Scalability to Many Cores

CS 202: Advanced Operating Systems

… specifically in Linux

An Analysis of Linux Scalability to Many Cores

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

OSDI 2010 (1 year after Barrelfish)

Key idea & methodology

- "Big idea" paper but through benchmarking & analysis
- Motivation
  - Research at that time argued that traditional OS structure design would not scale to multi-/many-core (e.g., the multikernel paper)
- Idea
  - Trying to answer whether traditional OS designs can be "patched" to allow applications to scale
- Approach
  - Benchmark Linux using a set of applications that should scale well
  - Identify and mitigate scalability bottleneck

Motivation

- Multicore architectures
- Do we need new kernel designs?
  - Barrelfish, Corey, fox, …
  - “Existing kernels don’t scale well”
- Can we fix traditional kernel architectures?
  - Can we achieve 100x speed up with 100 cores?
  - How hard is it to fix them?

Paper highlights

- Asks whether traditional kernel designs apply to multicore architectures
  - Hard to answer in general, but the paper sheds some light on the answer by analyzing Linux scalability
- Investigated 7 real-world applications
  - Running on a 48-core computer
  - Analysis of bottlenecks
- Concluded that most kernel bottlenecks could be eliminated using standard parallelizing techniques

Why Linux? Why these applications?

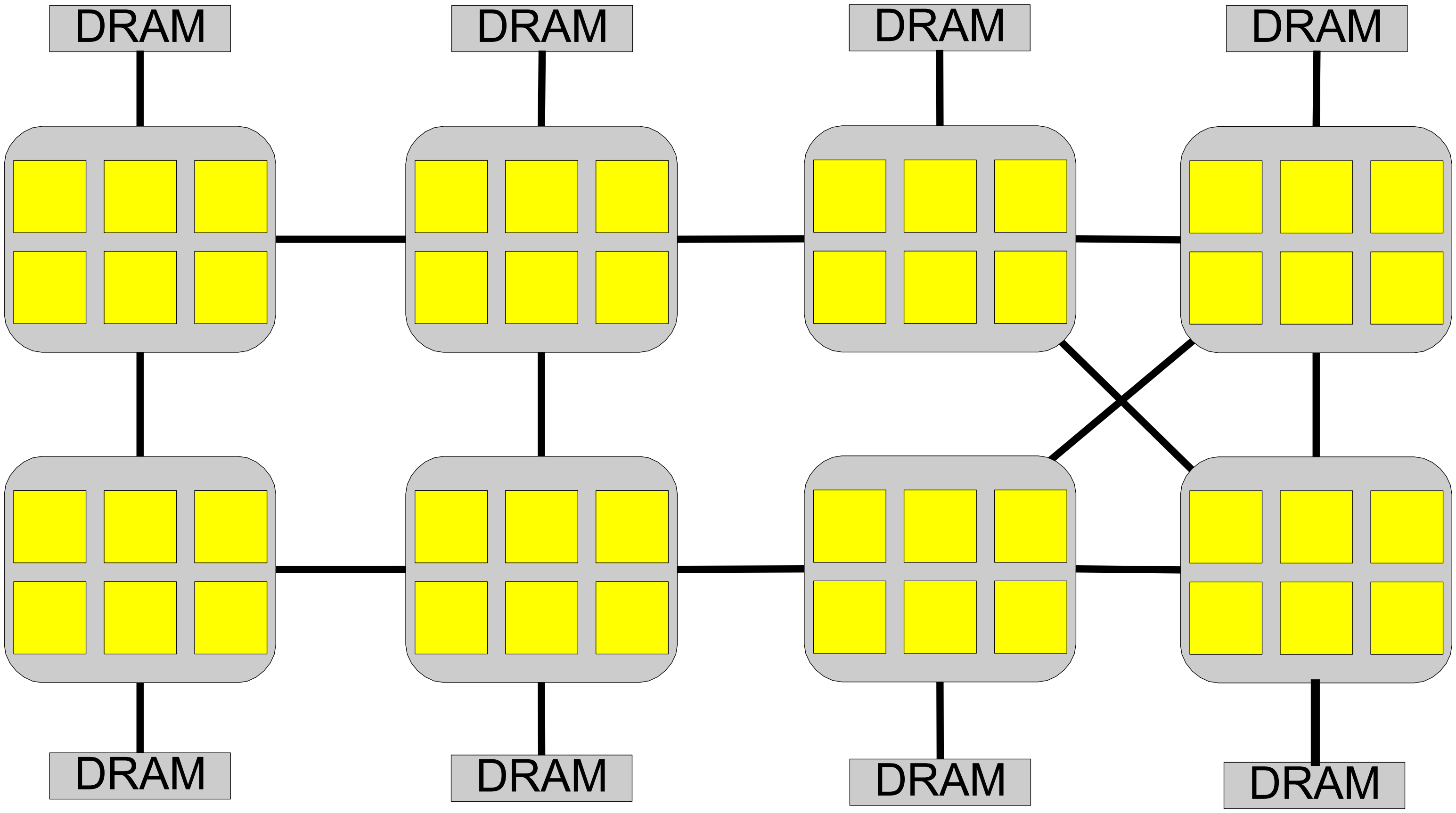
- Linux has a traditional kernel design
- Linux community has made a great progress in making it scalable
- The chosen applications are designed for parallel execution and stress many major Linux kernel components

How to decide if Linux is scalable?

- Measure scalability of the applications on a recent Linux kernel
- Understand and fix scalability problems
- Kernel design is scalable if the changes are modest
- Kinds of problems
  - Linux kernel implementation
  - Applications' user-level design
  - Applications' use of Linux kernel services

Off-the-shelf 48-core server

- 6 core x 8 chip AMD



MOSBENCH Applications

- 2 types of applications
  - Previous work has shown not to scale well on Linux
    - Memcached, Apache and Metis (MapReduce library)
  - Designed for parallel execution and kernel intensive
    - gmake, PosgtreSQL, Exim and Psearchy
- Synthetic user workloads to cause these apps to use the kernel intensively
  - Stress the network stack, file name cache, page cache, memory manager, process manager and scheduler
- <u>Sign of bad scalability</u>: spending more time in the kernel as the # of cores increases

Exim

- Exim is a mail server
- Single master process listens for incoming SMTP connections via TCP
  - The master forks a new process for each connection
  - Has a good deal of parallelism
  - Spends 69% of its time in the kernel on a single core
- Stresses process creation and small file creation and deletion

Memcached- object cache

- In-memory key-value store used to improve web application performance
- Has key-value hash table protected by internal lock
- Stresses the network stack, spending 80% of its time processing packets in the kernel at one core

Apache - web server

- Popular web server
- Single instance listening on port 80
- One process per core – each process has a thread pool to service connections
- On a single core, a process spends 60% of the time in the kernel
- Stresses network stack and the file system

PostgreSQL

- Popular open source SQL database
- Makes extensive internal use of shared data structures and synchronization
- Stores database tables as regular files accessed concurrently by all processes
- For read-only workload, it spends 1.5% of the time in the kernel with one core, and 82% with 48 cores

gmake

- Implementation of the standard make utility that supports executing independent build rules concurrently
- Unofficial default benchmark in the Linux community
- Creates more processes than cores, and reads and writes many files
- Spends 7.6% of the time in the kernel with one core

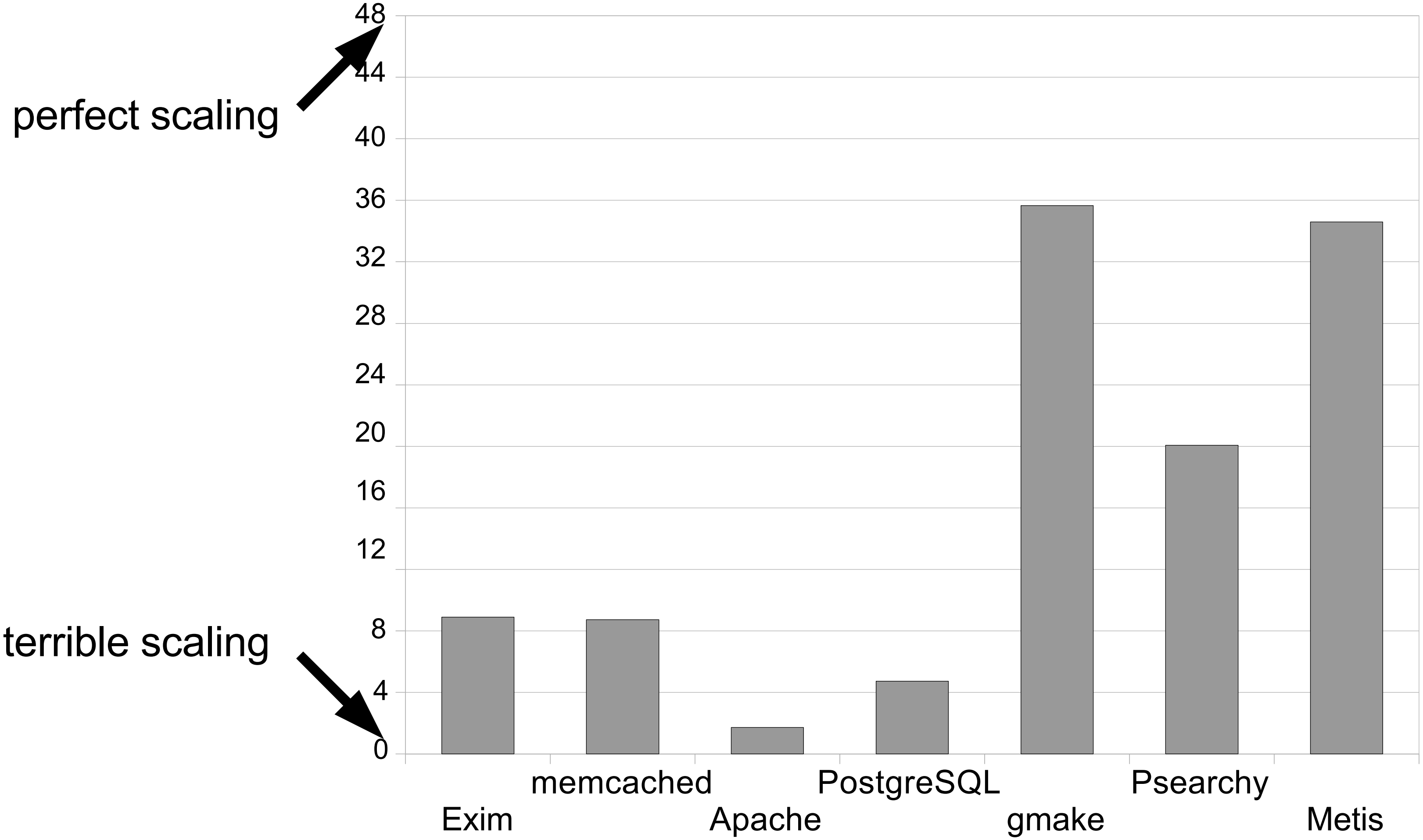
Pserchy - file indexer

- Parallel version of searchy, a program to index and query web pages
- Version in the article runs searchy indexer on each core, sharing a work queue of input files

Metis - MapReduce

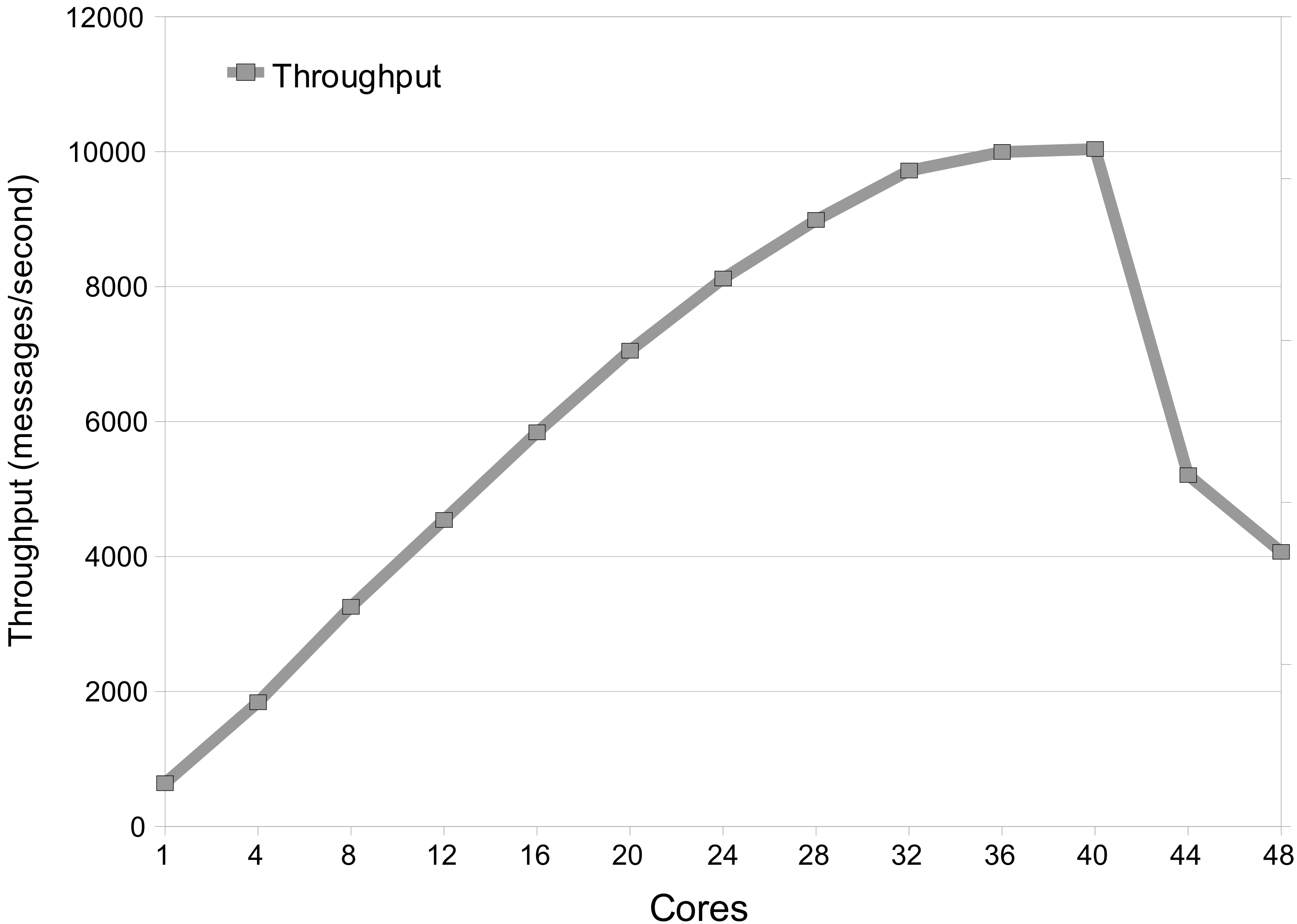
- MapReduce library for single multicore servers
- Allocates large amount of memory to hold temporary tables, stressing the kernel memory allocator
- Spends 3% of the time in the kernel with one core, 16% of the time with 48 cores

Poor scaling on stock Linux kernel

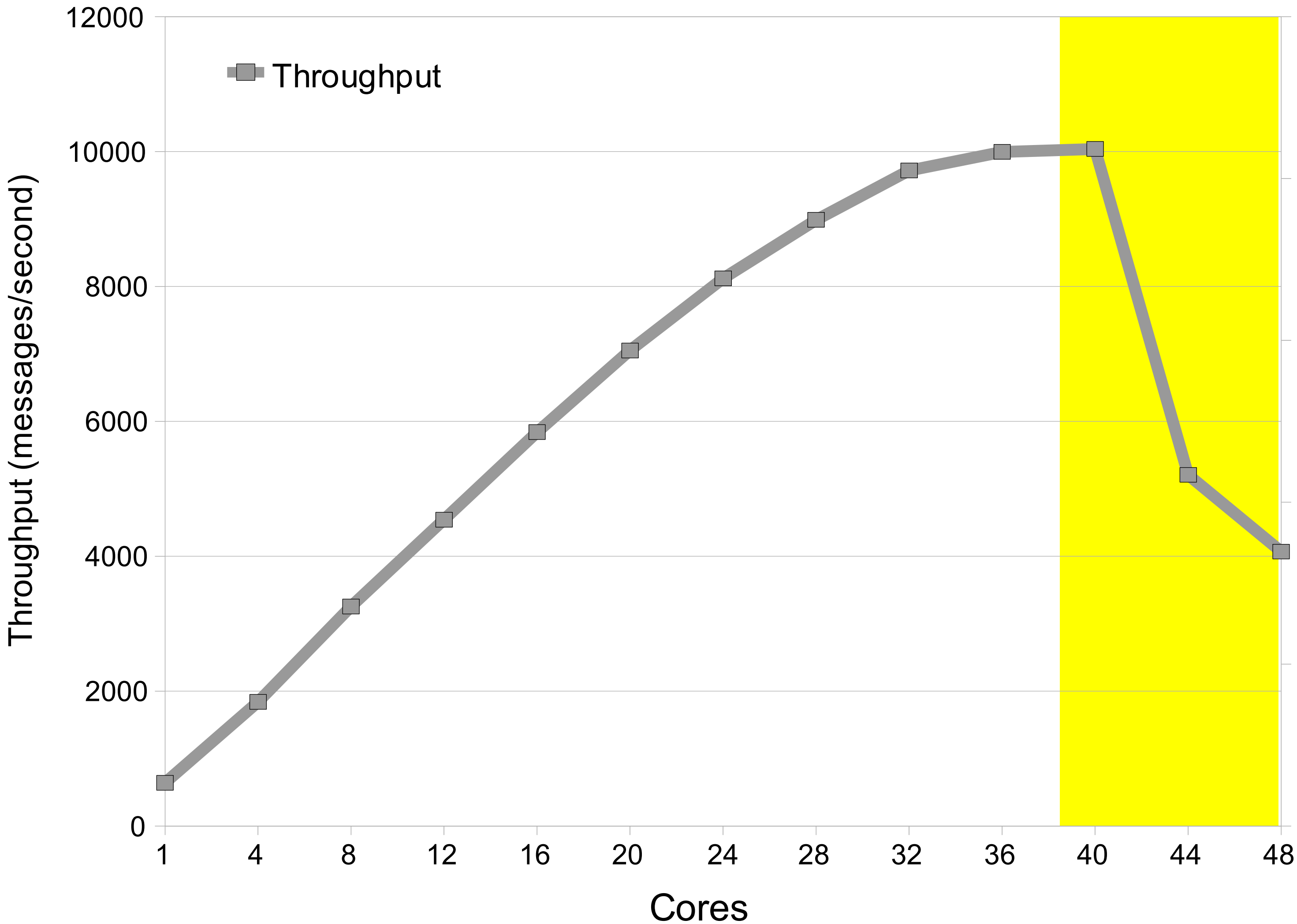


Y-axis: (throughput with 48 cores) / (throughput with one core)

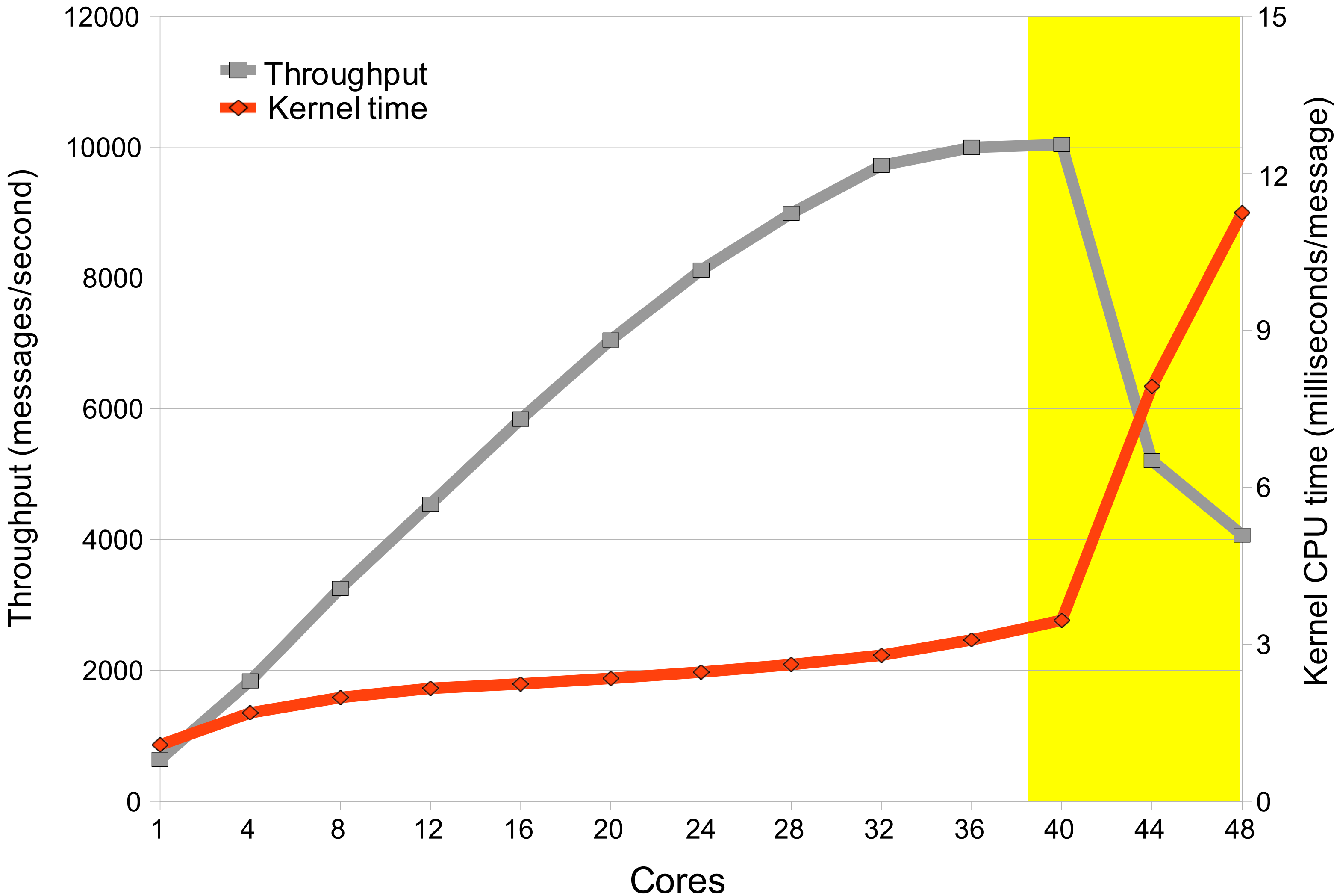
Exim on stock Linux: collapse



Exim on stock Linux: collapse



Exim on stock Linux: collapse



Common scalability issues

- Tasks may lock a shared data structure
  - Increasing core # increases the lock wait time
- Tasks may write a shared memory location
  - Cache coherence issues even in lock-free shared data structures
- Tasks may compete for shared hardware resources
  - Shared cache with limited size: cache miss rate
  - Inter-core interconnect, DRAM: memory stalls
- Too few tasks to keep all cores busy

Easy and hard fixes

- Some scalability issues might be hard to fix
- But still, many problems are easily fixable using well-known techniques such as
  - Lock-free protocols
  - Fine-grained locking
  - Per-core data structure
    - e.g., Linux's per-core runqueue

Multicore packet processing

- Packet processing
  - When a packet is received, it passes through multiple queues until it reaches application's socket queue
  - The performance would be better if each packet, queue and connection be handled by just one core
    - Avoid cache misses and queue locking
- Intel's 82599 10Gbit Ethernet (IXGBE) card
  - Multiple hardware queues
- Configure Linux to assign each HW queue to a different core
  - Transmitting: place packets on the HW queue for the current core
  - Receiving: enqueue incoming packets matching a particular criteria (source ip and port) on a specific queue

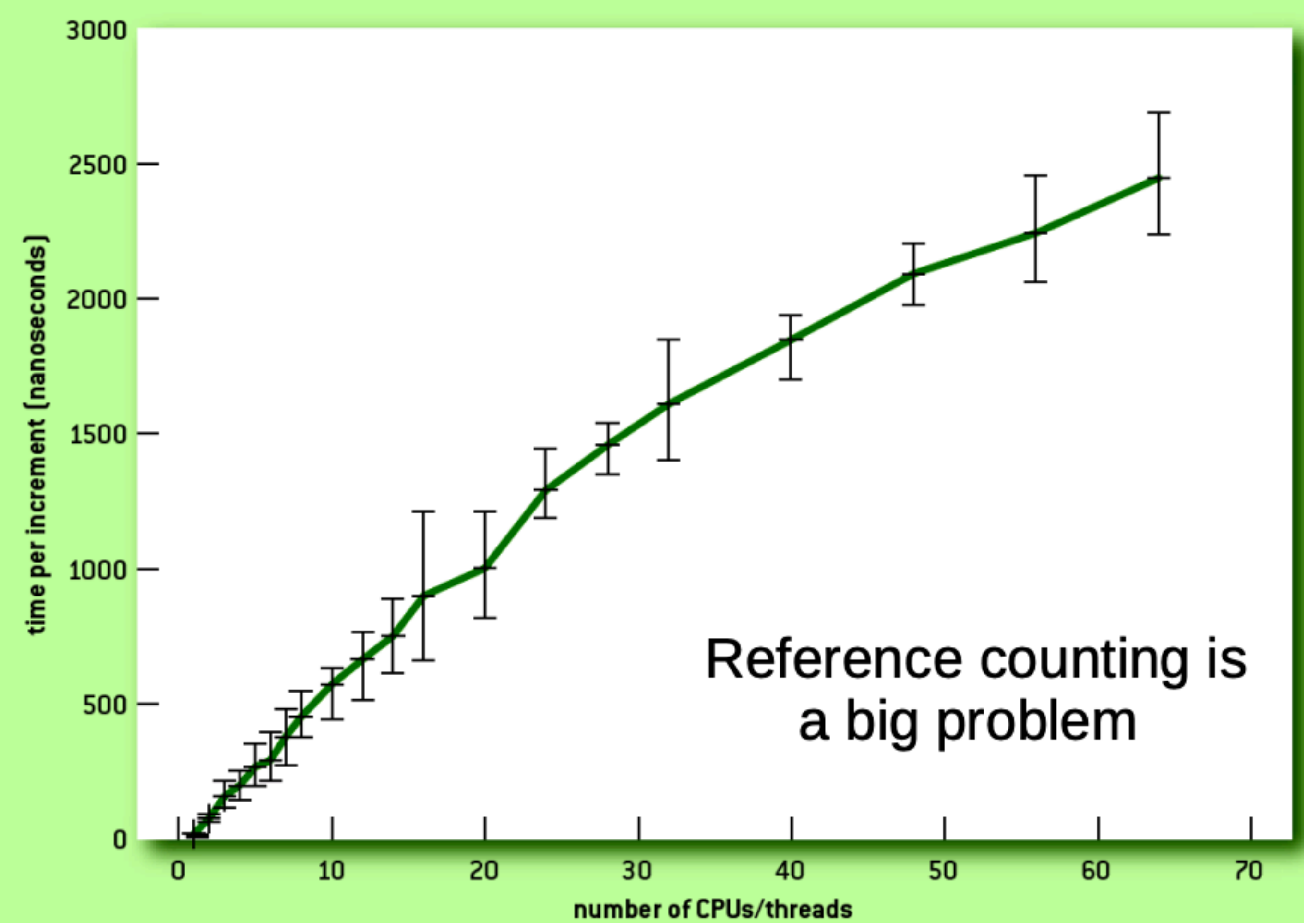
Problem: Reference counting

- Ref count indicates if kernel can free object
  - File name cache (dentry), physical pages, ...
  - Think of garbage collection
- Becomes bottleneck if many cores update them
- Atomic inc and dec do not help because of cache coherence

```
void dput(struct dentry *dentry)
{
    if (!atomic_dec_and_test(&dentry->ref))
        return;
    dentry_free(dentry);
}
```

A single atomic instruction limits scalability?!

Atomic increment on 64 cores



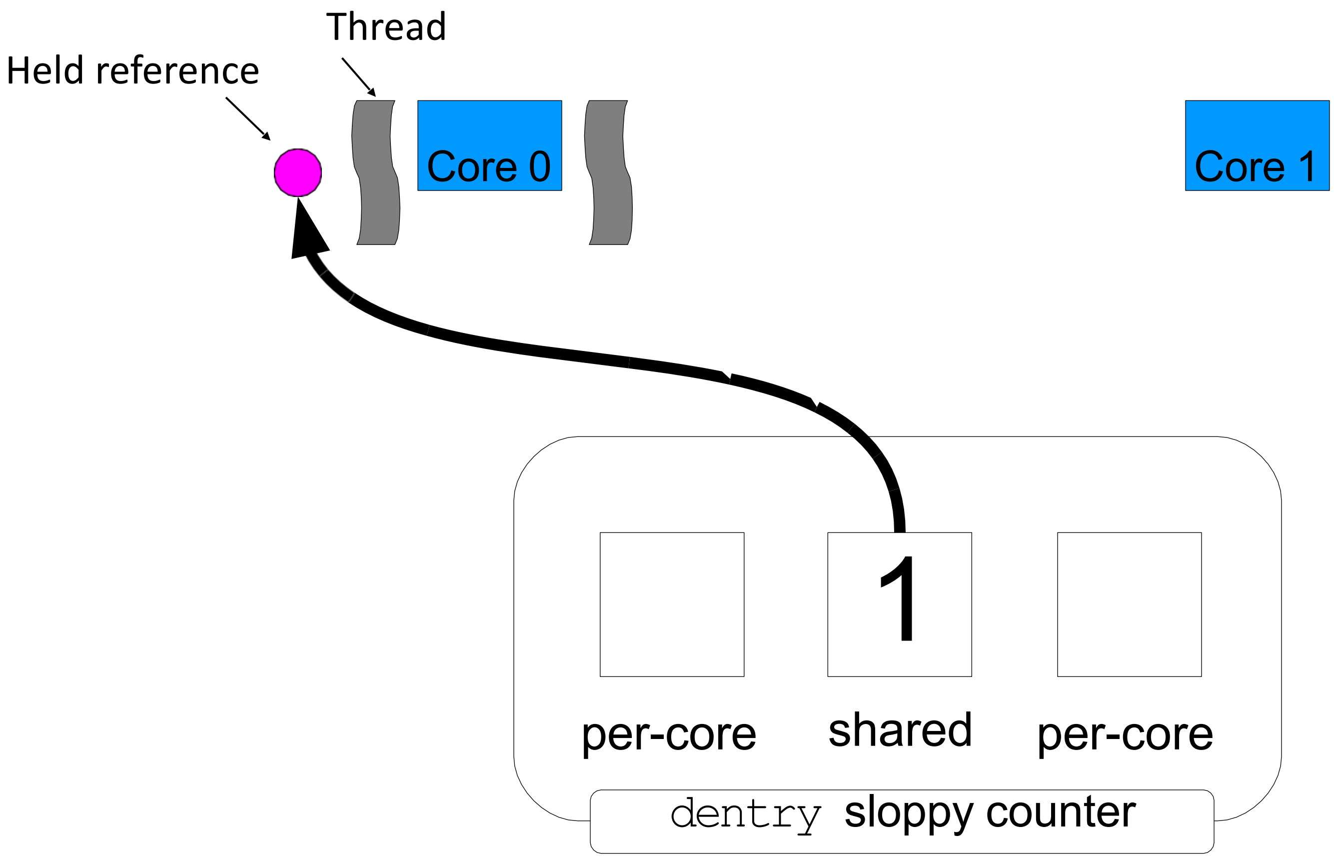
Sloppy counter

- Observation: Kernel rarely needs true value of ref counter
- Solution: Each core holds a few "spare" counters to an object
  - Gives ownership of local counters to threads on that core, without having to modify the global reference counter
- Global shared reference counter + core-local counters

  Spare references to an object
  - Local counter
    - Acts as a local reserve
    - Keep track of the number of spare references held by each core
  - Global counter
    - Keeps track of total # of references issued (local reserves + being used)

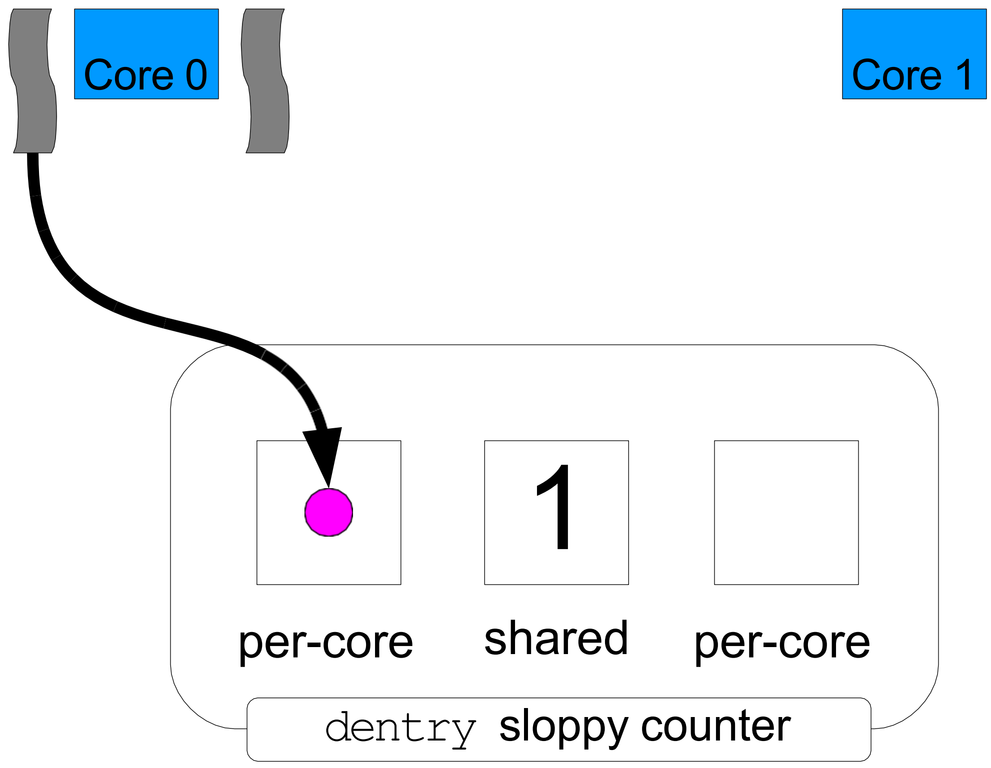
Sloppy counters

- Each core holds a few "spare" counters to an object



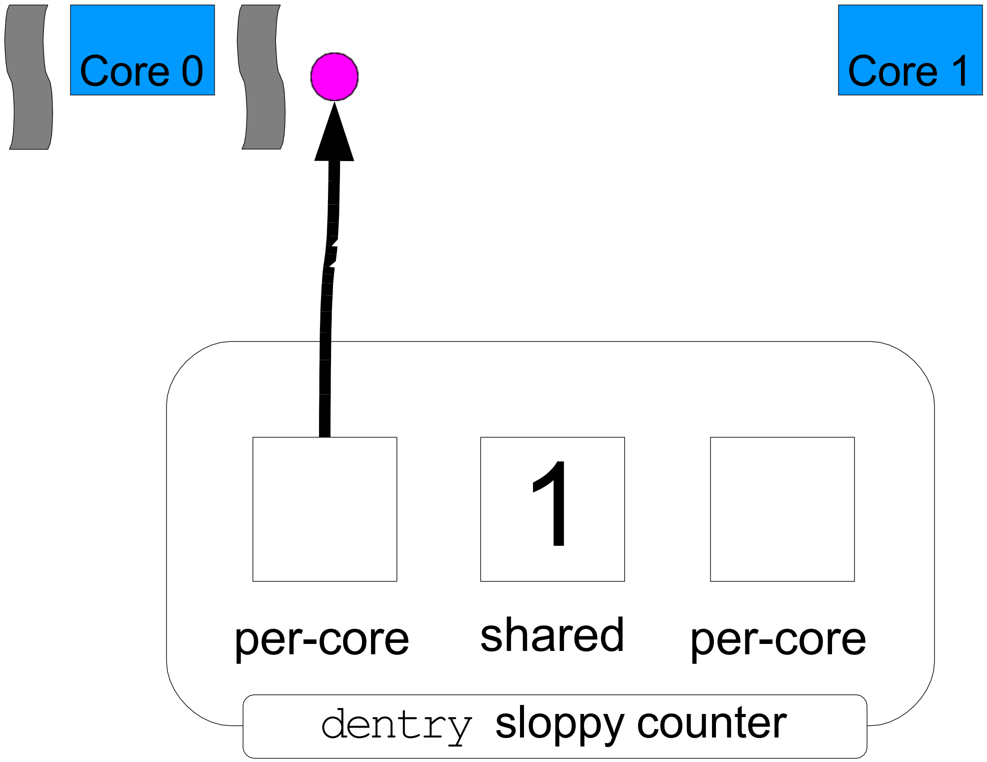
Sloppy counters

- Each core holds a few “spare” counters to an object



Sloppy counters

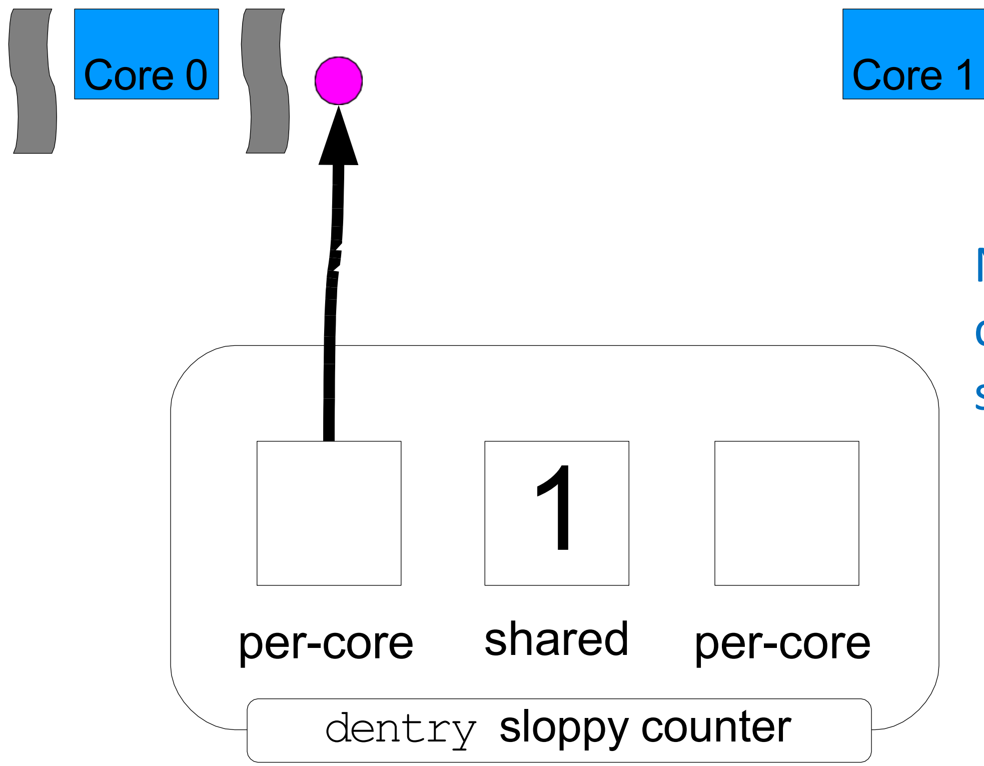
- Each core holds a few “spare” counters to an object



Another thread on core 0 can get the spare local (per-core) reference without updating the central shared counter

Sloppy counters

- Each core holds a few "spare" counters to an object



No waiting for locks or cache-coherence serialization

Another thread on core 0 can get the spare local (per-core) reference without updating the central shared counter

Sloppy counter algorithm (intuitively)

- Core increments the sloppy counter by V:
  - If local count is at least V: get V references and decrement local counter by V
  - Otherwise, the core must acquire the references from the global counter, so it increments the global counter by V
- Core decrements the sloppy counter by V:
  - Release V references for local use and increment the local counter by V
  - Sync.: If any $local\ count \geq threshold$ (i.e., sloppiness), adding the value of the local counter to the shared counter and resetting the local counter to zero
  - Sync. overhead v.s. counter accuracy
- Invariant:
  - Sum of local counters + number of used resources = shared counter

Sloppy counter example

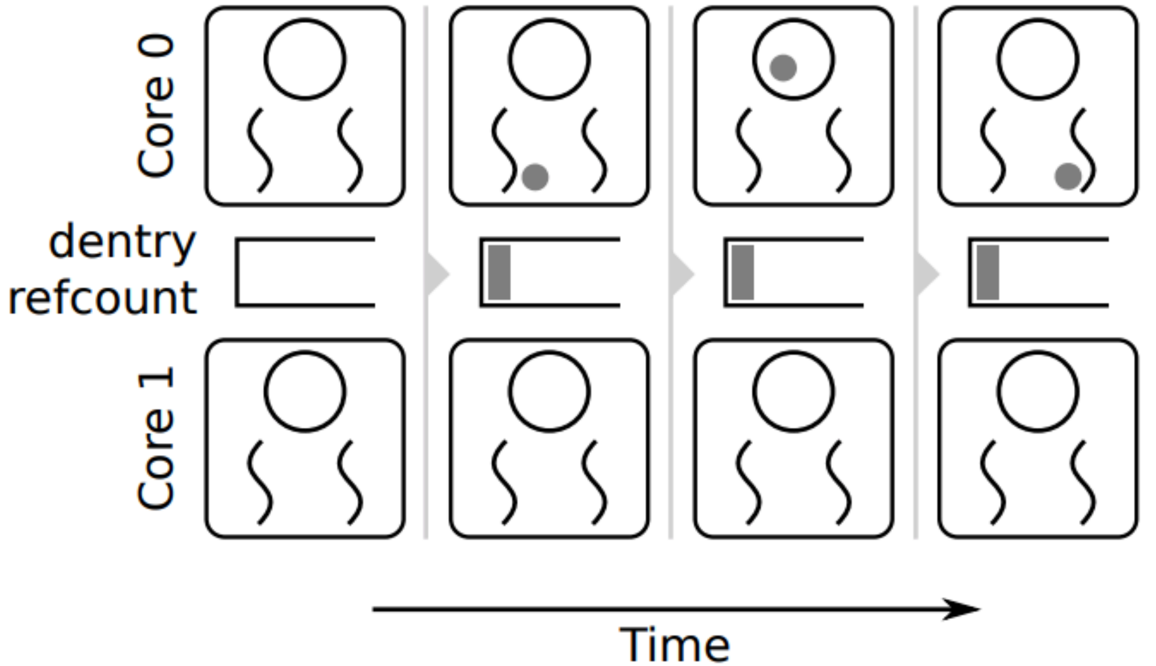
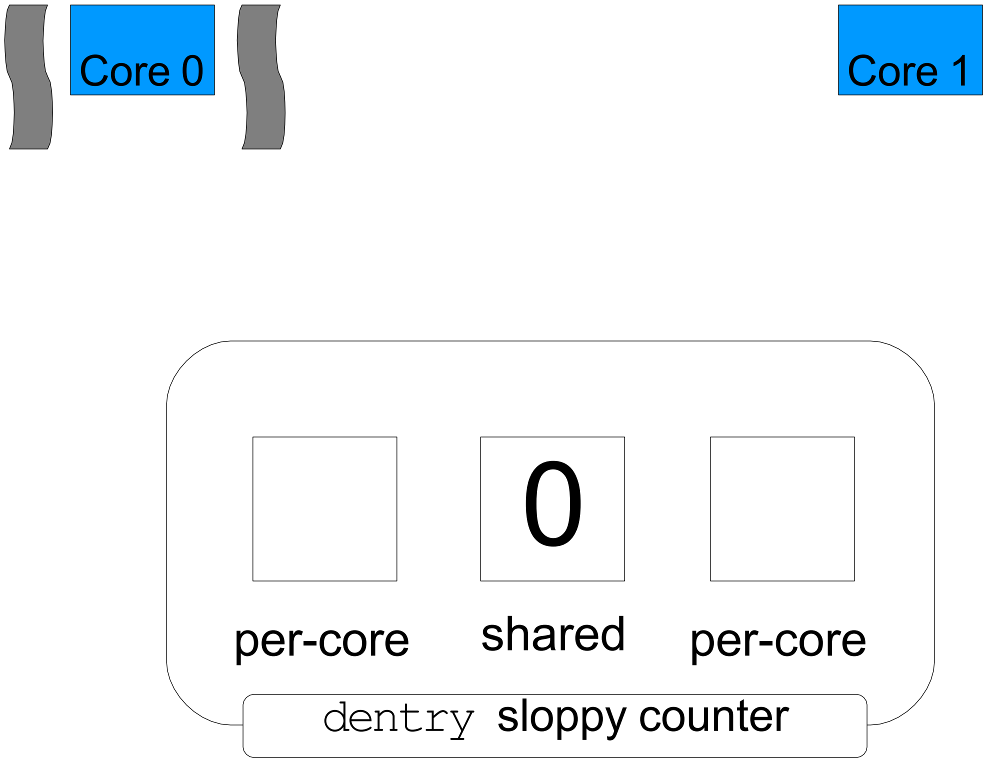


Figure 2: An example of the kernel using a sloppy counter for `dentry` reference counting. A large circle represents a local counter, and a gray dot represents a held reference. In this figure, a thread on core 0 first acquires a reference from the central counter. When the thread releases this reference, it adds the reference to the local counter. Finally, another thread on core 0 is able to acquire the spare reference without touching the central counter.

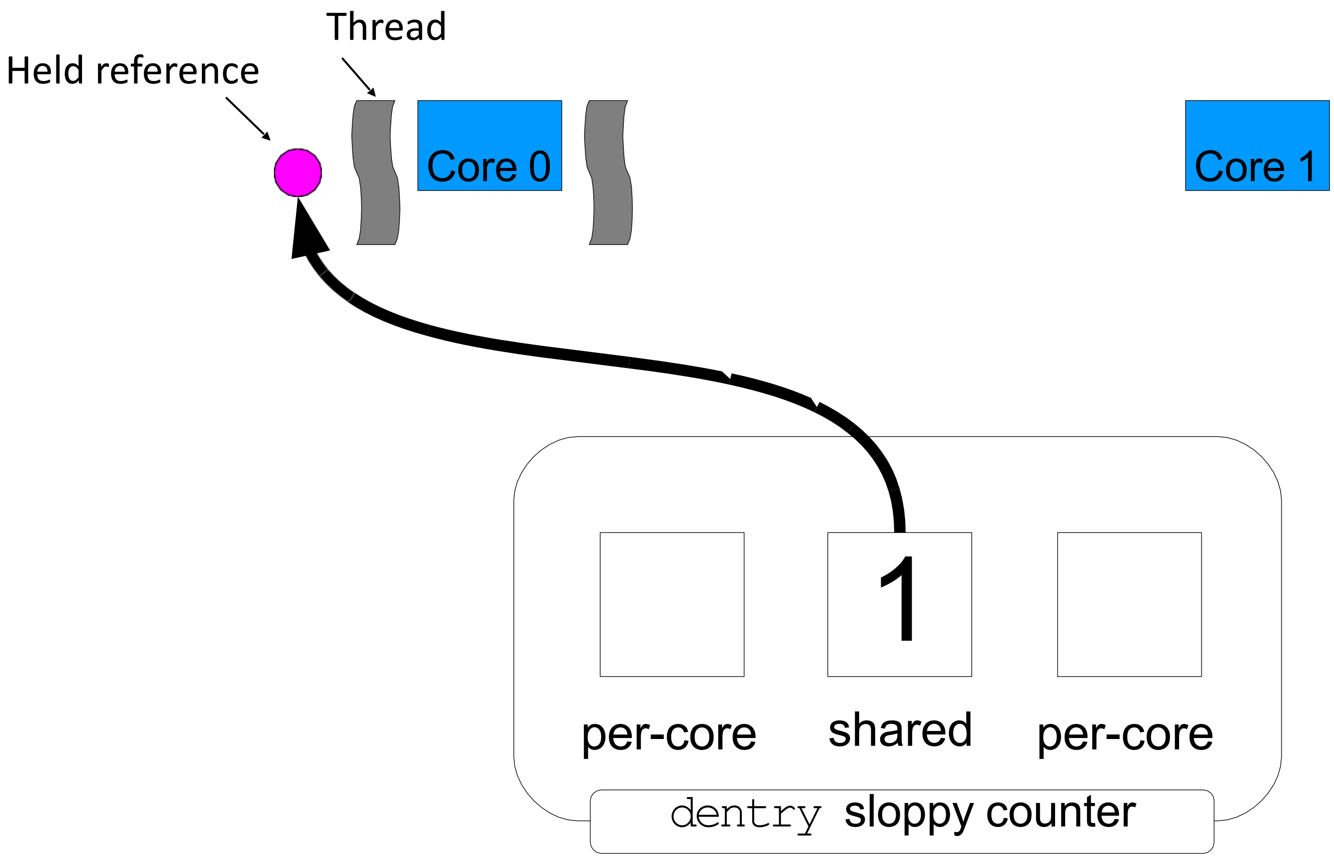
Sloppy counters

- Each core holds a few “spare” counters to an object



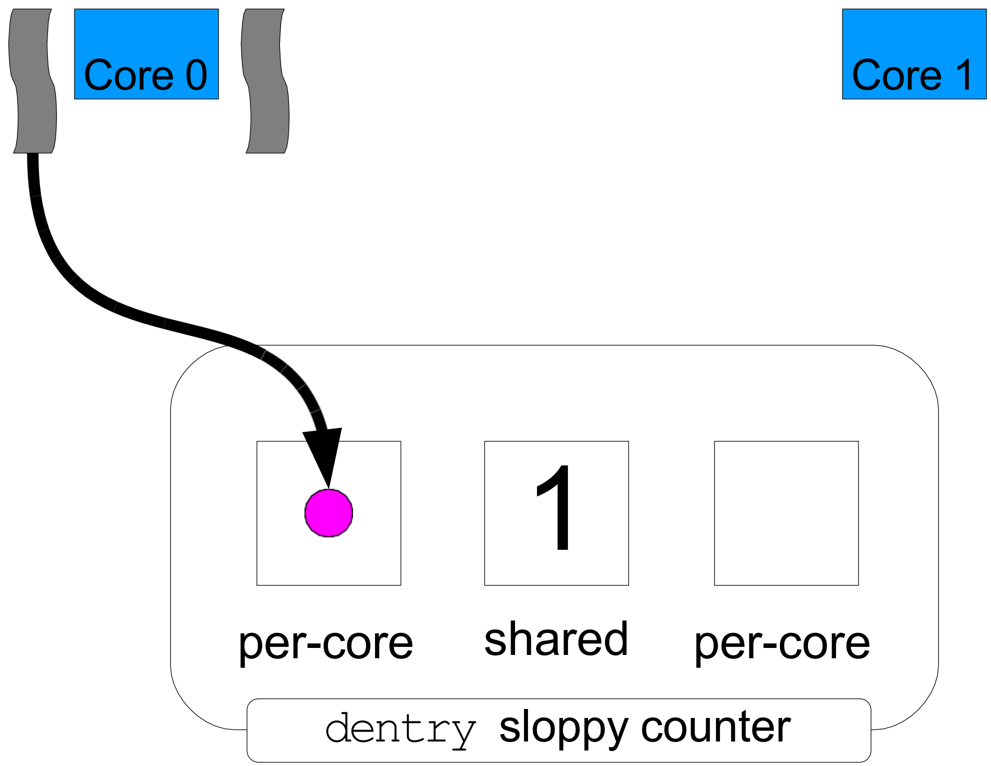
Sloppy counters

- Each core holds a few "spare" counters to an object



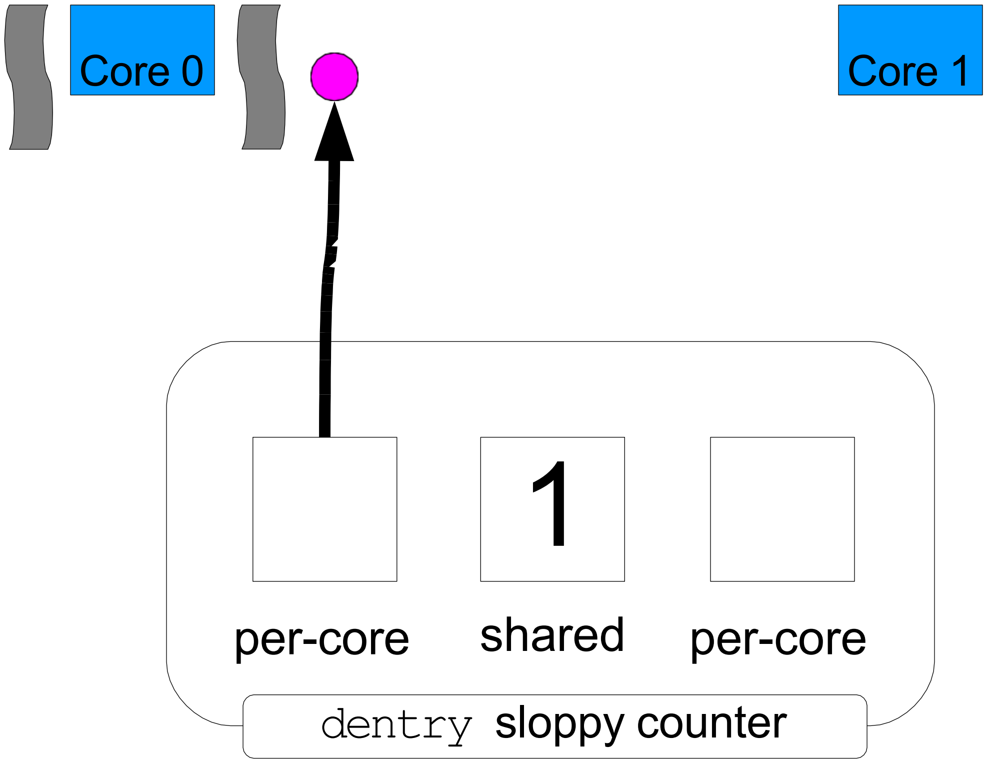
Sloppy counters

- Each core holds a few "spare" counters to an object



Sloppy counters

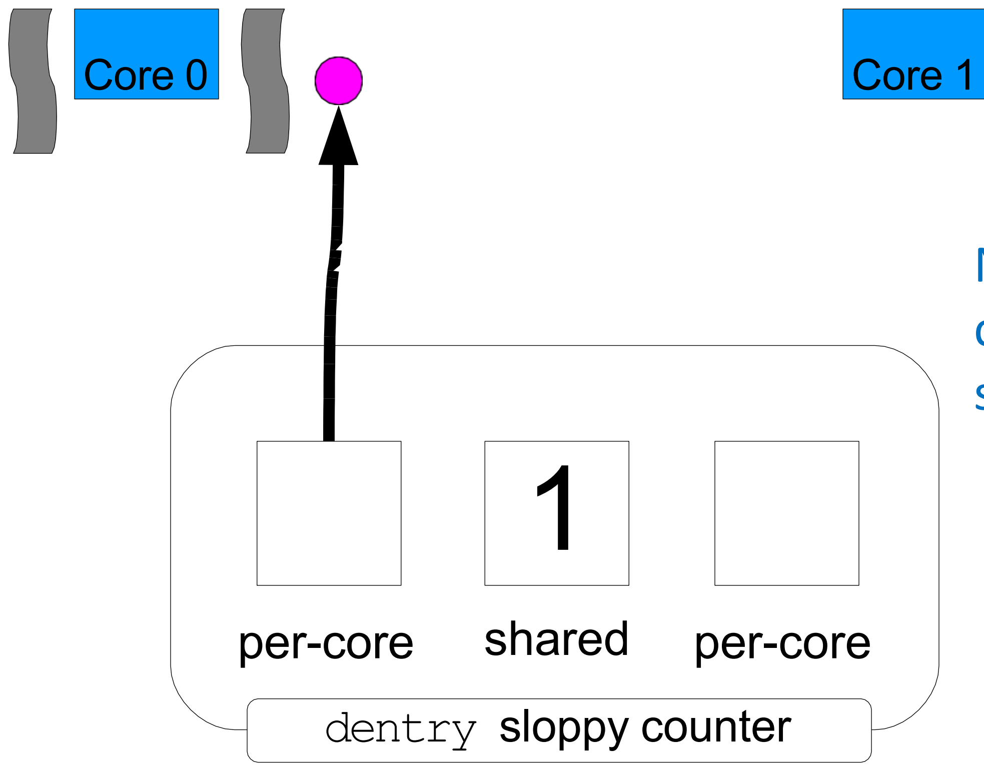
- Each core holds a few "spare" counters to an object



Another thread on core 0 can get the spare local (per-core) reference without updating the central shared counter

Sloppy counters

- Each core holds a few "spare" counters to an object

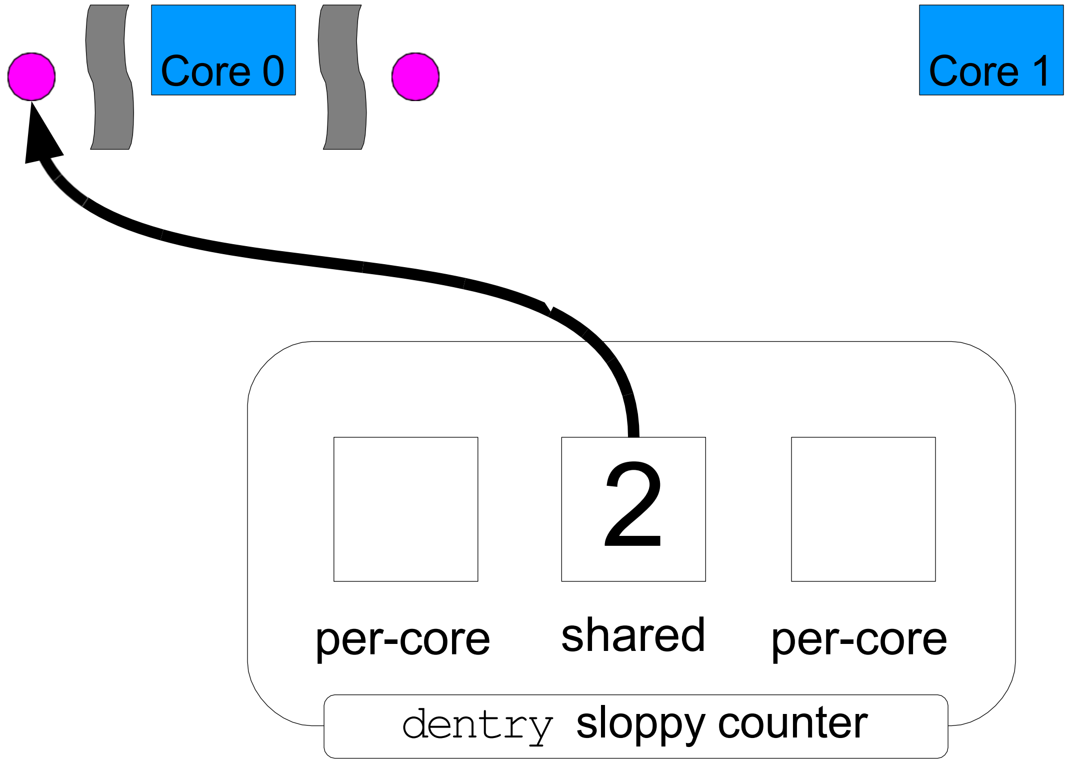


No waiting for locks or cache-coherence serialization

Another thread on core 0 can get the spare local (per-core) reference without updating the central shared counter

Sloppy counters

- Each core holds a few "spare" counters to an object

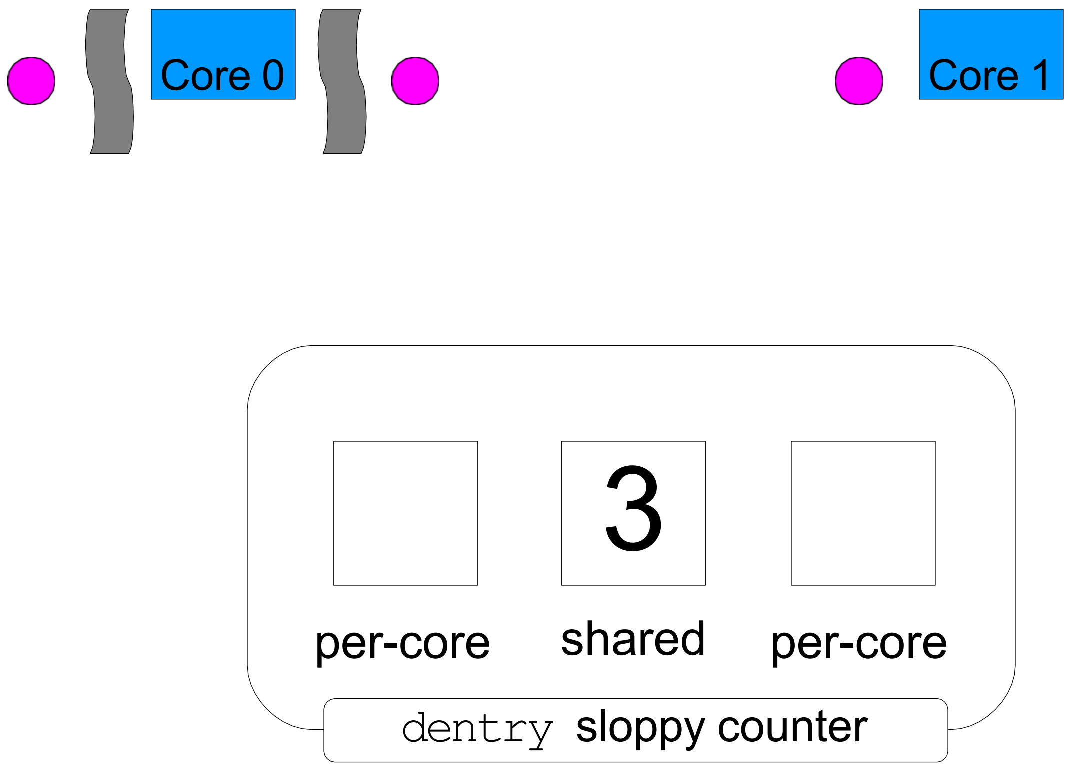


Local reference is insufficient

→ Acquire ref from the central shared counter

Sloppy counters

- Each core holds a few “spare” counters to an object

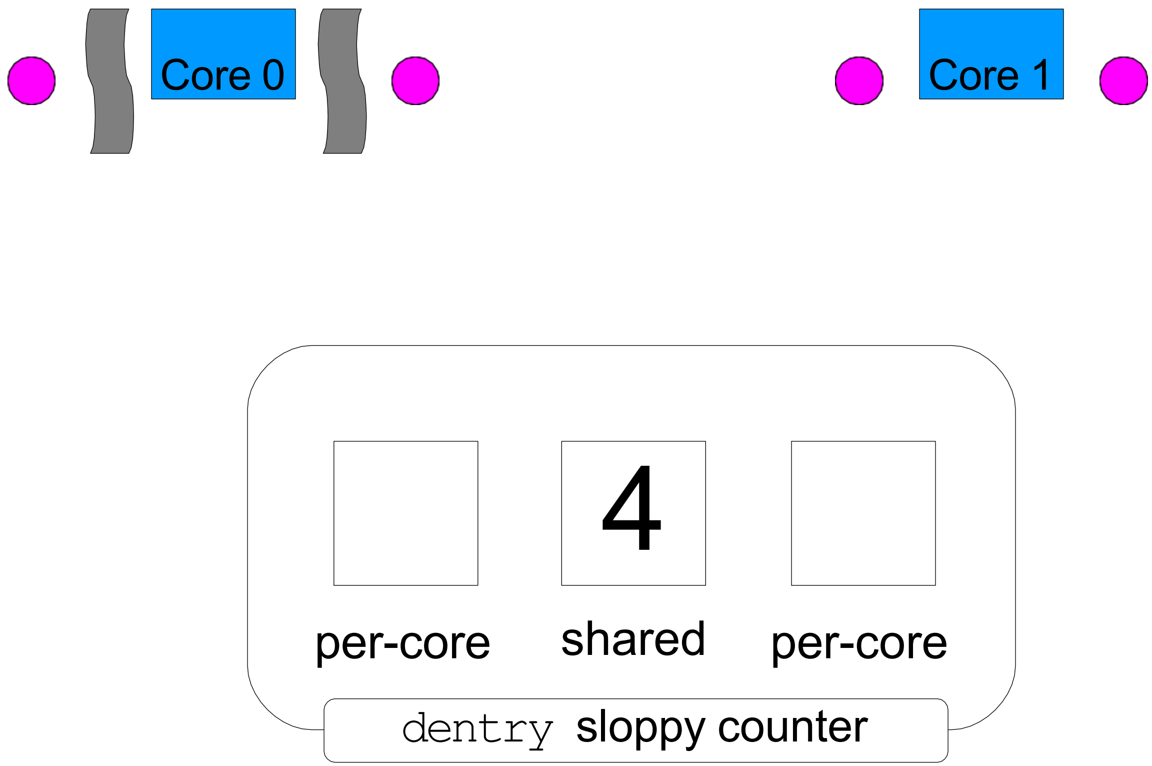


Local reference is insufficient

→ Acquire ref from the central shared counter

Sloppy counters

- Each core holds a few "spare" counters to an object

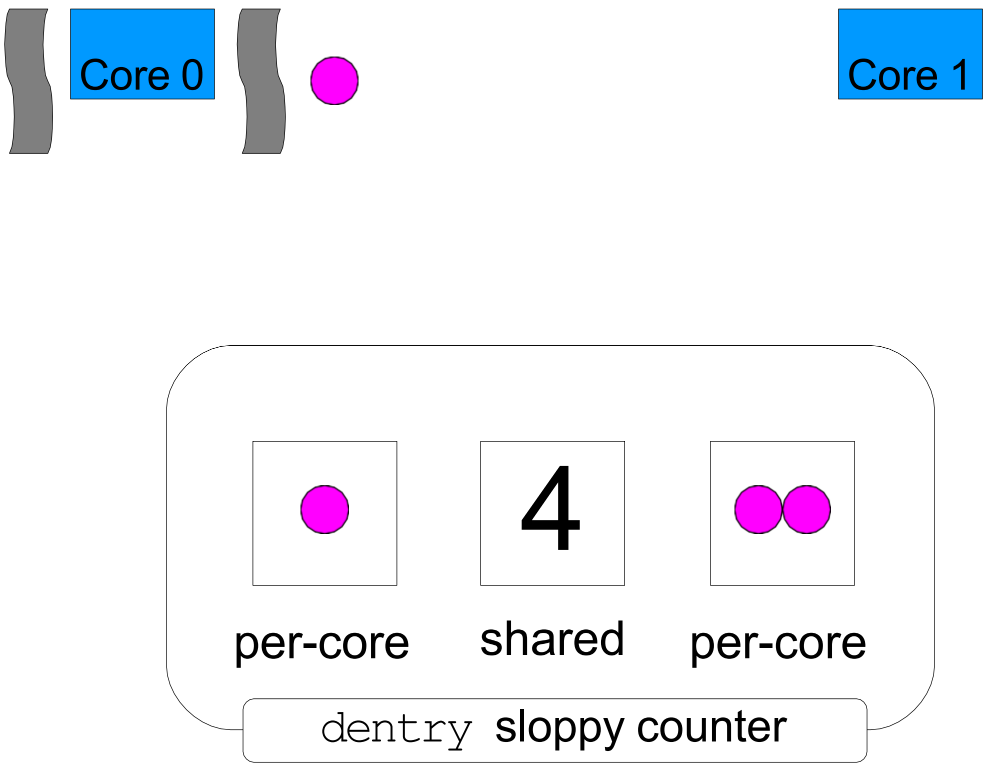


Local reference is insufficient

→ Acquire ref from the central shared counter

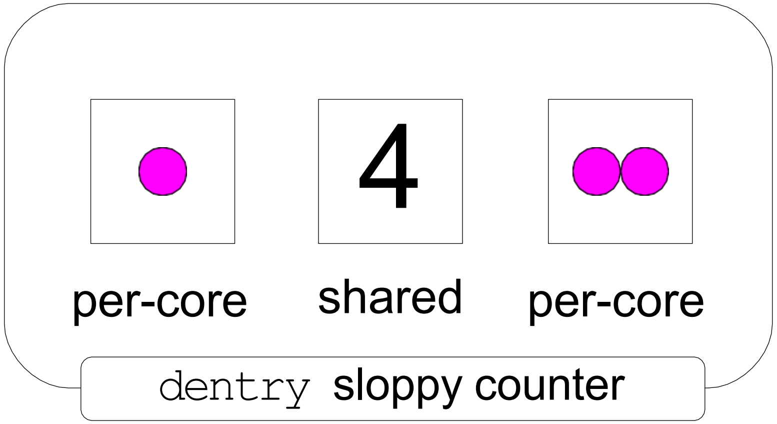
Sloppy counters

- Each core holds a few “spare” counters to an object



Sloppy counters

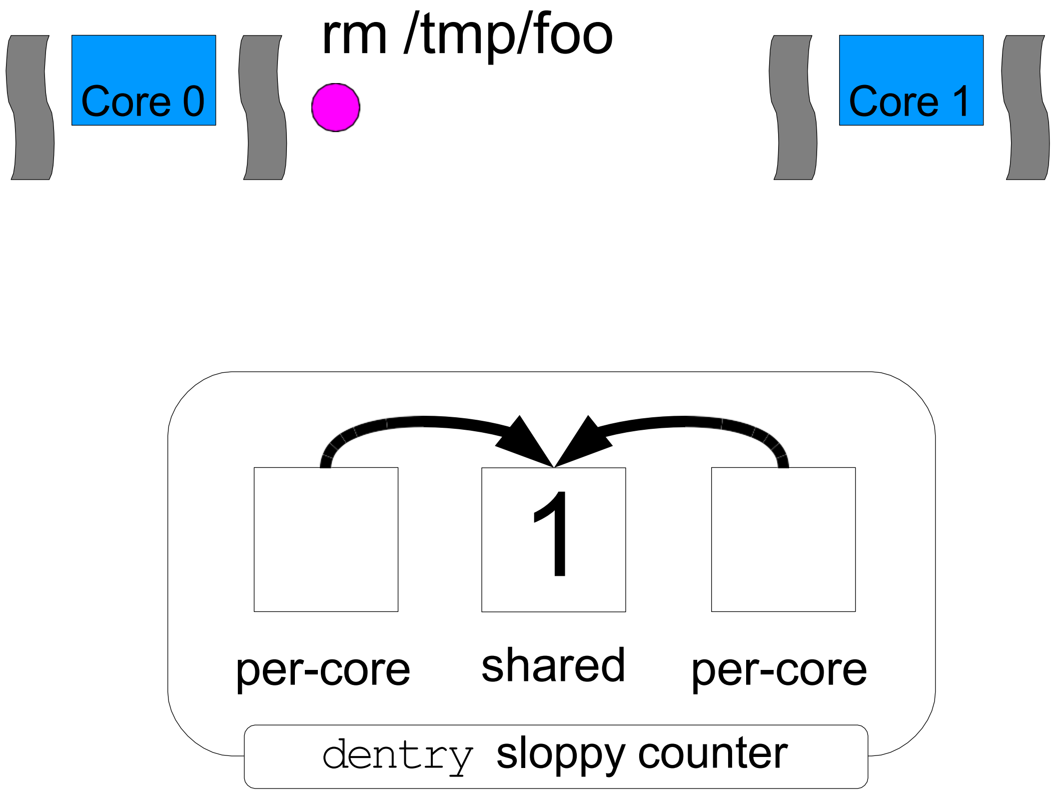
- Each core holds a few "spare" counters to an object



Occasionally, need to reconcile the central and per-core counters

Sloppy counters

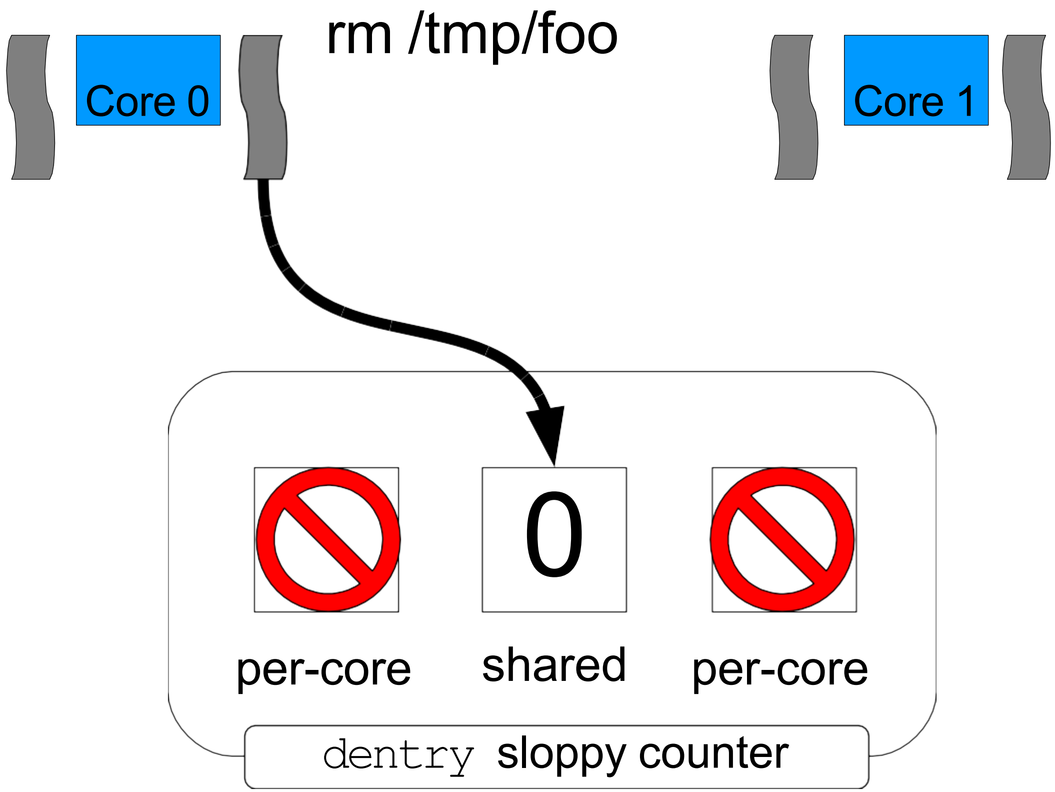
- Each core holds a few “spare” counters to an object



Occasionally, need to reconcile the central and per-core counters

Sloppy counters

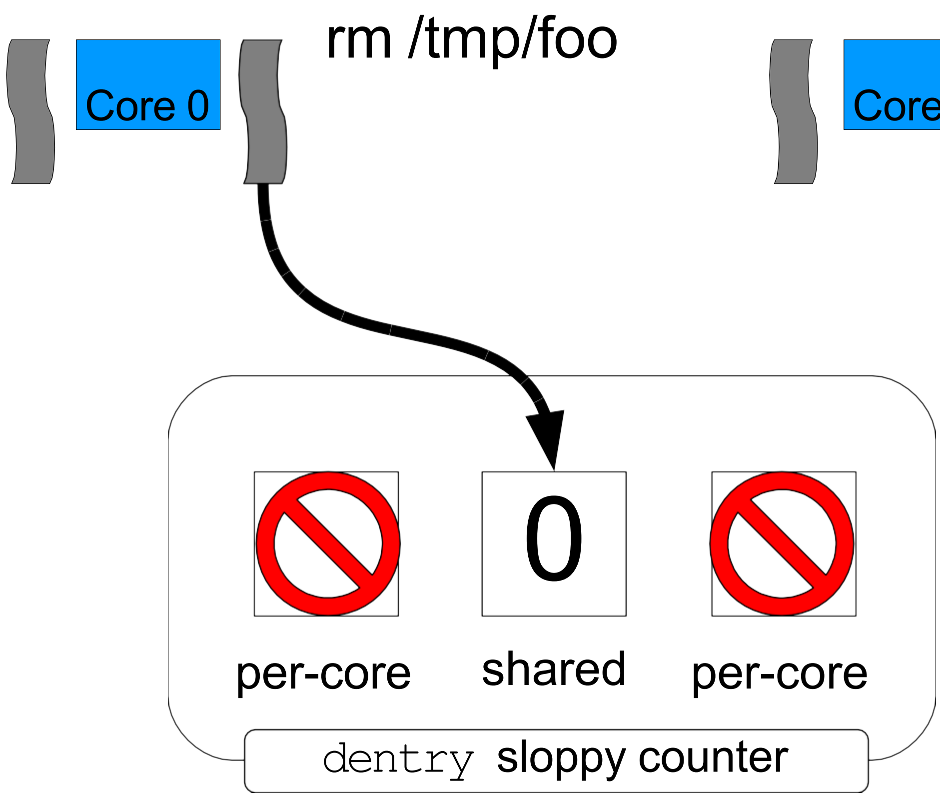
- Each core holds a few “spare” counters to an object



Occasionally, need to reconcile the central and per-core counters

Sloppy counters

- Each core holds a few "spare" counters to an object

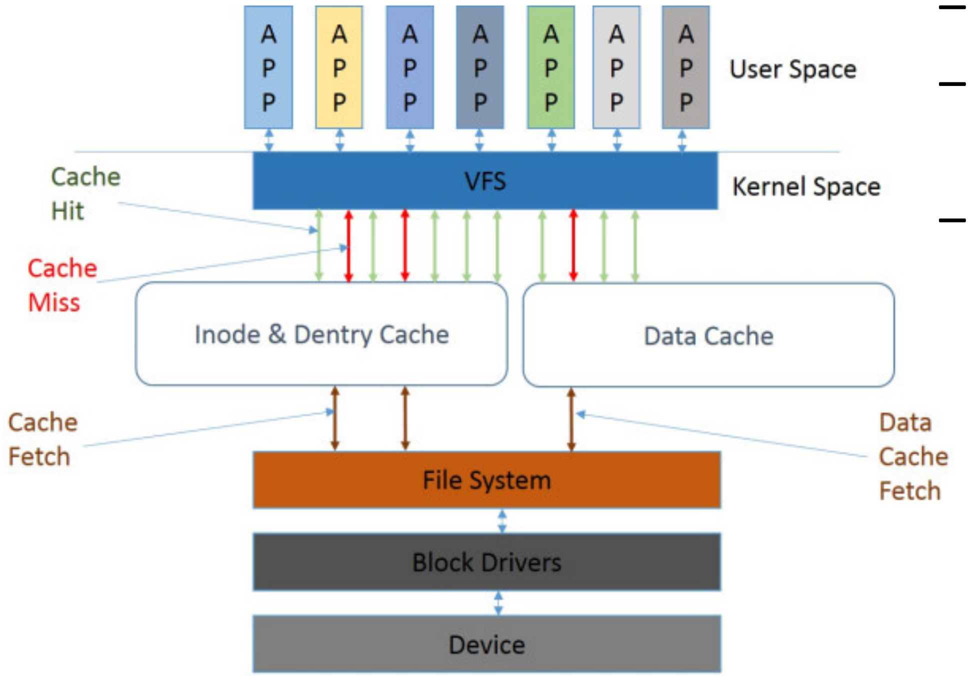


Expensive to reconcile local and global counters
→ Good for objects that are not frequently deallocated

Occasionally, need to reconcile the central and per-core counters

Lock-free comparison

- Observed low scalability in directory entry cache (dentry cache)
- Background: Linux file system



- inode: File metadata
- dentry: Directory structure
- dentry cache:
  - Speeds up name lookup (e.g. /usr)
  - Maps directory/file name to dentry
  - When a potential dentry is found, per-dentry spinlock is used to atomically compare if it matches the requested argument
    - Causes a bottleneck; only one core can lookup at a time

Lock-free comparison

- Solution: lock-free comparison for dentries
- How? Use a *generation counter*
  - Similar to version control
  - Incremented after every modification to the dentry
  - Temporary set to zero during modification
- Multiple cores can access dentry without requesting a lock if
  - Generation counter is not zero (== not being modified)
  - Has not changed

Per-core data structures

- Kernel data structures that caused scaling bottlenecks due to lock contention:
  - Per-super-block list of open files
  - Table of mount points
  - Pool of free packet buffers

- Make these per-core data
  - So that each core uses different data
  - But, may cause increased memory usage and complexity

Eliminating false sharing

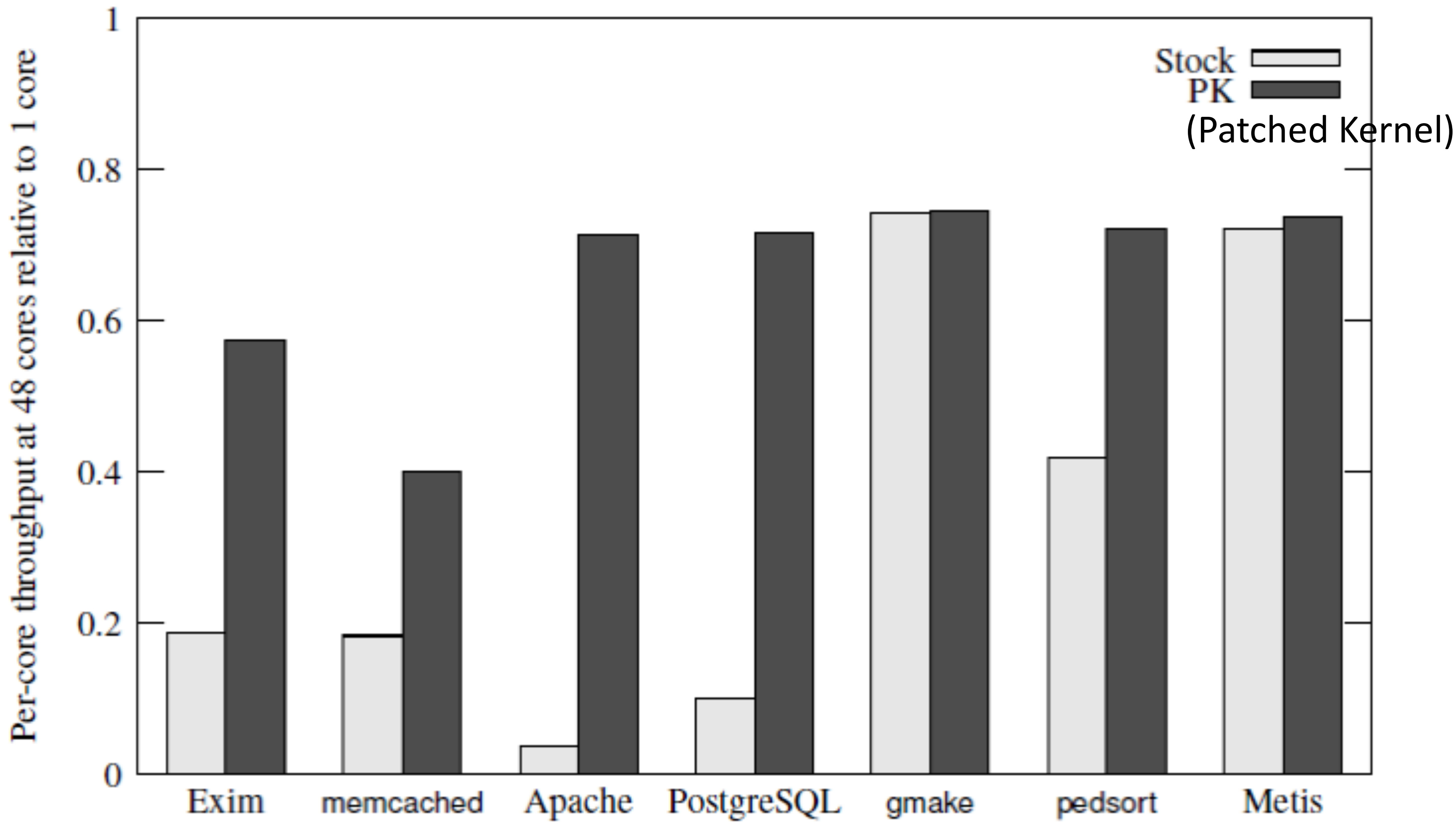
- Problem
  - Two or more threads update different variables that happen to be located in the same cache line
    - Cache line: Minimum data transfer unit between memory and cache
  - Cores contended for the falsely shared line
    - Degraded Exim per-core performance
    - memcached, Apache, and PostgreSQL faced similar false sharing problems

```
static struct {
    int x;
    int y;
} f;
```

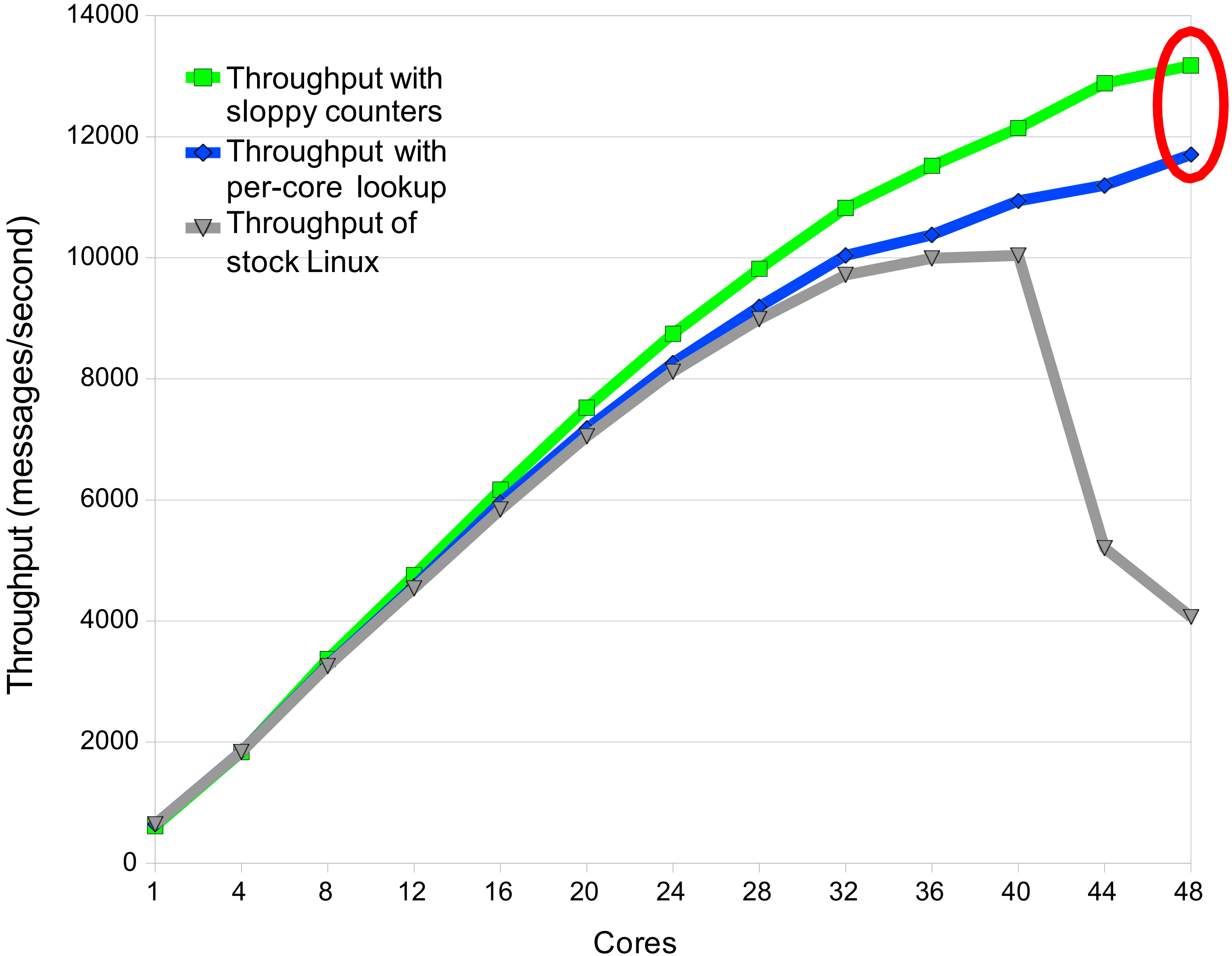
- Solution
  - Placing the heavily modified data on separate cache lines (How?)

Evaluation

Linux kernel 2.6.35-rc5 (July 12,2010)



Exim: Better scalability



Major bottlenecks

Application	Bottleneck
memcached	HW: transmit queues on NIC
Apache	HW: receive queues on NIC
Exim	App: contention on spool directories
gmake	App: serial stages and stragglers
PostgreSQL	App: spin lock
Psearchy	HW: cache capacity
Metis	HW: DRAM throughput

- Kernel code is not the bottleneck
- Further kernel changes might help apps. or hw

Summary

- Most applications can scale well to many cores with modest modifications to the applications and to the kernel
  - Basically making data local to core. Similar to multikernel?
- Results suggest that traditional kernel designs may be able to achieve application scalability on multicore computers
  - More bottlenecks may be revealed when running on more cores

Limitations

- Results limited to 48 cores and small set of applications
- Looming problems
  - fork/virtual memory book-keeping
  - Page allocator
  - File system
  - Concurrent modifications to address space
- In-memory FS instead of disk
- 6 core x 8 chip AMD machine ≠ single 48-core chip