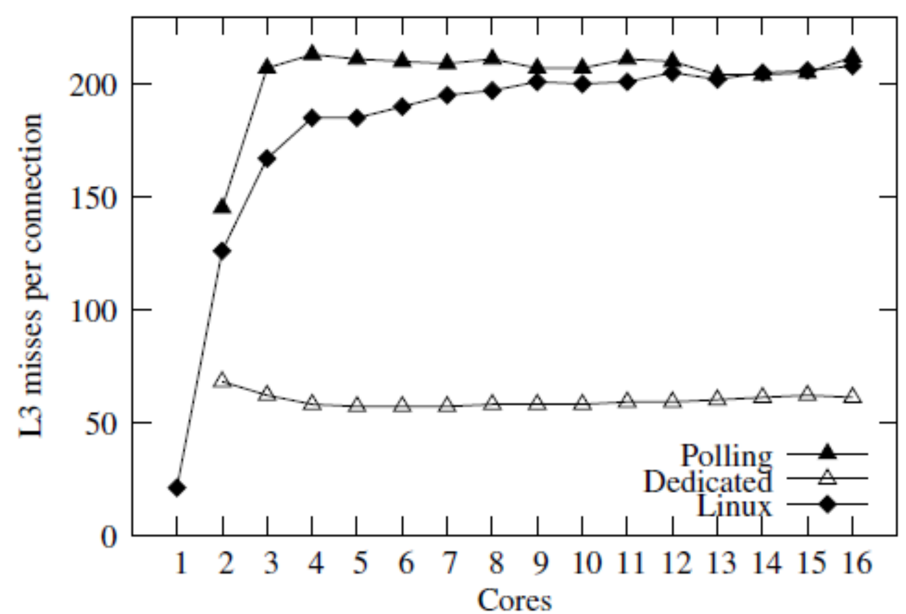
(b) mempass

Evaluation (cont.)

- **Simple TCP service**
  - accepts incoming connection requests
  - writes 128 bytes to the connection then close
- **Two configurations**
  - "Dedicated" uses a kernel core for all network processing
  - "Polling" uses a kernel core only to poll for packet notifications and transmit completions



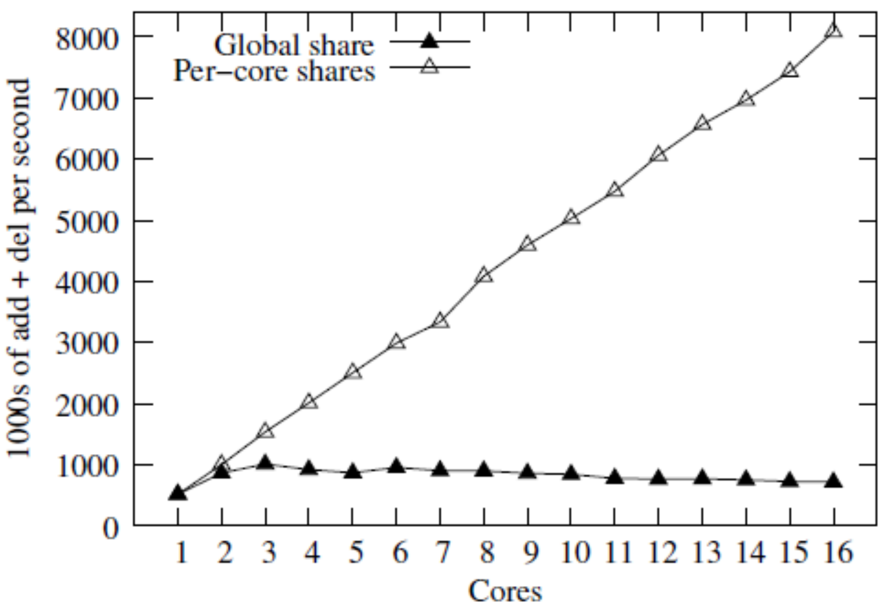
(a) Throughput.



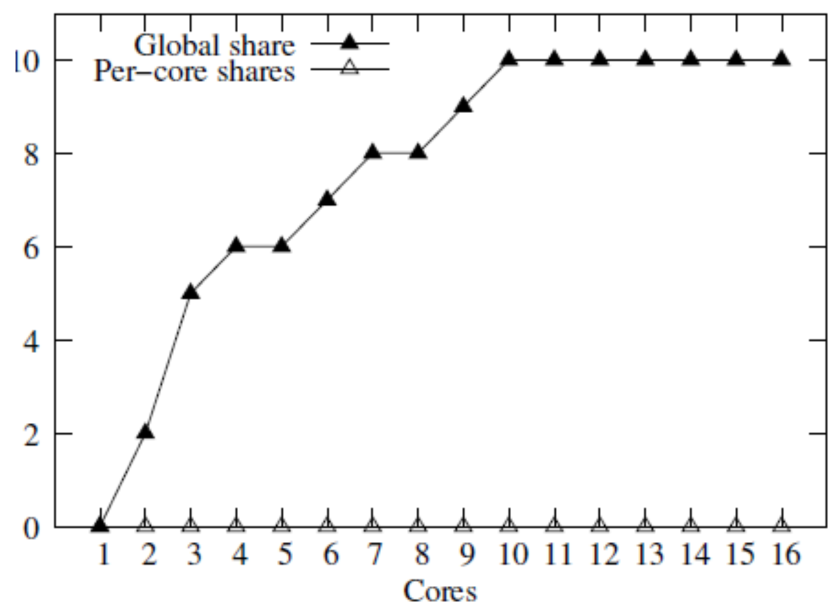
(b) L3 cache misses.

Evaluation (cont.)

- **Two micro benchmarks**
  - Global Share:
    each core calls share_addobj() to add a per core segment to a global share then calls share_delobj() to delete that
  - Per-core Shares:
    same but per-core segment is added to a local share



(a) Throughput.



(b) L3 cache misses.

16

THANK YOU