

Plans:

- shared-memory TSP scalability on Beehive
- Case study: AMD CC shared memory
- Case study: Linux scalability

BEEHIVE

Scalability challenges for shared-memory TSP on Beehive

- work distribution
 - master?
 - enough jobs
- bound
 - when to invalidate?

Graphs

- speedup
- performance
- notice speedups are getting less good, but performance is improving!

Optimizations:

Master-less parallelization -OR- two-phase where master becomes a worker when it finishes

Eliminate redundant permutations of inner cities in a tour prefix to prune search space (only the permutation with the shortest path can be the best path)

Seed the min bound by first running a TSP approximation algorithm (e.g. Greedy TSP)

Expanding partial tours according to the greedy heuristic (ordered by path cost)

Use a distributed work queue

Optimize the frequency of bound updates (e.g. update frequently in the beginning then more slowly as time goes on)

Bit vector implementation of present function

CASE STUDY: AMD CACHE COHERENCE

- AMD slides
- topology of the machine in the paper (3 Hyperlinks per core):

```
DRAM -1 -- 3 -- 5
      |      |  x |
DRAM -2 -- 4 -- 6
```

CASE STUDY: Linux

- Big parallel program
- OSDI slides



Blade Computing with the **AMD Opteron™ Processor (“Magny-Cours”)**

Pat Conway (Presenter)

Nathan Kalyanasundharam

Gregg Donley

Kevin Lepak

Bill Hughes





Agenda

Processor Architecture

- AMD driving the x86 64-bit processor evolution
- Driving forces behind the Twelve-Core AMD Opteron™ processor codenamed “Magny-Cours”
- CPU silicon
- MCM 2.0 package, speeds and feeds






























Performance and scalability

- 2P/4P blade and rack topologies
- HyperTransport™ technology HT Assist design
 - Cache coherence protocol
 - Transaction scenarios and frequencies
 - Coverage ratio
 - Memory latency and bandwidth

A look ahead



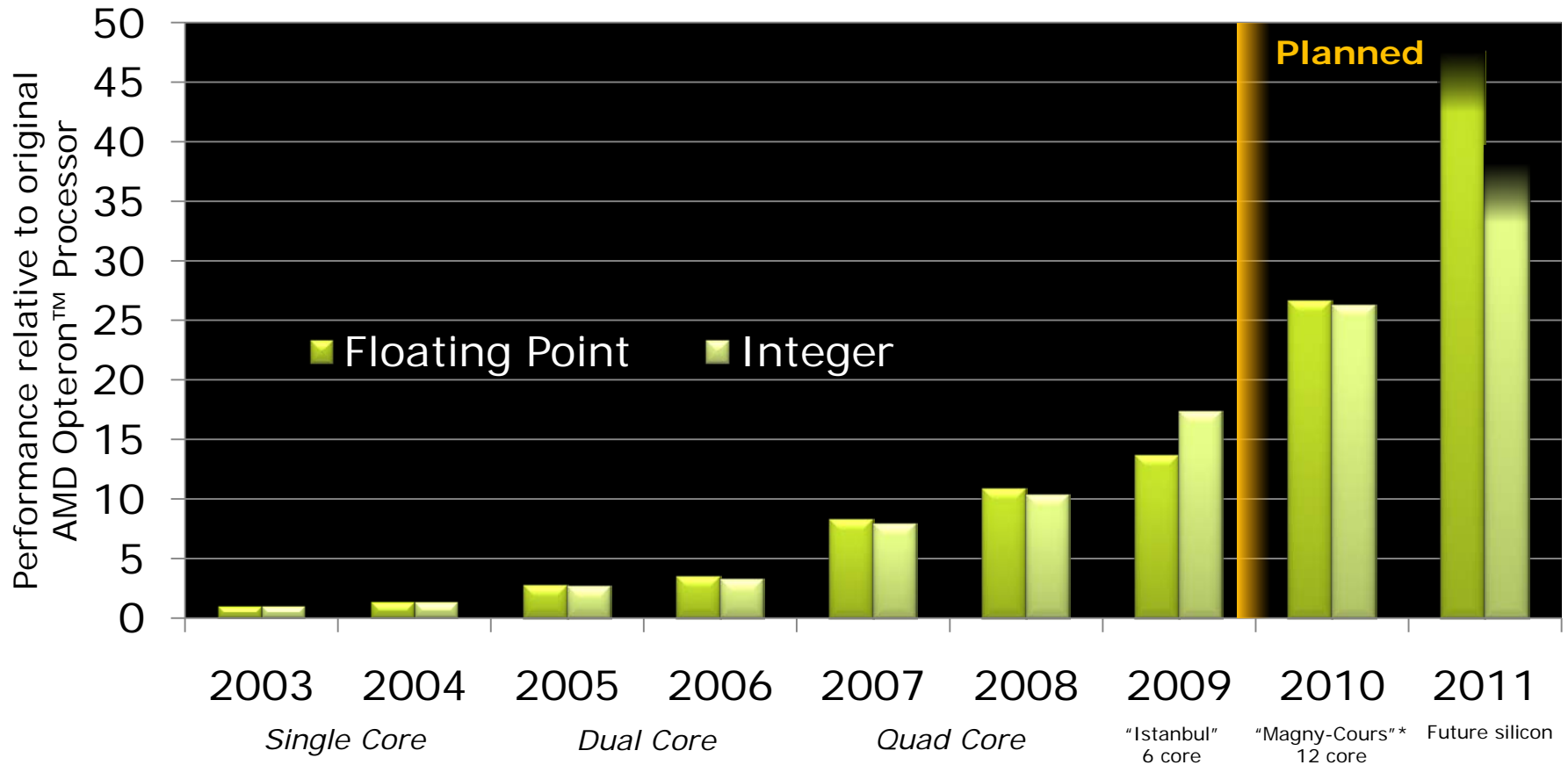
x86 64-bit Architecture Evolution

	2003	2005	2007	2008	2009	2010
	AMD Opteron™	AMD Opteron™	"Barcelona"	"Shanghai"	"Istanbul"	"Magny-Cours"
Mfg. Process	90nm SOI	90nm SOI	65nm SOI	45nm SOI	45nm SOI	45nm SOI
CPU Core	K8 	K8  	Greyhound    	Greyhound+    	Greyhound+      	Greyhound+            
L2/L3	1MB/0	1MB/0	512kB/2MB	512kB/6MB	512kB/6MB	512kB/12MB
Hyper Transport™ Technology	3x 1.6GT/s	3x 1.6GT/s	3x 2GT/s	3x 4.0GT/s	3x 4.8GT/s	4x 6.4GT/s
Memory	2x DDR1 300	2x DDR1 400	2x DDR2 667	2x DDR2 800	2x DDR2 1066	4x DDR3 1333

Max Power Budget Remains Consistent



Dramatic Back-to-back Gains



"Shanghai" to "Istanbul" delivers 34% more performance in the same power envelope

**"Magny-Cours" and Future silicon data is based on AMD projections*



Driving Forces Behind “Magny-Cours”

Server Throughput

- Exploit thread level parallelism
- Leverage Directly Connected MCM 2.0

Virtualization

- Maximize compute density in 2P/4P blades and racks
- Run more VMs per server
- Provide hardware context (thread) based QOS

Energy Proportional Computing

- More performance, same power envelope
- Power conservation when idle

Economics

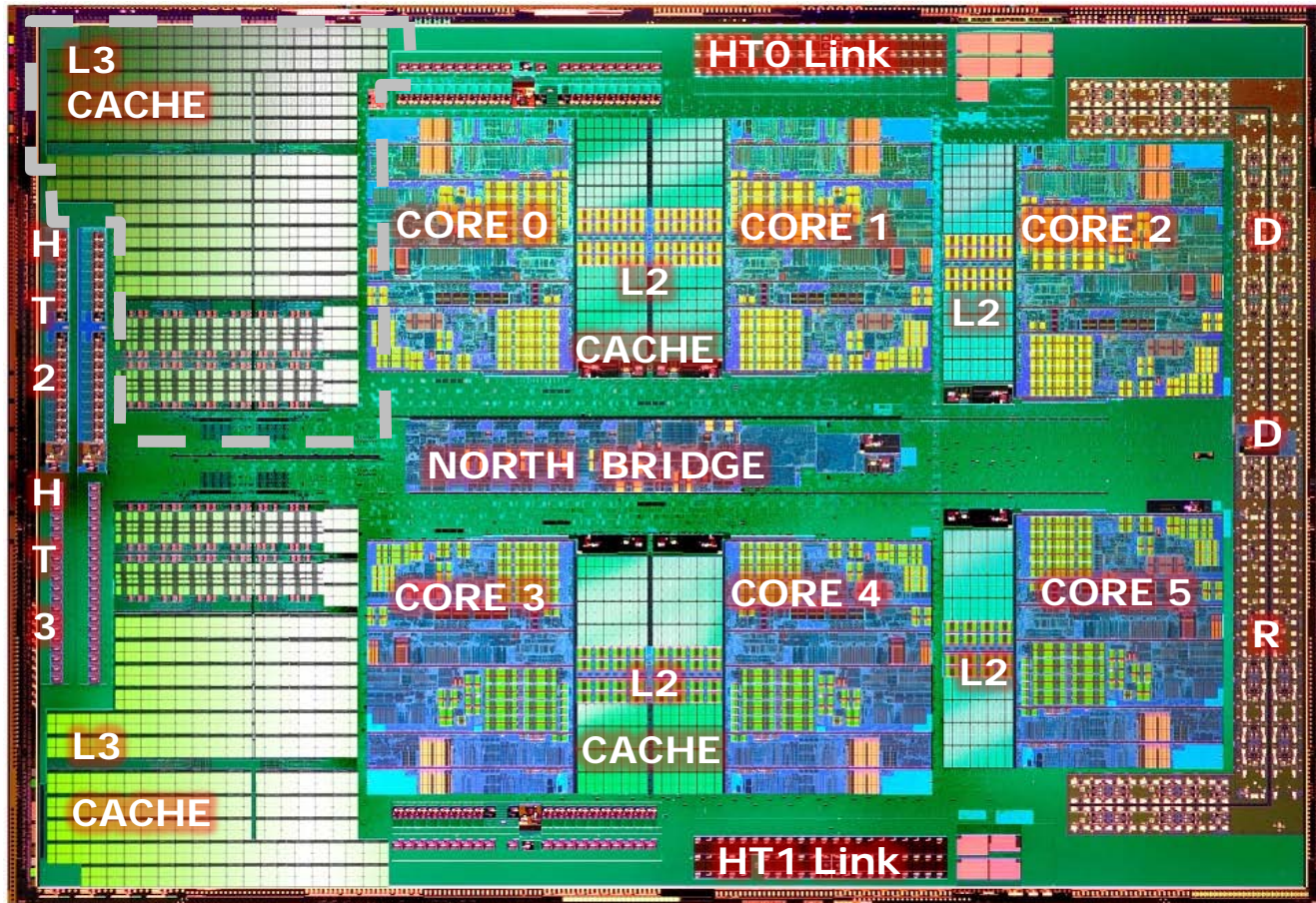
- Design efficiency – “Magny-Cours” silicon same as “Istanbul”
 - *Can help speed qualification times and customers’ time to market*
- Reasonable die size permits 2 die per reticle (Yield ↑ Manufacturing Cost ↓)
 - *Yield improvements can help ensure supply chain stability*
 - *Manufacturing cost savings ultimately benefit customers*



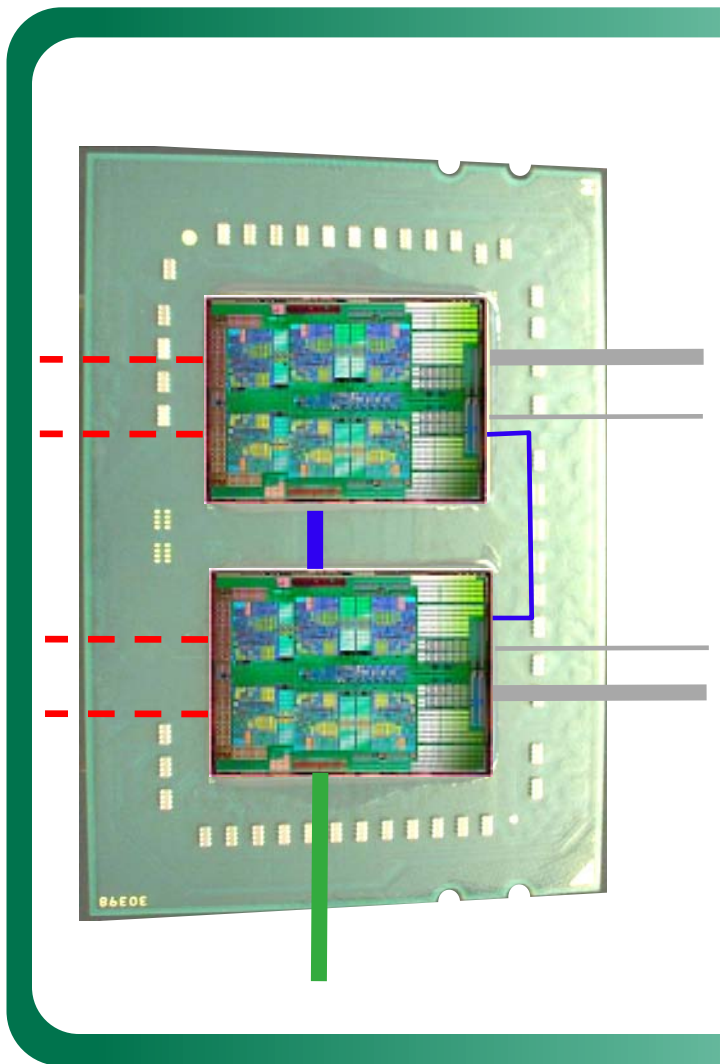
"Magny-Cours" Silicon



GLOBALFOUNDRIES



MCM 2.0 Logical View



G34 Socket

"Magny-Cours" utilizes a ***Directly Connected*** MCM

Package has 12 cores,
4 HT ports, & 4 memory
channels

Die (Node) has 6 cores,
4 HT ports & 2 memory
channels

DDR3 Memory
Channel

P0

x16 cHT

x8 cHT

x16
(NC)

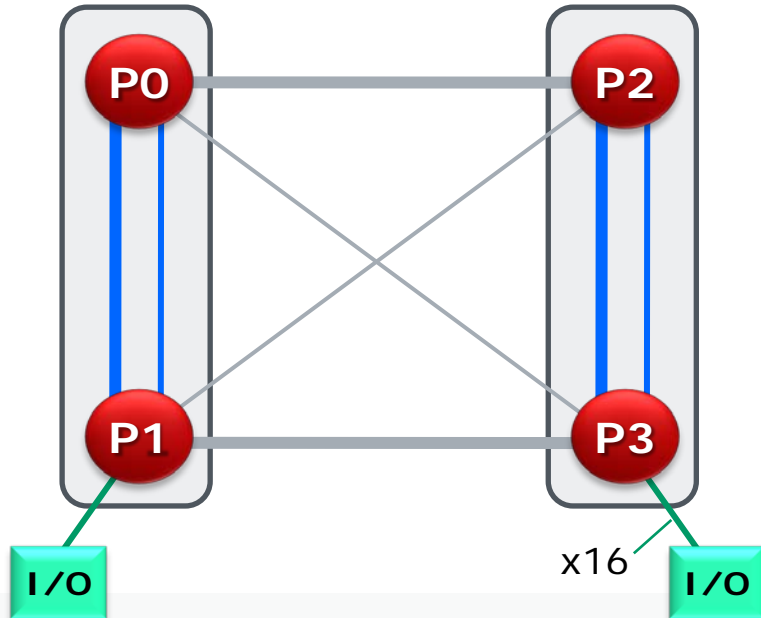
P1

x16 cHT



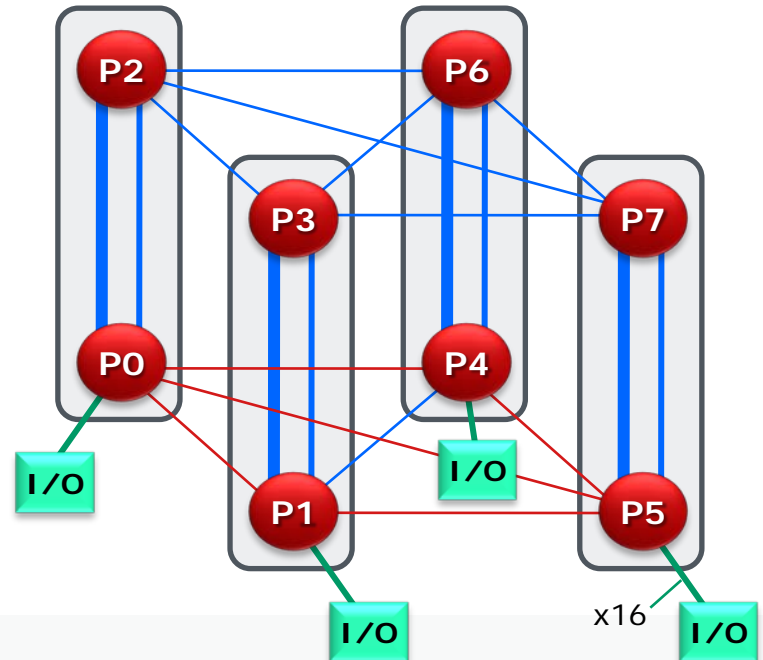
Topologies

2P



Diameter 1
Avg Diam 0.75
DRAM BW 85.6 GB/s
XFIRE BW 71.7 GB/s (*)

4P

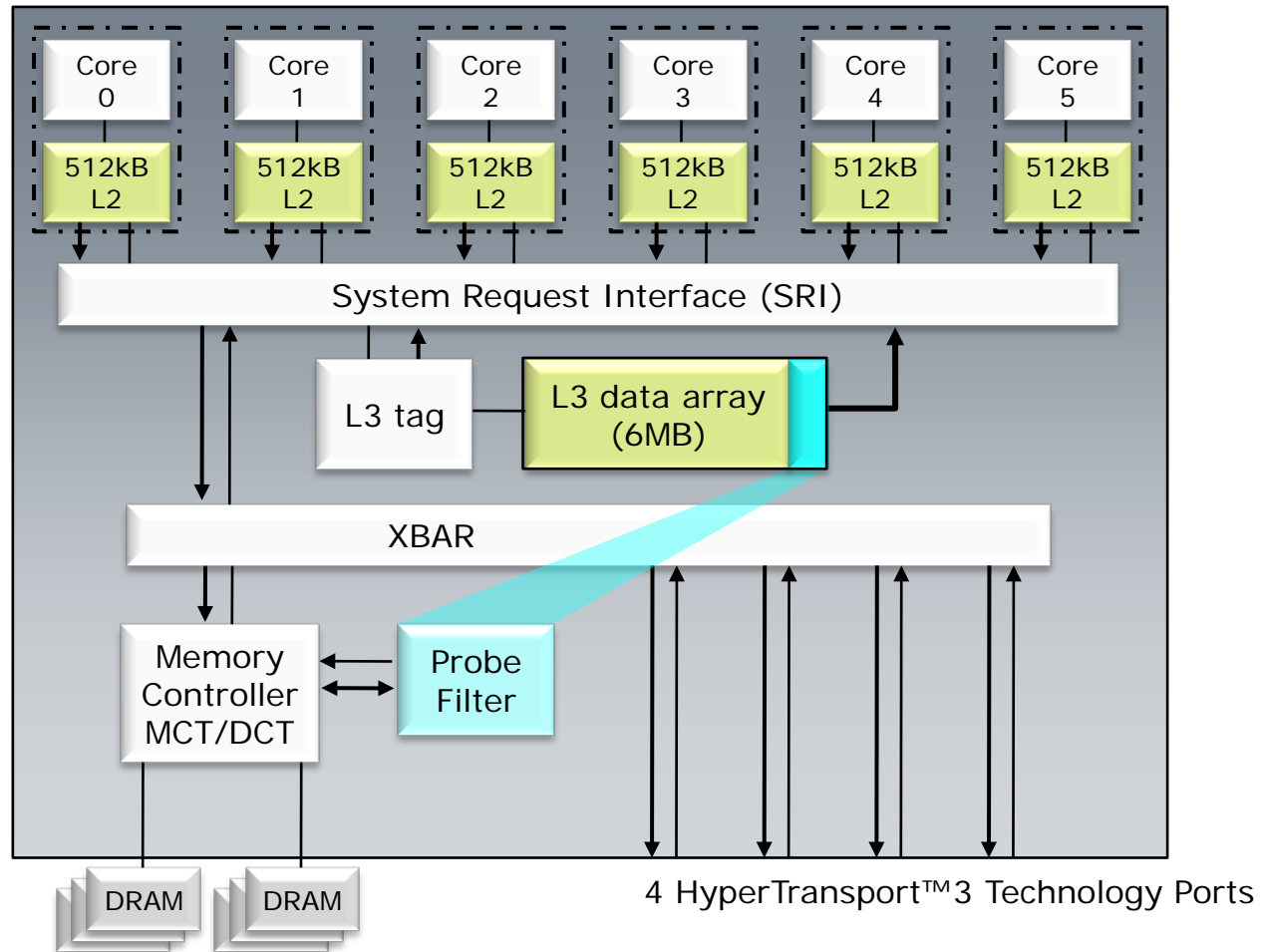


Diameter 2
Avg Diam 1.25
DRAM BW 170.4 GB/s
XFIRE BW 143.4 GB/s



Block Diagram

"Magny-Cours" Die (Node)



HyperTransport™ Technology HT Assist (Probe Filter)

Key enabling technology on “Istanbul” and
“Magny-Cours”

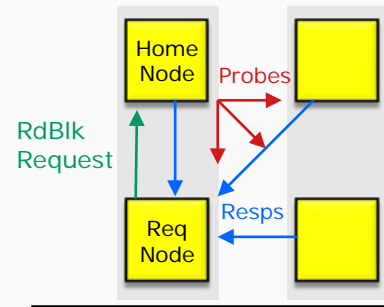
HT Assist is a sparse directory cache

- Associated with the memory controller on the home node
- Tracks all lines cached in the system from the home node

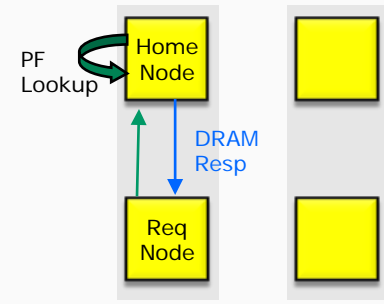
Eliminates most probe broadcasts (see diagram)

- Lowers latency
 - local accesses get local DRAM latency, no need to wait for probe responses
 - less queuing delay due to lower HT traffic overhead
- Increases system bandwidth by reducing probe traffic

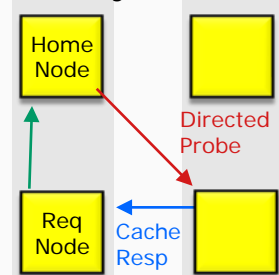
“Old” broadcast protocol



PF – clean data



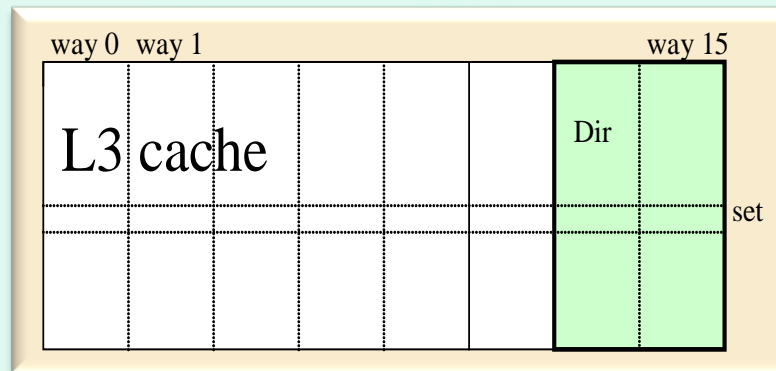
PF – dirty data



Where Do We Put the HT Assist Probe Filter?

Q: Where do we store probe filter entries without adding a large on-chip probe filter RAM which is not used in a 1P desktop system?

A: Steal 1MB of 6MB L3 cache per die in “Magny-Cours” systems



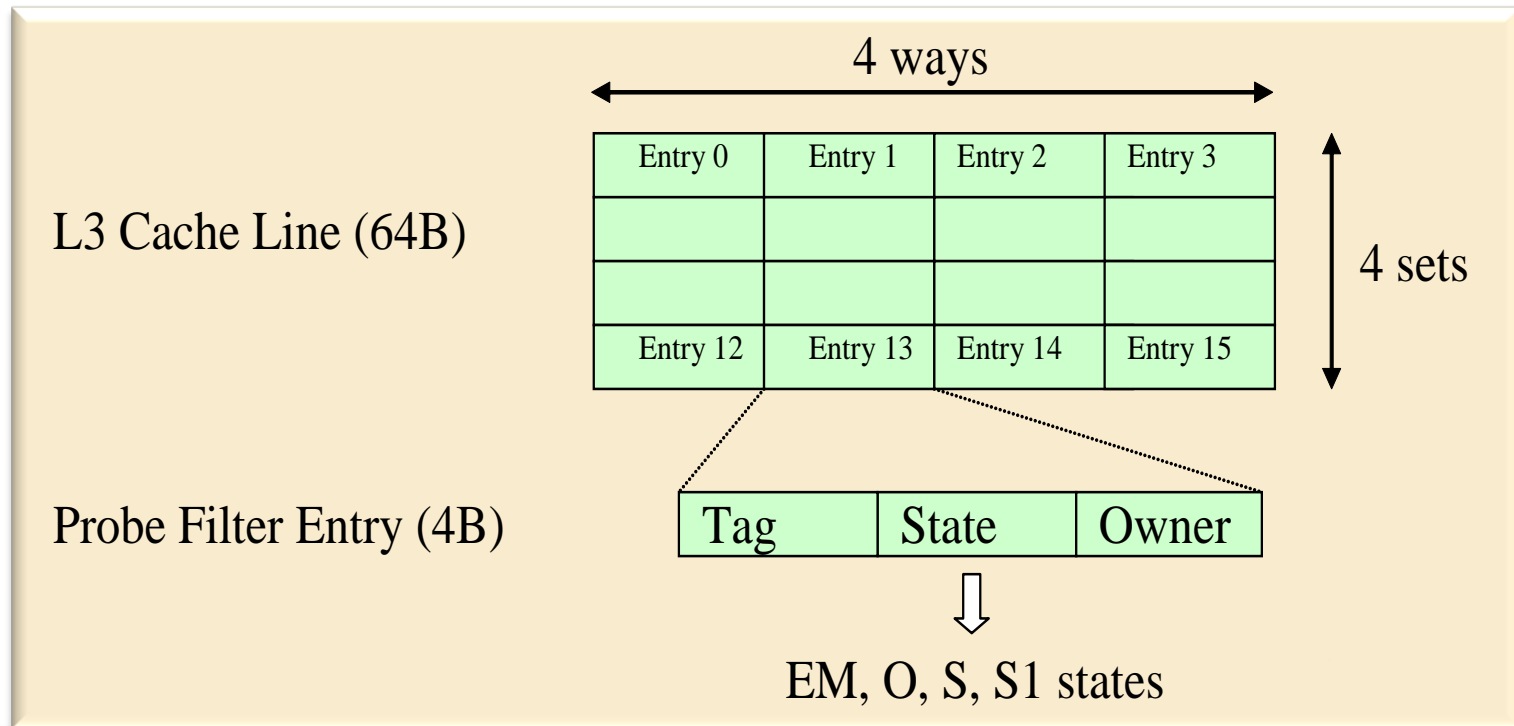
Implementation in fast SRAM (L3) minimizes

- Access latency
- Port occupancy of read-modify-write operations
- Indirection latency for cache-to-cache transfers




Format of a Probe Filter Entry

- 16 probe filter entries per L3 cache line (64B), 4B per entry, 4-way set associative
- 1MB of a 6MB L3 cache per die holds 256k probe filter entries and covers 16MB of cache



Cache Coherence Protocol

- Track lines in M, E, O or S state in probe filter
- PF is fully inclusive of all cached data in system
 - if a line is cached, then a PF entry must exist.
- Presence of probe filter entry says line in M, E, O or S state
-  ■ Absence of probe filter entry says line is uncached
- New messages
 - Directed probe on probe filter hit
 - Replacement notification E ->I (clean VicBlk)



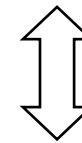
Probe Filter Transaction Scenarios

	PF Hit					PF Miss (*)				
	I	O	S	S1	EM	I	O	S	S1	EM
FETCH	-	D	-	-	D	-	B	B	DI	DI
LOAD	-	D	-	-	D	-	B	B	DI	DI
STORE	-	B	B	B	DI	-	B	B	DI	DI

Legend

-	Filtered
D	Directed
DI	Directed Invalidate
B	Broadcast Invalidate

"Effective"



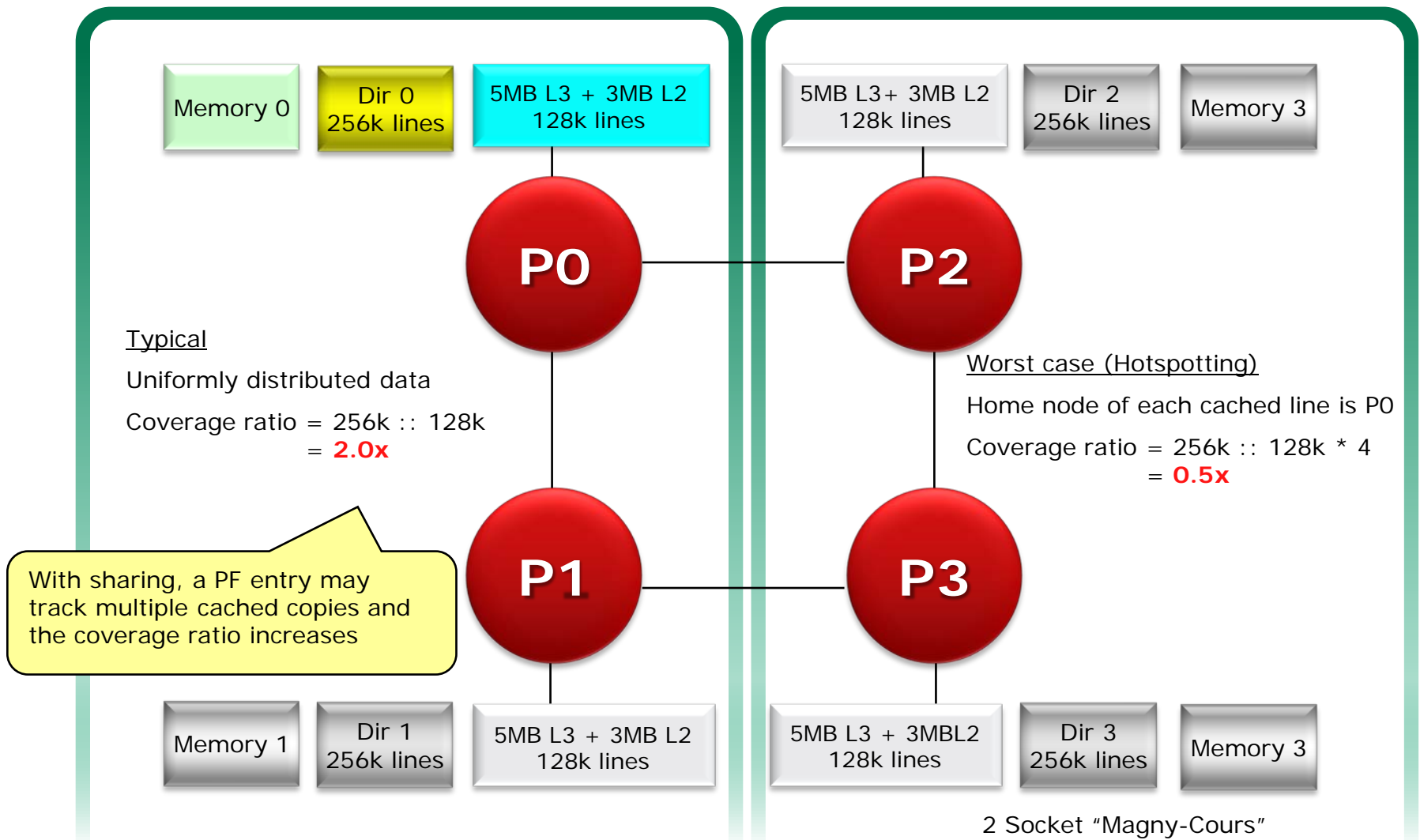
"Ineffective"

(*) PF miss implies line is Uncached (no broadcast necessary). State refers to the state of the line to be replaced upon allocation of new PF entry.

Traditional "Cache Hit Ratio" does not measure effectiveness of probe filter



Probe Filter Coverage Ratio



HT Assist and Memory Latency

With “old” broadcast coherence protocol, the latency of a memory access is the longer of 2 paths:

- time it takes to return data from DRAM and
- the time it takes to probe all caches

With HT Assist, local memory latency is significantly reduced as it is not necessary to probe caches on other nodes.

Several server workloads naturally have ~100% local accesses

- SPECint®, SPECfp®
- VMARK™ typically run with 1 VM per core
- SPECpower_ssj® with 1 JVM per core
- STREAM

Probe Filter amplifies benefit of any NUMA optimizations in OS/application which make memory accesses local



SPEC, SPECint, SPECfp, and SPECpower_ssj are trademarks or registered trademarks of the Standard Performance Evaluation Corporation.

A Look Ahead

Socket compatible upgrade to “Magny-Cours” is planned with

- More cores for additional thread-level parallelism
- More cache to maintain cache-per-core balance
- Same power envelope
- Finer grain power management

New processor core (“Bulldozer”)

- Planned brand new x86 64-bit microarchitecture
- 32nm design
- Instruction set extensions
- Higher memory level parallelism



Thank you!



Disclaimer & Attribution

DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2009 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, AMD Opteron, and combinations thereof are trademarks of Advanced Micro Devices, Inc. HyperTransport is a licensed trademark of the HyperTransport Technology Consortium. Other names are for informational purposes only and may be trademarks of their respective owners.

SPEC, SPECint, SPECfp, and SPECpower_ssj are trademarks or registered trademarks of the Standard Performance Evaluation Corporation.



XXX

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

What is scalability?

- Application does N times as much work on N cores as it could on 1 core
- Scalability may be limited by Amdahl's Law:
 - Locks, shared data structures, ...
 - Shared hardware (DRAM, NIC, ...)

Why look at the OS kernel?

- Many applications spend time in the kernel
 - E.g. On a uniprocessor, the Exim mail server spends 70% in kernel
- These applications should scale with more cores
- If OS kernel doesn't scale, apps won't scale

Speculation about kernel scalability

- Several kernel scalability studies indicate existing kernels don't scale well
- Speculation that fixing them is hard
- New OS kernel designs:
 - Corey, Barrelfish, fos, Tessellation, ...
- How serious are the scaling problems?
- How hard is it to fix them?
- Hard to answer in general, but we shed some light on the answer by analyzing Linux scalability

Analyzing scalability of Linux

- Use a off-the-shelf 48-core x86 machine
- Run a recent version of Linux
 - Used a lot, competitive baseline scalability
- Scale a set of applications
 - Parallel implementation
 - System intensive

Contributions

- Analysis of Linux scalability for 7 real apps.
 - Stock Linux limits scalability
 - Analysis of bottlenecks
- Fixes: 3002 lines of code, 16 patches
 - Most fixes improve scalability of multiple apps.
 - Remaining bottlenecks in HW or app
 - Result: no kernel problems up to 48 cores

Method

- Run application
 - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application
- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

Run application

- Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application
- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
 - Use in-memory file system to avoid disk bottleneck
- ➔ Find bottlenecks
- Fix bottlenecks, re-run application
- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Method

- Run application
 - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- ➔ Fix bottlenecks, re-run application
- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

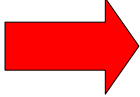
Method

Run application

- Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application
- Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

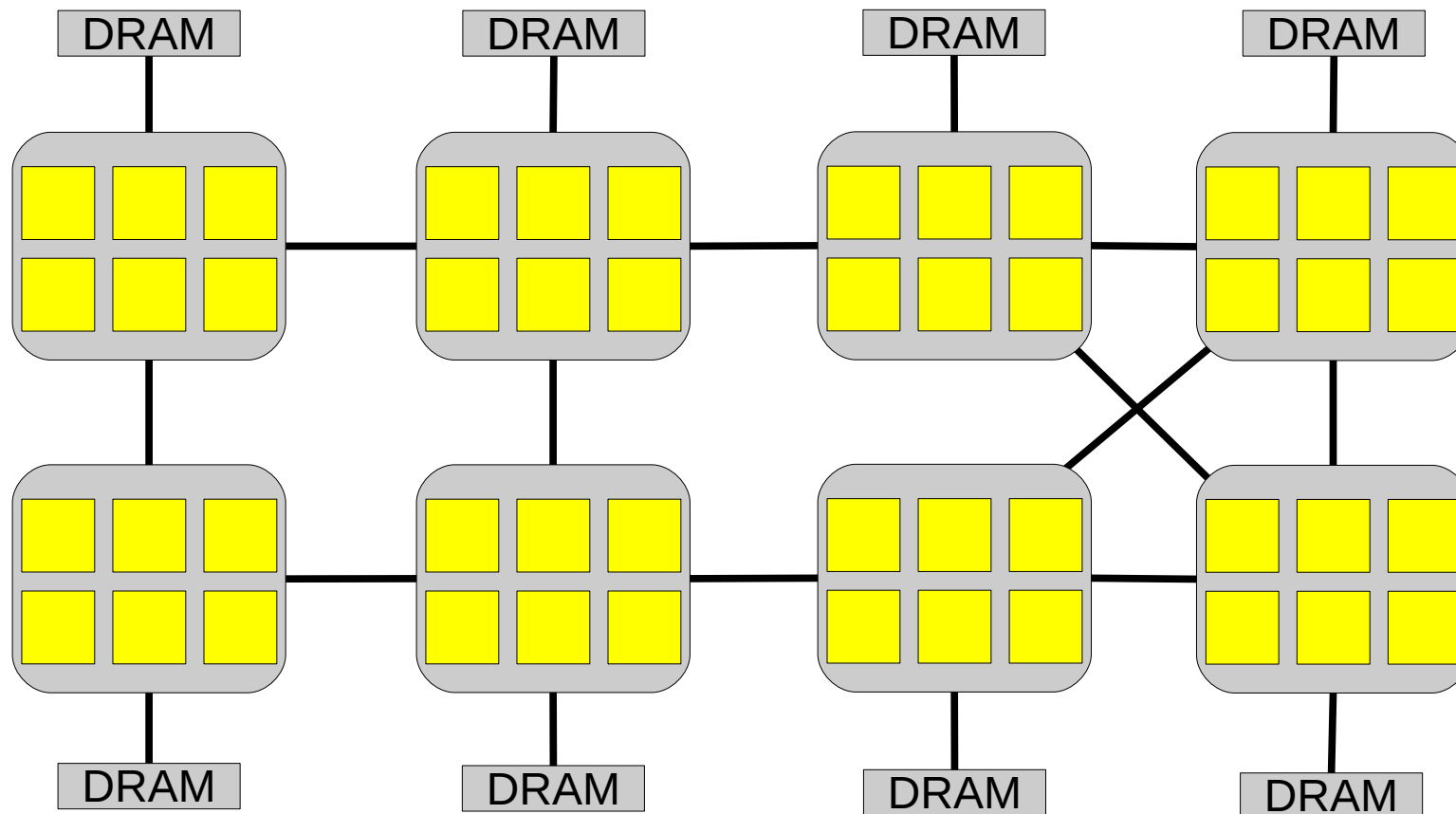
Method

- Run application
 - Use in-memory file system to avoid disk bottleneck
- Find bottlenecks
- Fix bottlenecks, re-run application

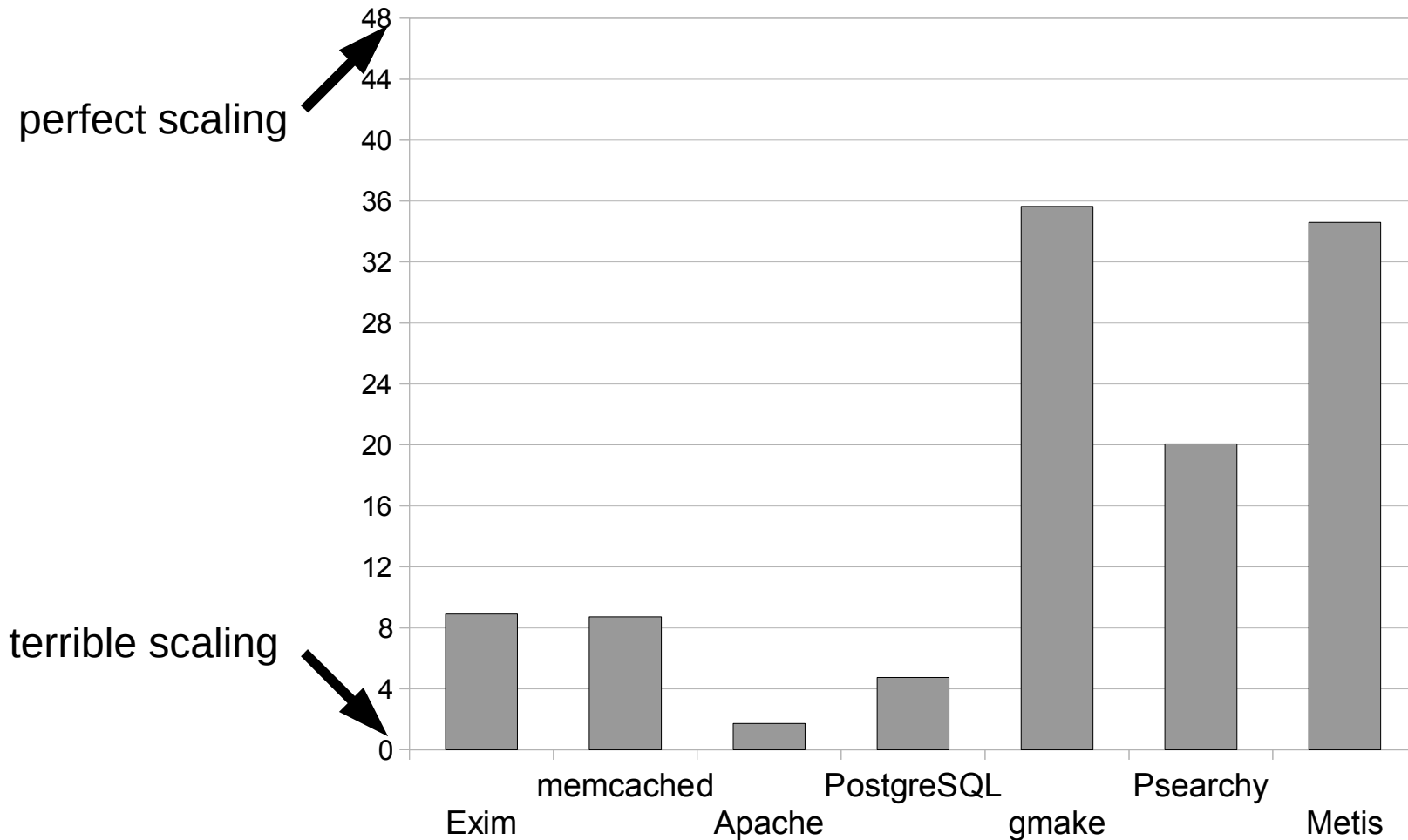
 Stop when a non-trivial application fix is required, or bottleneck by shared hardware (e.g. DRAM)

Off-the-shelf 48-core server

- 6 core x 8 chip AMD

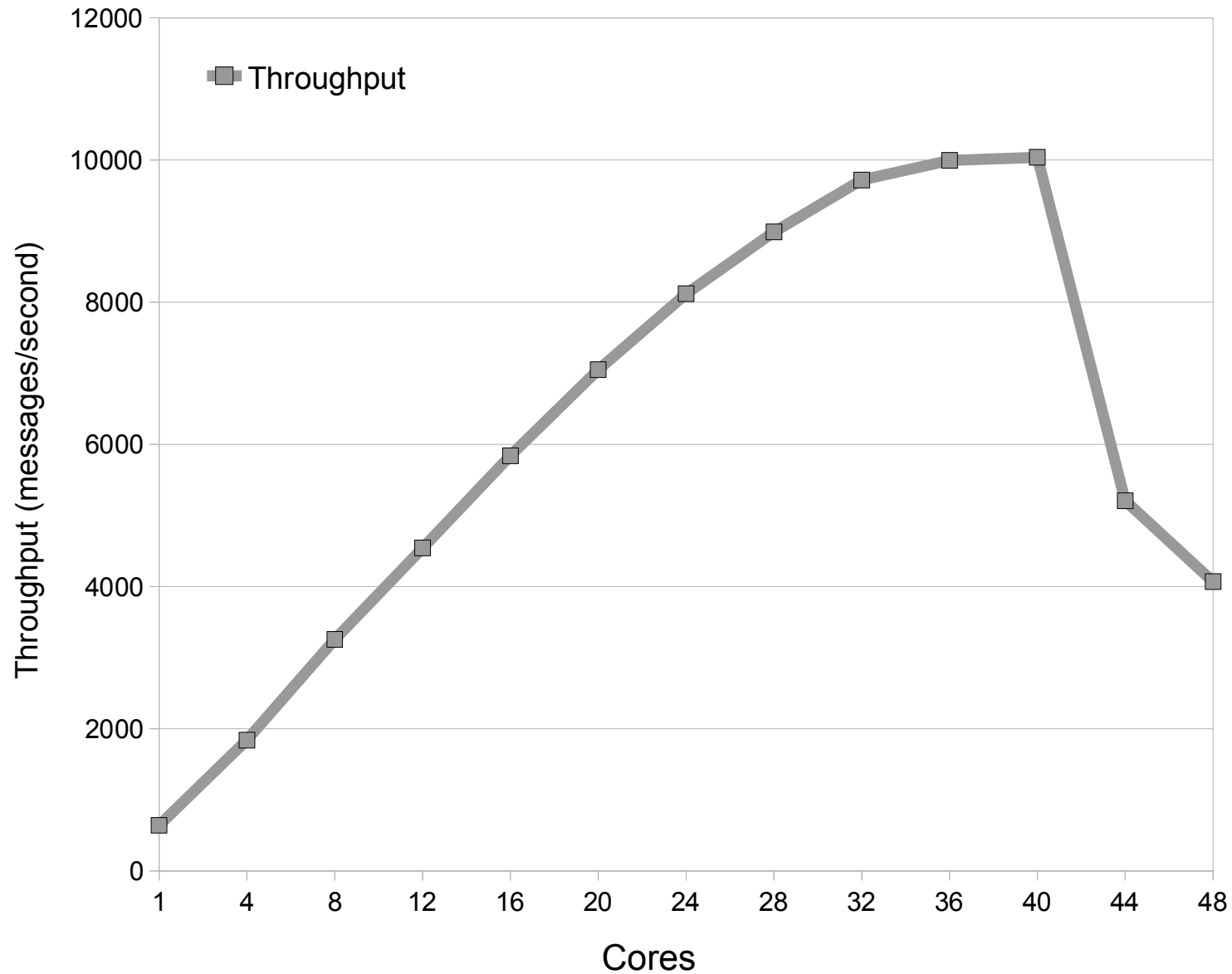


Poor scaling on stock Linux kernel

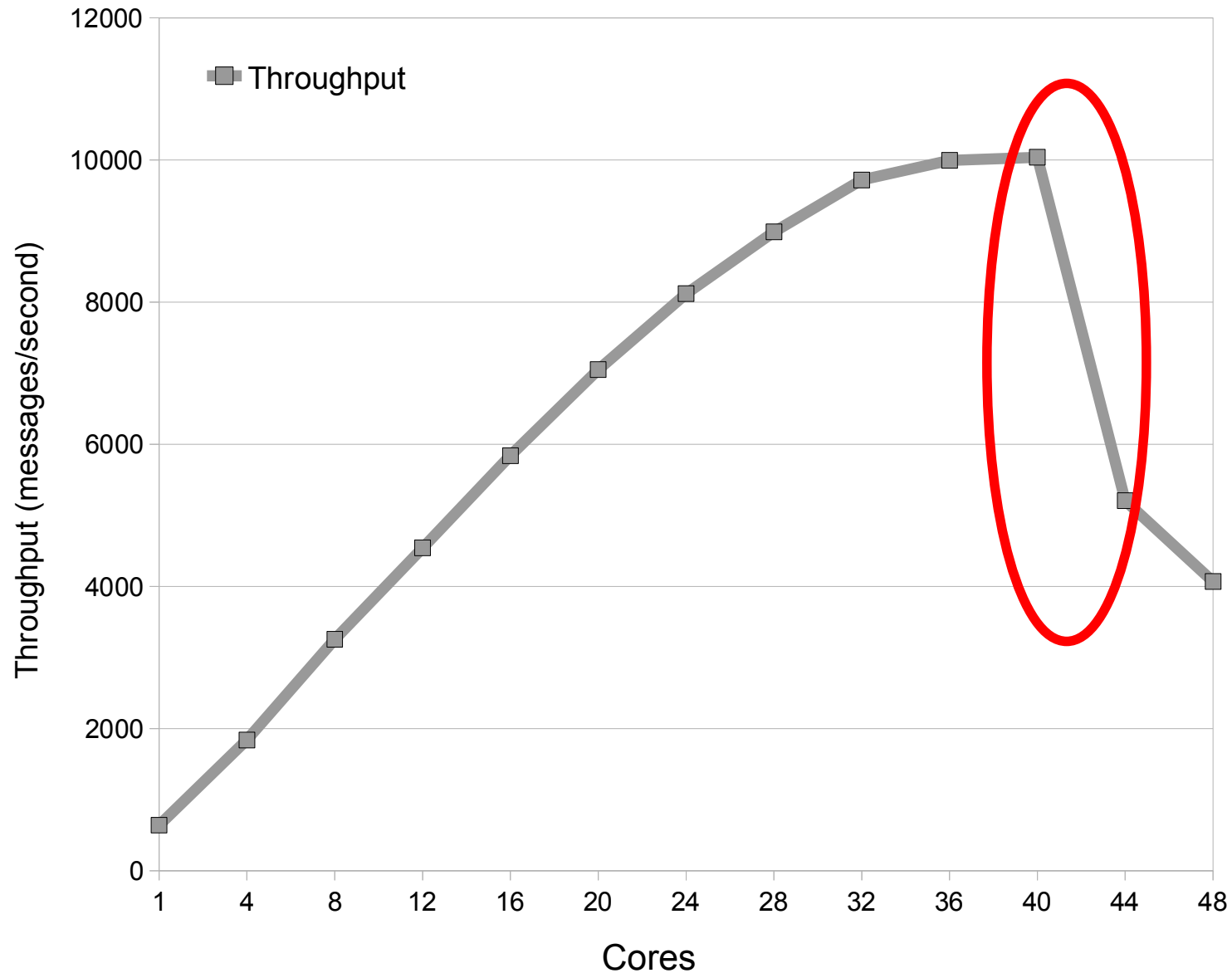


Y-axis: (throughput with 48 cores) / (throughput with one core)

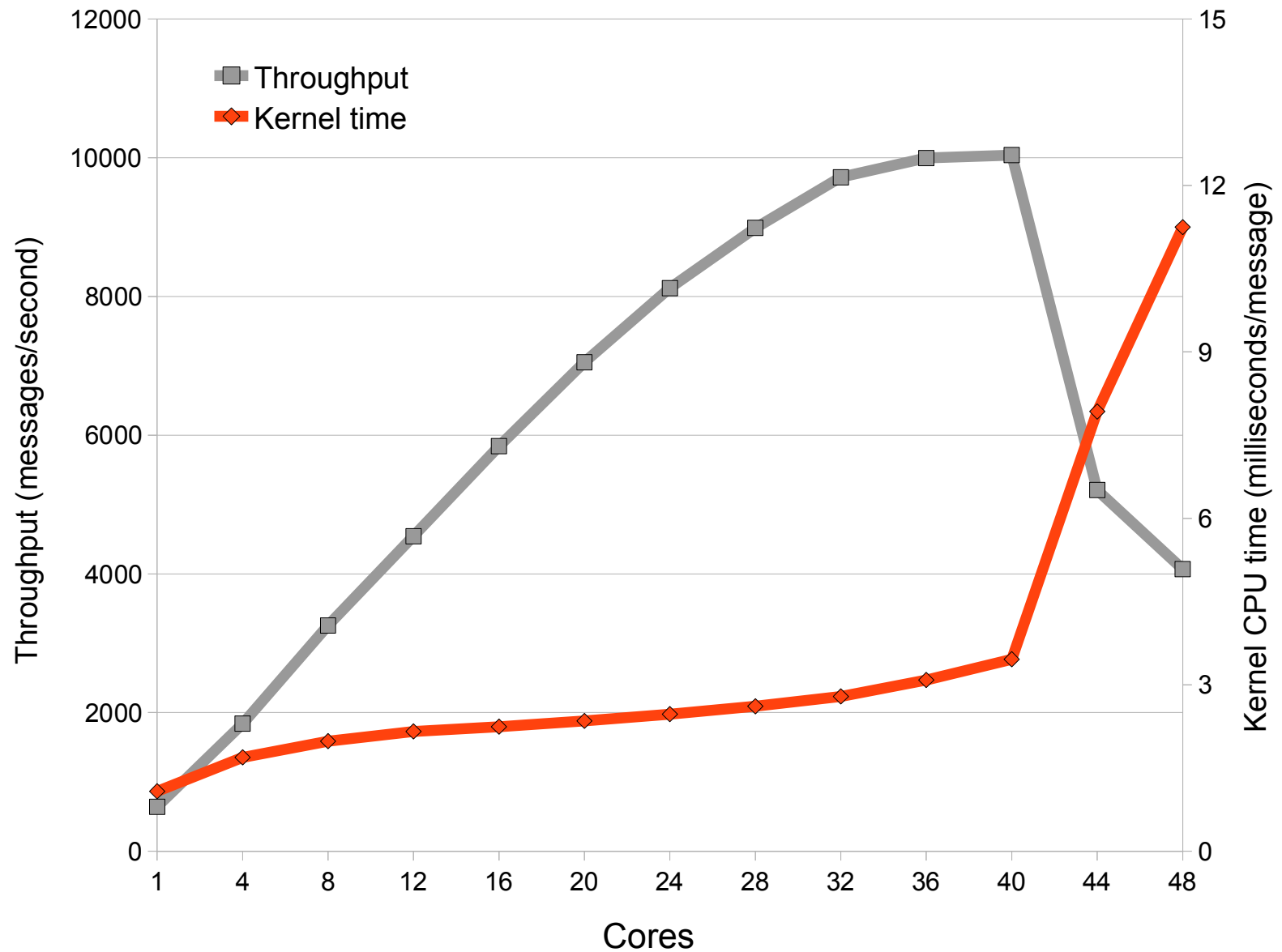
Exim on stock Linux: collapse



Exim on stock Linux: collapse



Exim on stock Linux: collapse



Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Oprofile shows an obvious problem

40 cores:
10000 msg/sec

samples	%	app name	symbol name
2616	7.3522	vmlinux	radix_tree_lookup_slot
2329	6.5456	vmlinux	unmap_vmas
2197	6.1746	vmlinux	filemap_fault
1488	4.1820	vmlinux	__do_fault
1348	3.7885	vmlinux	copy_page_c
1182	3.3220	vmlinux	unlock_page
966	2.7149	vmlinux	page_fault

48 cores:
4000 msg/sec

samples	%	app name	symbol name
13515	34.8657	vmlinux	lookup_mnt
2002	5.1647	vmlinux	radix_tree_lookup_slot
1661	4.2850	vmlinux	filemap_fault
1497	3.8619	vmlinux	unmap_vmas
1026	2.6469	vmlinux	__do_fault
914	2.3579	vmlinux	atomic_dec
896	2.3115	vmlinux	unlock_page

Bottleneck: reading mount table

- `sys_open` eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

Bottleneck: reading mount table

- `sys_open` eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

Bottleneck: reading mount table

- `sys_open` eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

← Critical section is short. Why does it cause a scalability bottleneck?

Bottleneck: reading mount table

- `sys_open` eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

← Critical section is short. Why does it cause a scalability bottleneck?

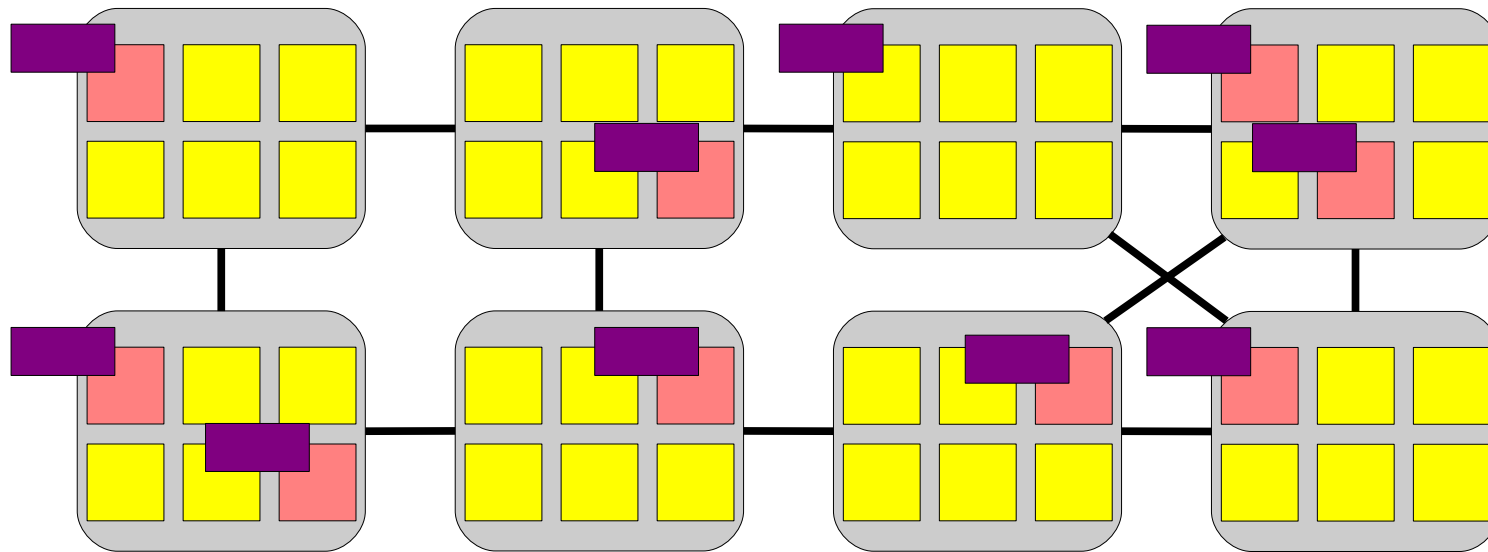
- `spin_lock` and `spin_unlock` use many more cycles than the critical section

Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

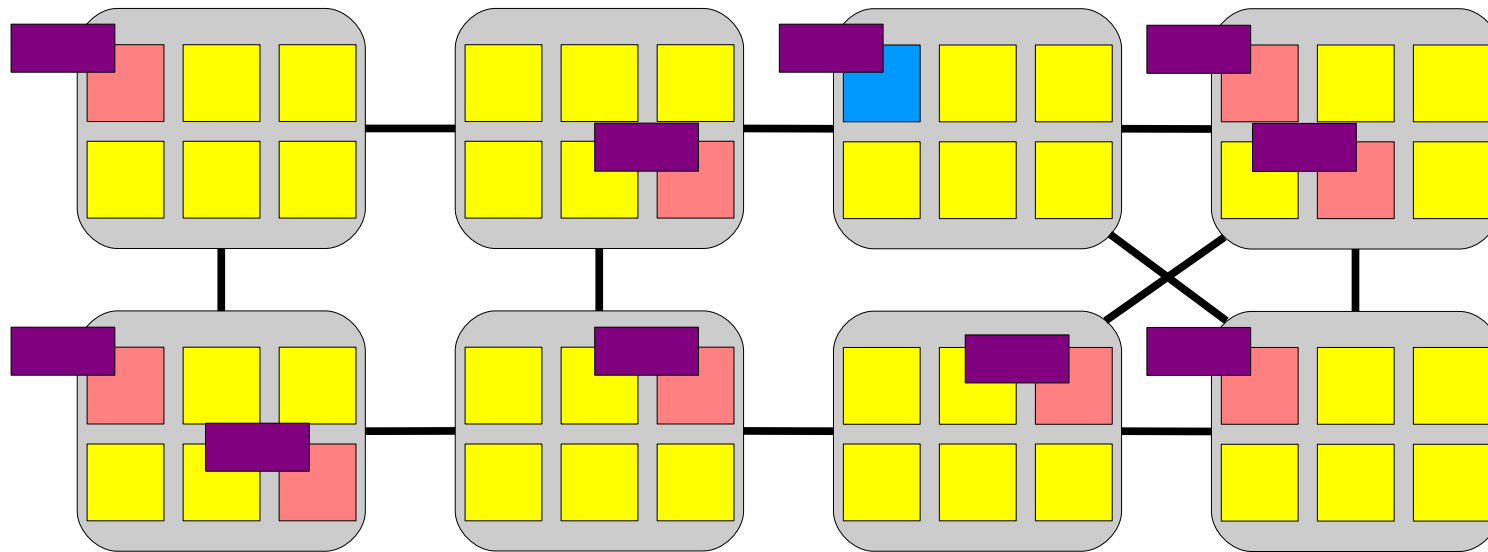


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

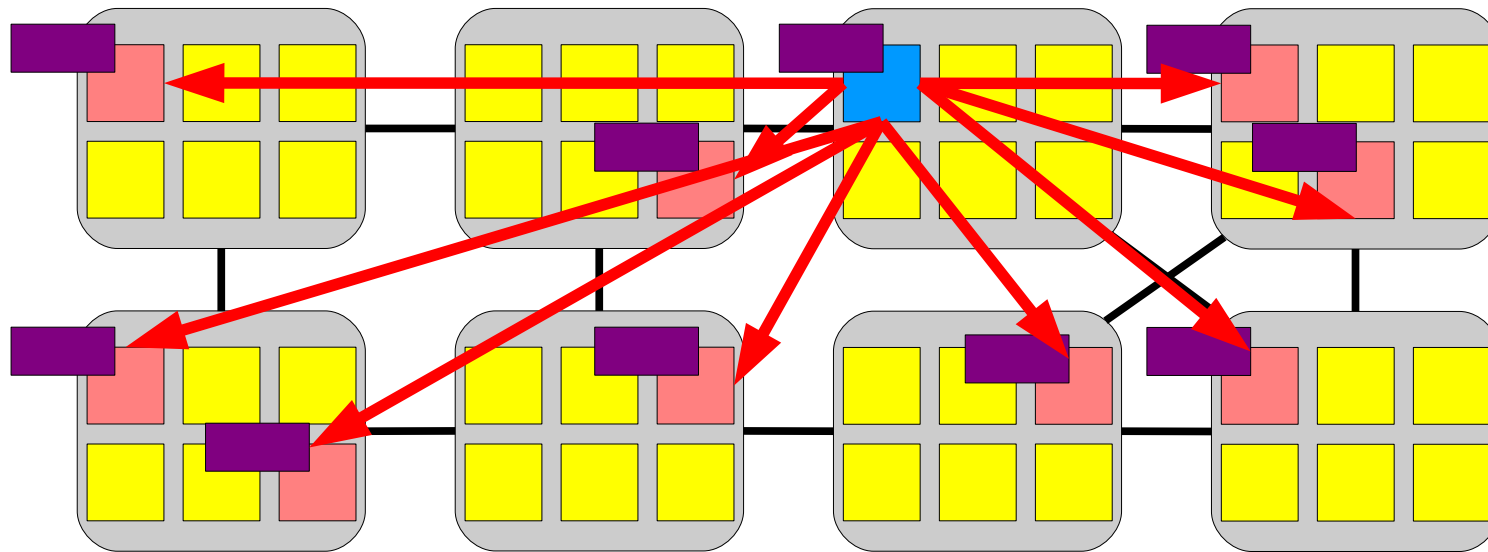


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

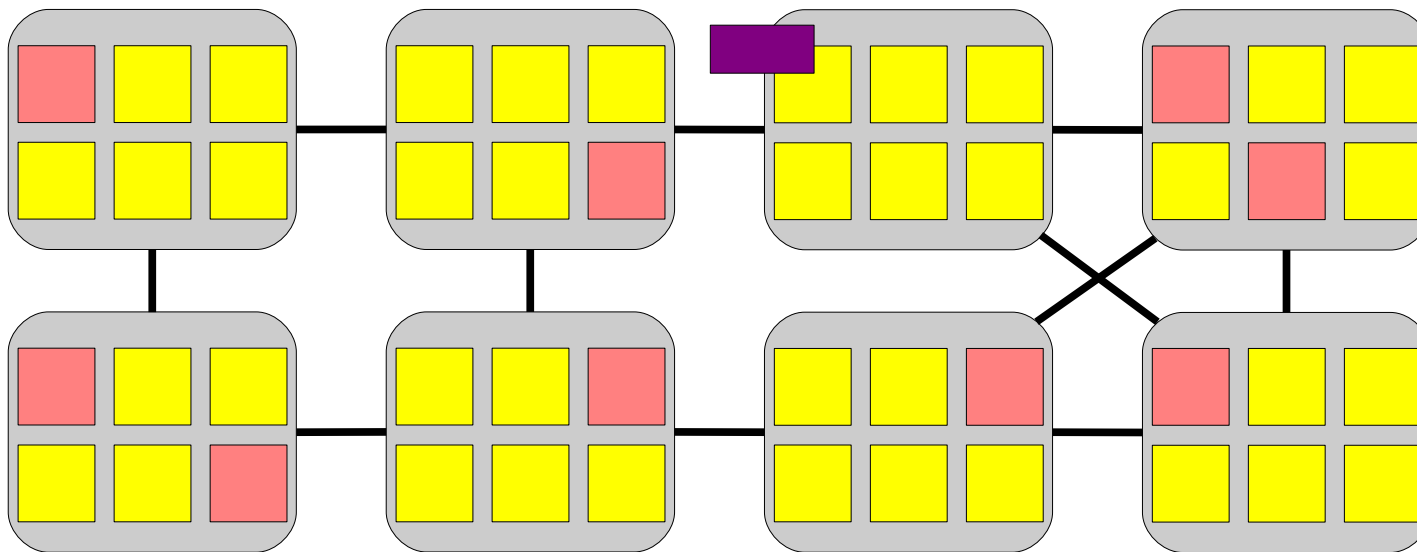


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

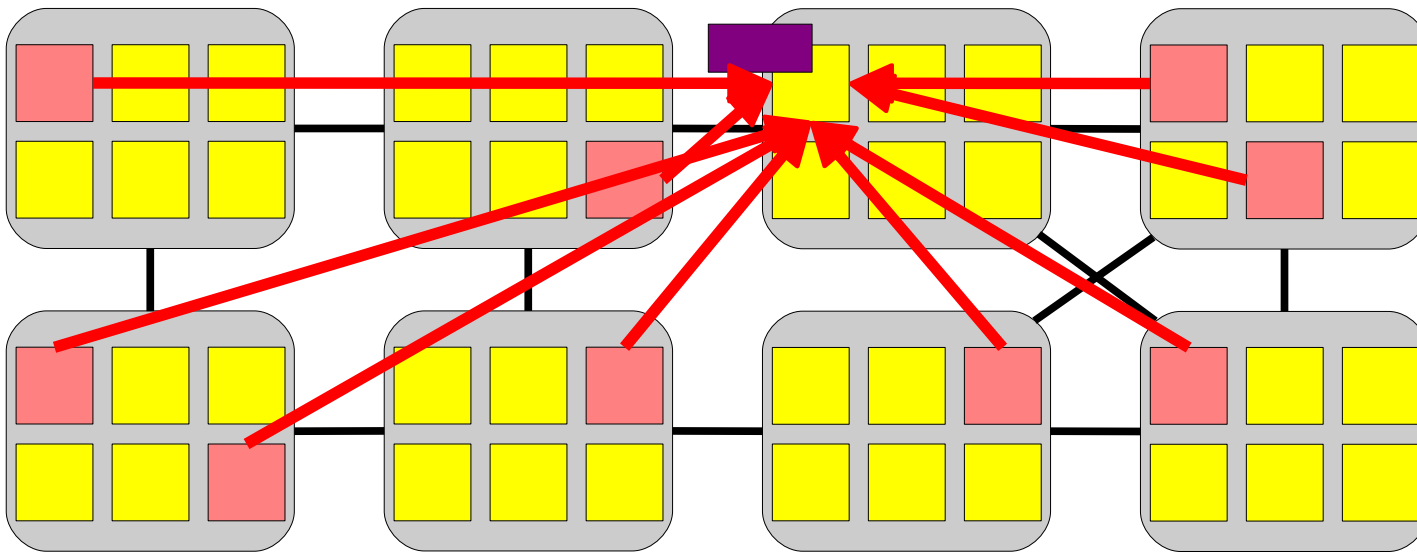


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



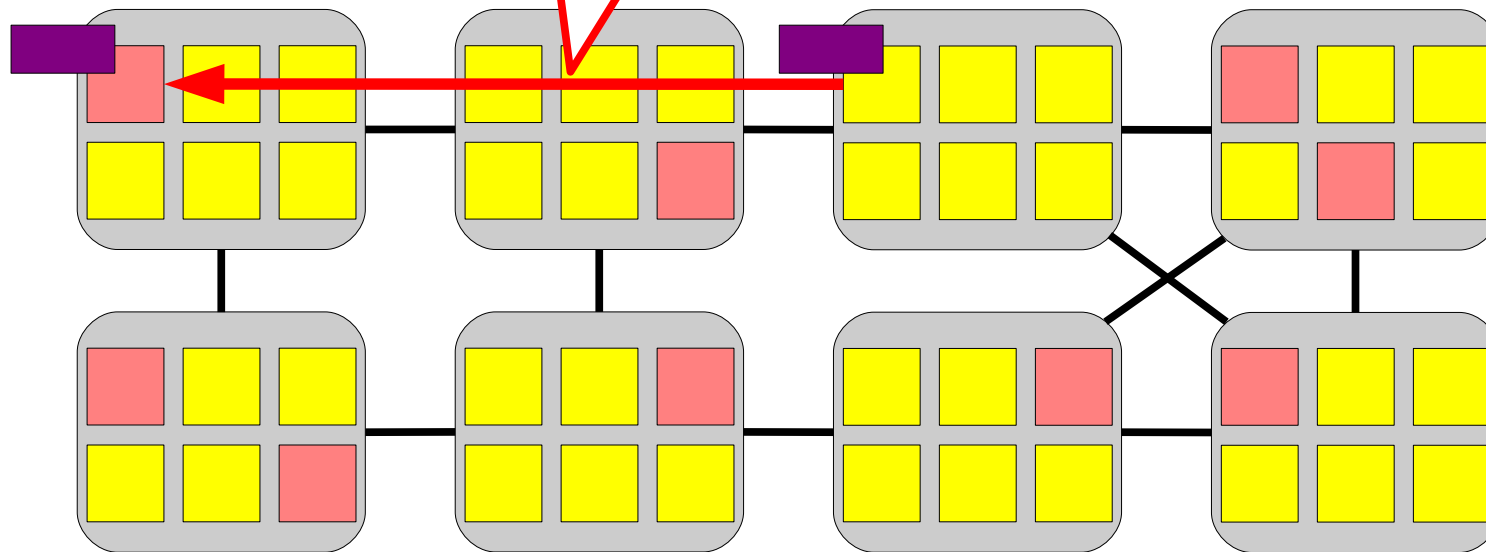
Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

500 – 4000 cycles!!

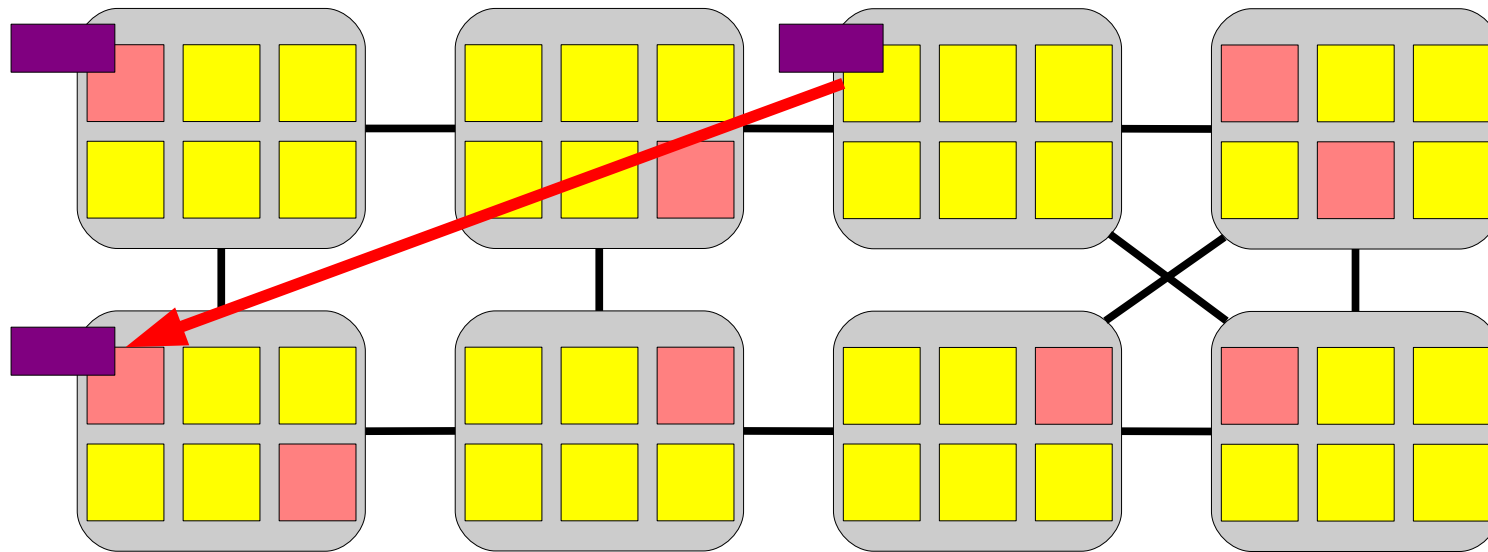


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

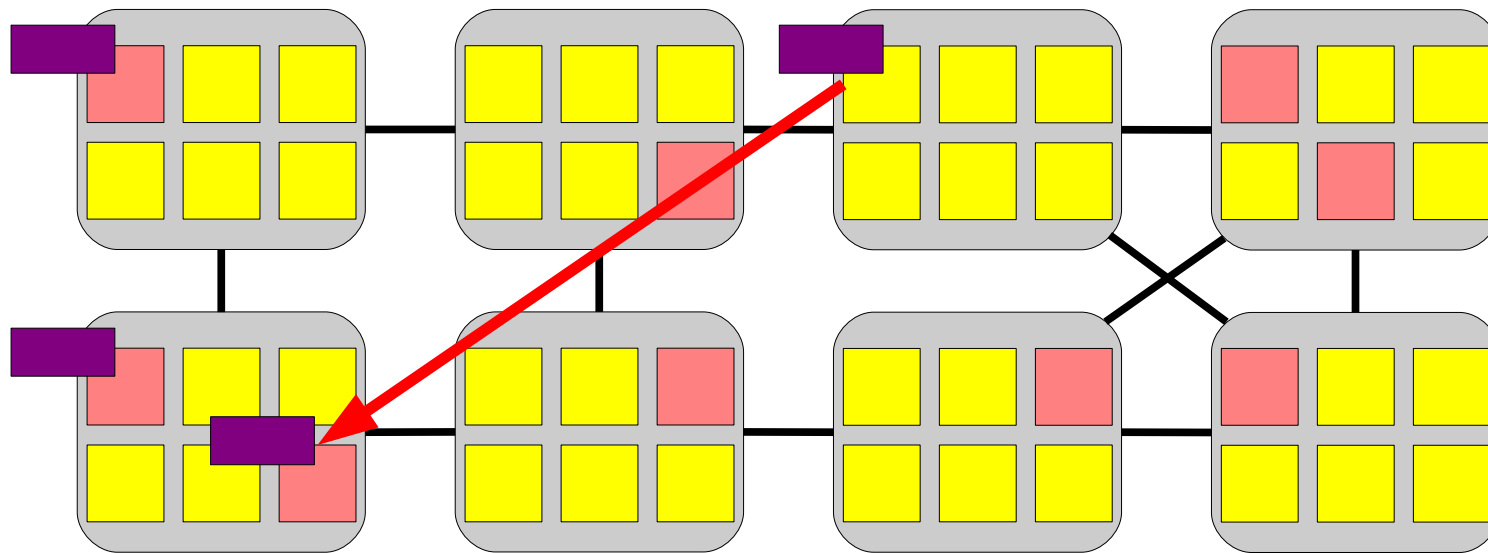


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

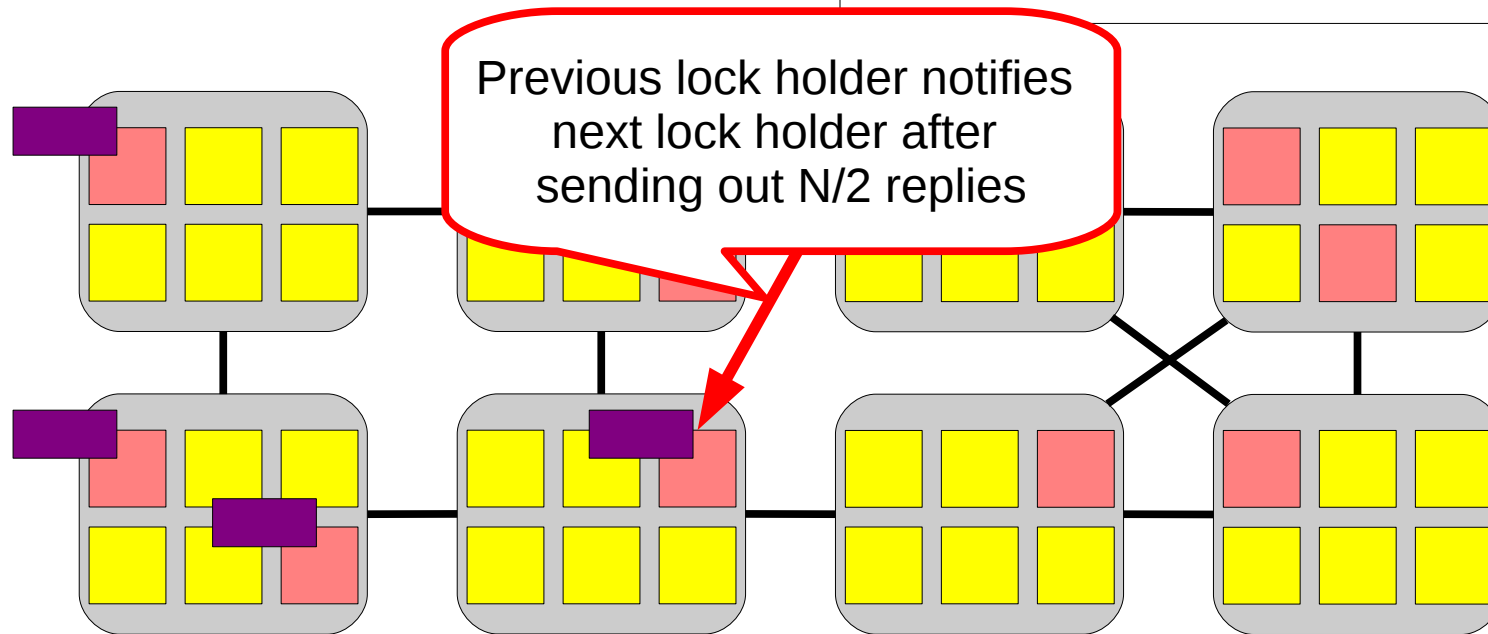


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

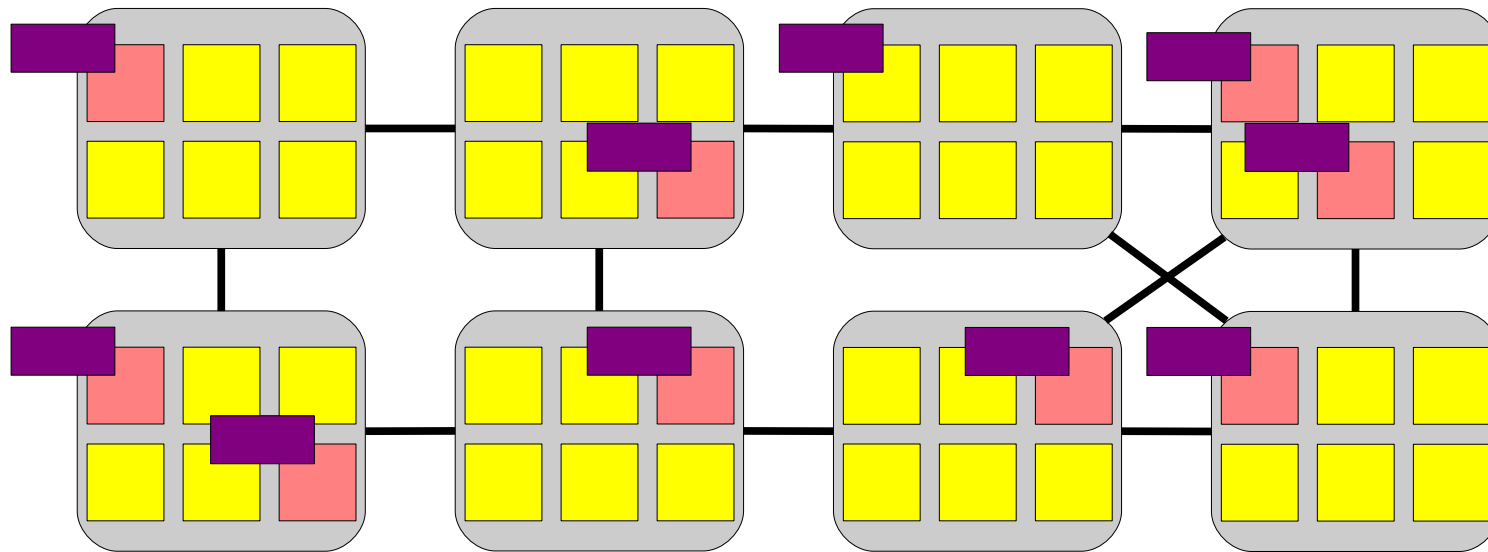


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

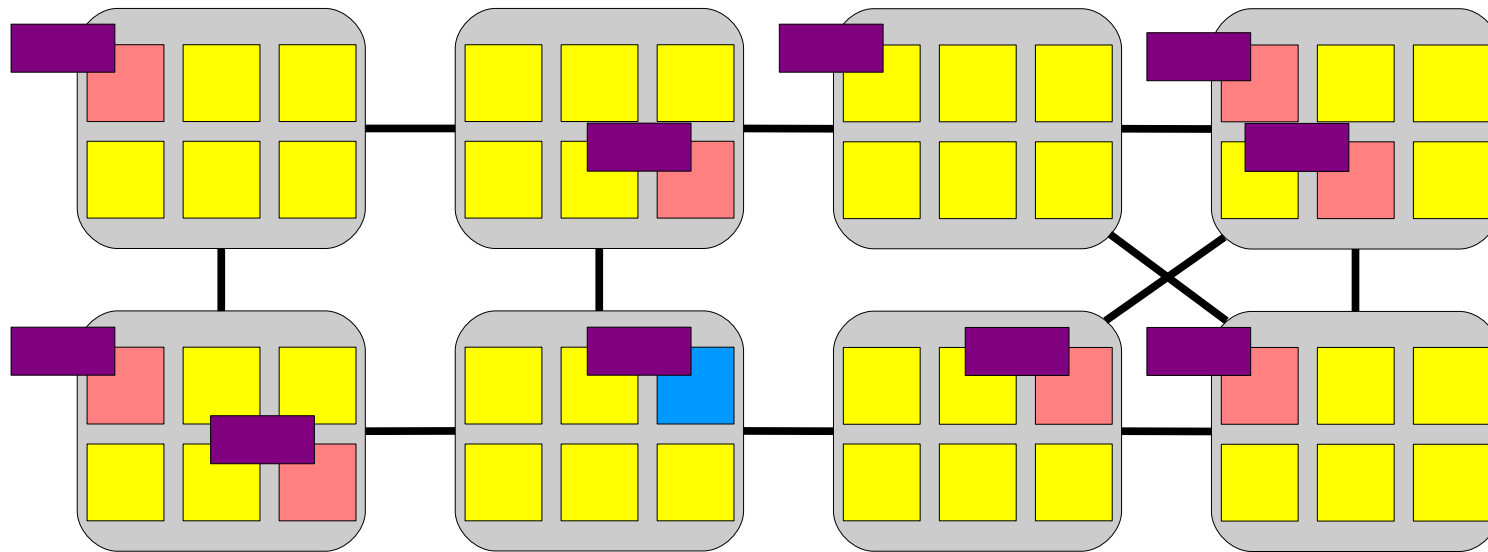


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

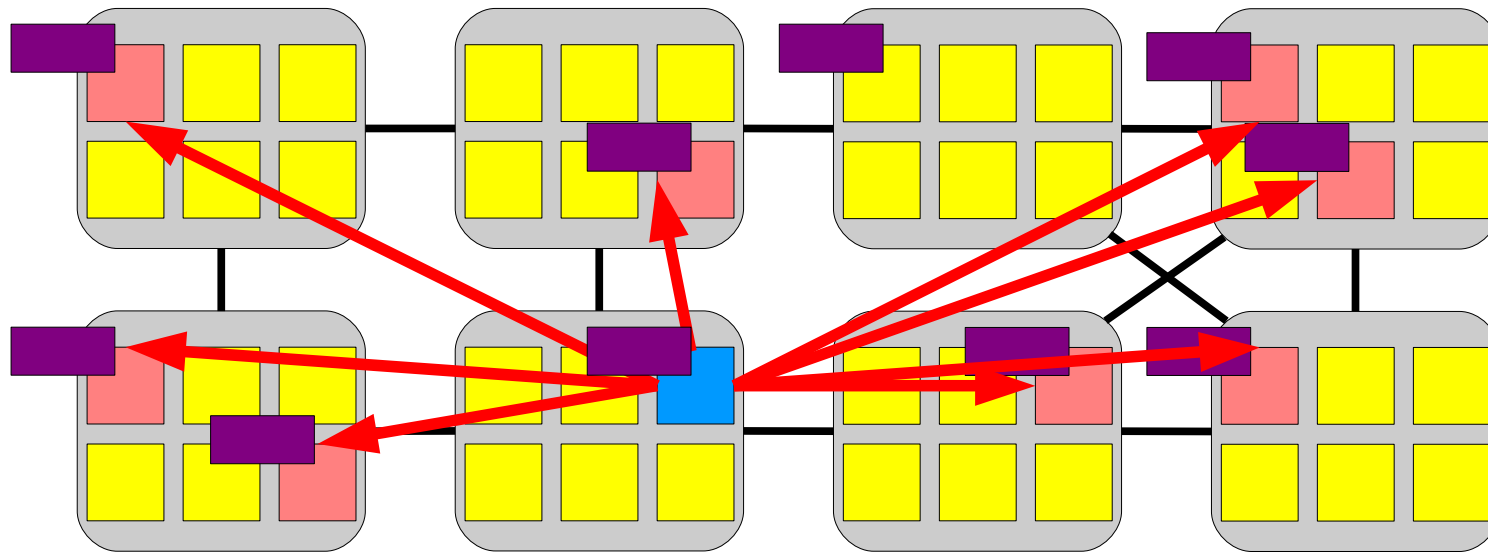


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

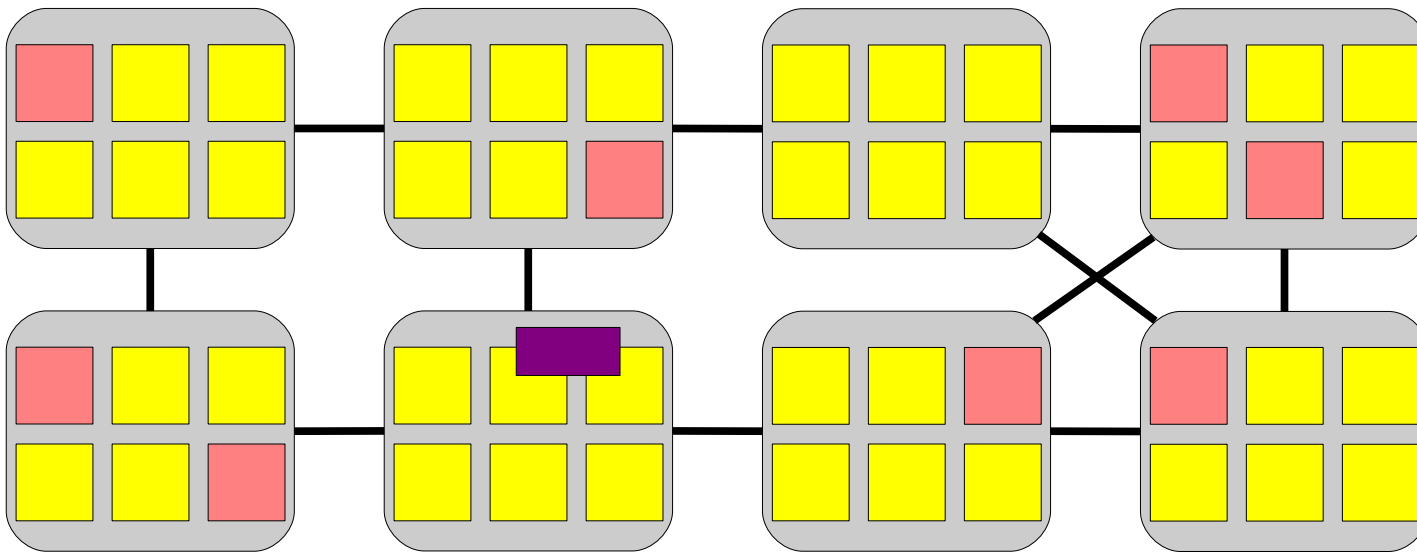


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

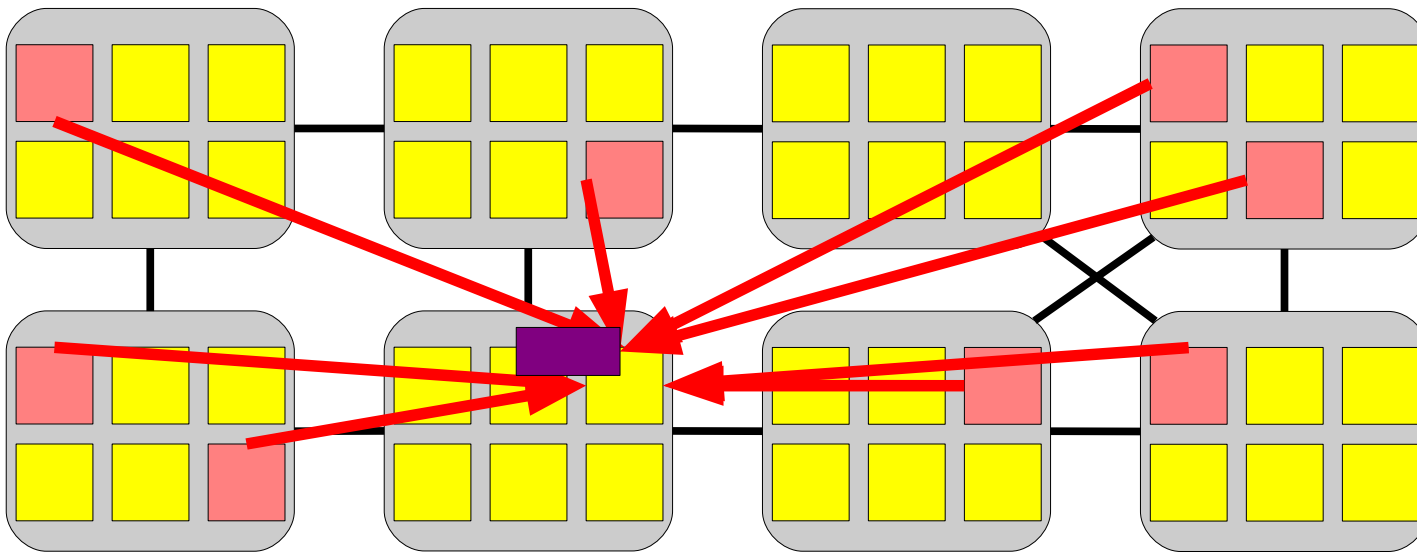


Scalability collapse caused by non-scalable locks [Anderson 90]

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



Bottleneck: reading mount table

- `sys_open` eventually calls:

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    return mnt;
}
```

- Well known problem, many solutions
 - Use scalable locks [MCS 91]
 - Use message passing [Baumann 09]
 - Avoid locks in the common case

Solution: per-core mount caches

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

Solution: per-core mount caches

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

Solution: per-core mount caches

- Observation: mount table is rarely modified

```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

- Common case: cores access per-core tables
- Modify mount table: invalidate per-core tables

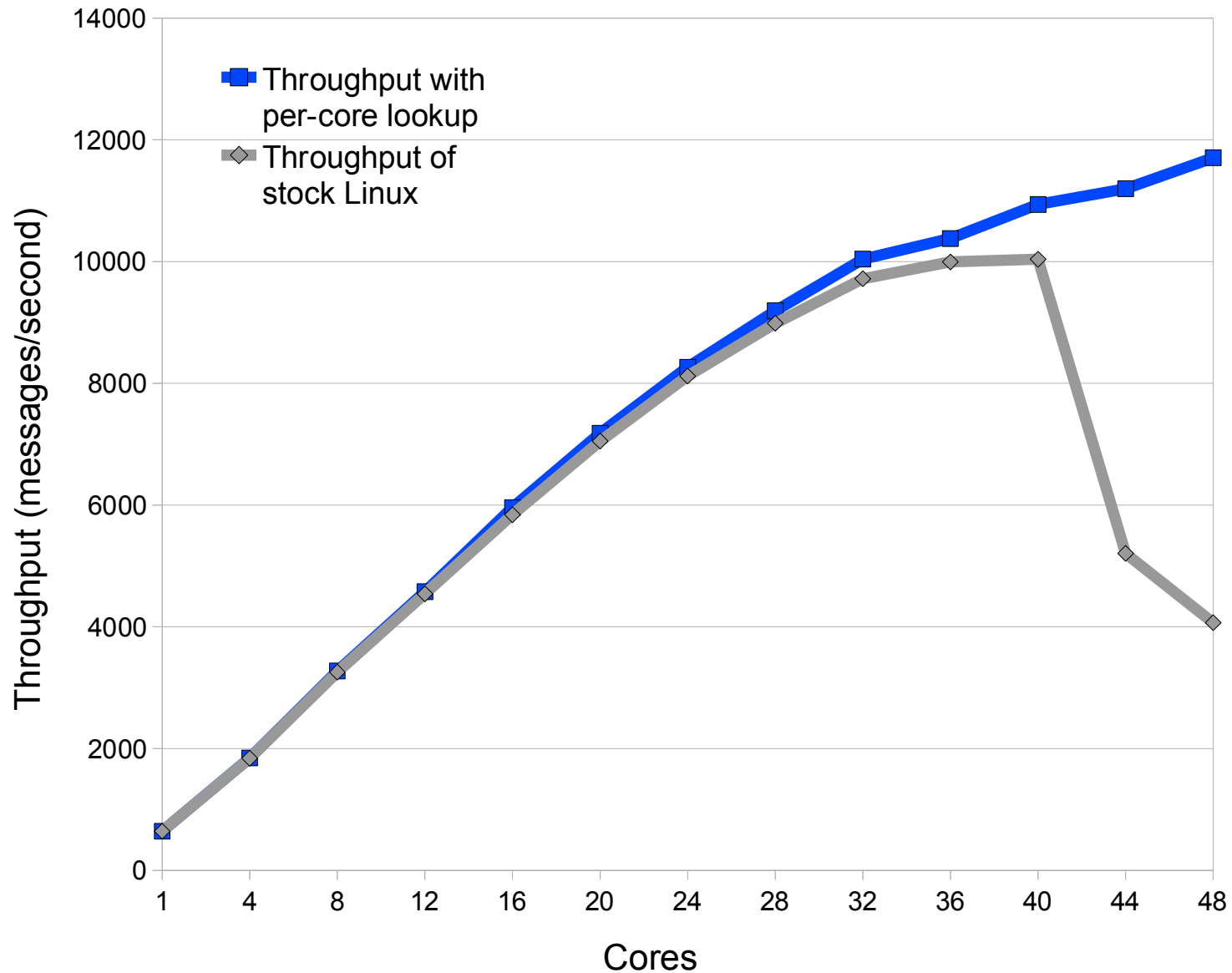
Solution: per-core mount caches

- Observation: mount table is rarely modified

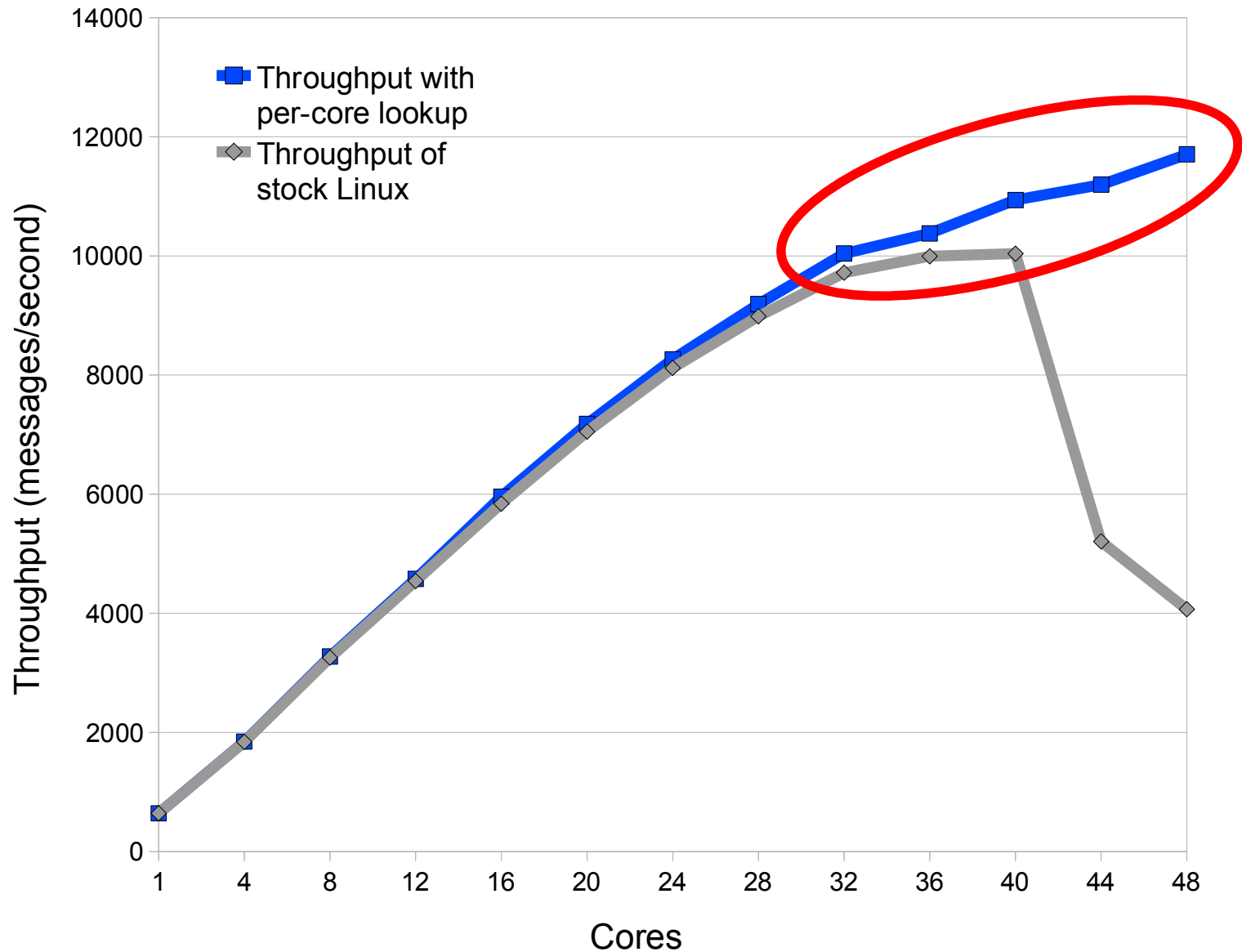
```
struct vfsmount *lookup_mnt(struct path *path)
{
    struct vfsmount *mnt;
    if ((mnt = hash_get(percore_mnts[cpu()], path)))
        return mnt;
    spin_lock(&vfsmount_lock);
    mnt = hash_get(mnts, path);
    spin_unlock(&vfsmount_lock);
    hash_put(percore_mnts[cpu()], path, mnt);
    return mnt;
}
```

- Common case: cores access per-core tables
- Modify mount table: invalidate per-core tables

Per-core lookup: scalability is better



Per-core lookup: scalability is better



No obvious bottlenecks

32 cores: 10041 msg/sec	samples	%	app name	symbol name
	3319	5.4462	vmlinux	radix_tree_lookup_slot
	3119	5.2462	vmlinux	unmap_vmas
	1966	3.3069	vmlinux	filemap_fault
	1950	3.2800	vmlinux	page_fault
	1627	2.7367	vmlinux	unlock_page
	1626	2.7350	vmlinux	clear_page_c
48 cores: 11705 msg/sec	1578	2.6542	vmlinux	kmem_cache_free
	samples	%	app name	symbol name
	4207	5.3145	vmlinux	radix_tree_lookup_slot
	4191	5.2943	vmlinux	unmap_vmas
	2632	3.3249	vmlinux	page_fault
	2525	3.1897	vmlinux	filemap_fault
	2210	2.7918	vmlinux	clear_page_c
	2131	2.6920	vmlinux	kmem_cache_free
	2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	unmap_vmas
2632	3.3249	vmlinux	page_fault
2525	3.1897	vmlinux	filemap_fault
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	unmap_vmas
2632	3.3249	vmlinux	page_fault
2525	3.1897	vmlinux	filemap_fault
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

- Functions execute more slowly on 48 cores

No obvious bottlenecks

32 cores:
10041 msg/sec

samples	%	app name	symbol name
3319	5.4462	vmlinux	radix_tree_lookup_slot
3119	5.2462	vmlinux	unmap_vmas
1966	3.3069	vmlinux	filemap_fault
1950	3.2800	vmlinux	page_fault
1627	2.7367	vmlinux	unlock_page
1626	2.7350	vmlinux	clear_page_c
1578	2.6542	vmlinux	kmem_cache_free

48 cores:
11705 msg/sec

samples	%	app name	symbol name
4207	5.3145	vmlinux	radix_tree_lookup_slot
4191	5.2943	vmlinux	
2632	3.3249	vmlinux	
2525	3.1897	vmlinux	
2210	2.7918	vmlinux	clear_page_c
2131	2.6920	vmlinux	kmem_cache_free
2000	2.5265	vmlinux	dput

dput is causing other
functions to slow down

- Functions execute more slowly on 48 cores

Bottleneck: reference counting

- Ref count indicates if kernel can free object
 - File name cache (dentry), physical pages, ...

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

Bottleneck: reference counting

- Ref count indicates if kernel can free object
 - File name cache (dentry), physical pages, ...

```
void dput(struct dentry *dentry)
```

```
{
```

```
    if (!atomic_dec_and_test(&dentry->ref))
```

```
        return;
```

```
    dentry_free(dentry);
```

```
}
```



A single atomic instruction
limits scalability?!

Bottleneck: reference counting

- Ref count indicates if kernel can free object
 - File name cache (dentry), physical pages, ...

```
void dput(struct dentry *dentry)
```

```
{
```

```
    if (!atomic_dec_and_test(&dentry->ref))
```

```
        return;
```

```
    dentry_free(dentry);
```

```
}
```



A single atomic instruction
limits scalability?!

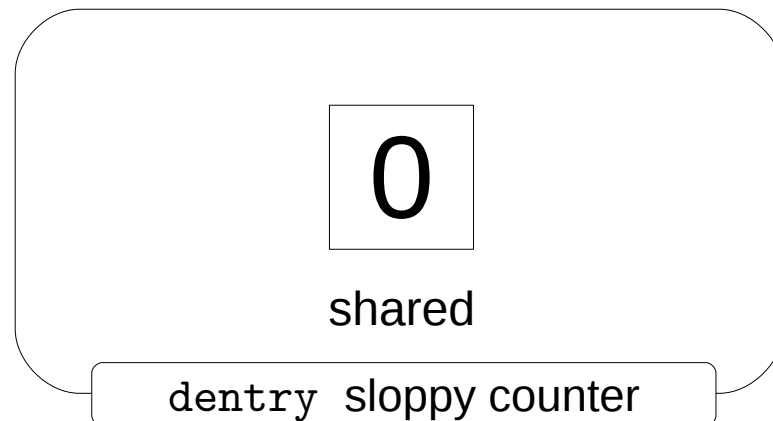
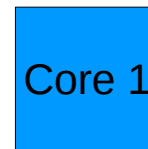
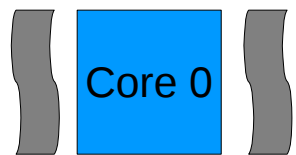
- Reading the reference count is slow
- Reading the reference count delays memory operations from other cores

Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references

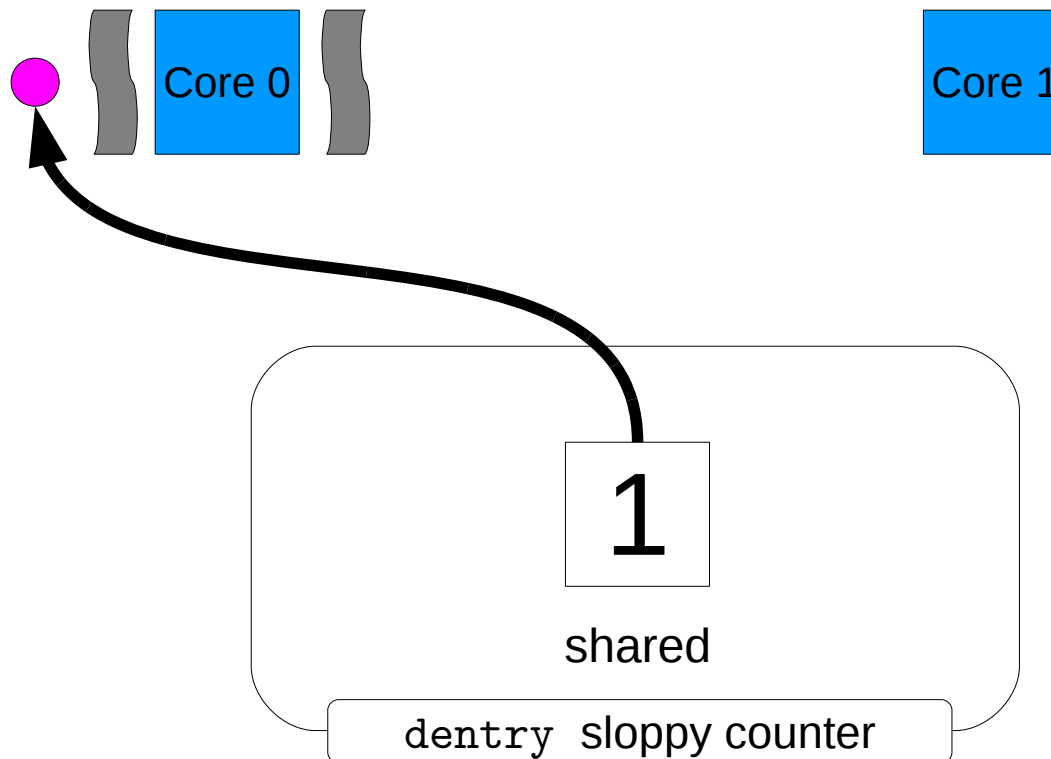
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



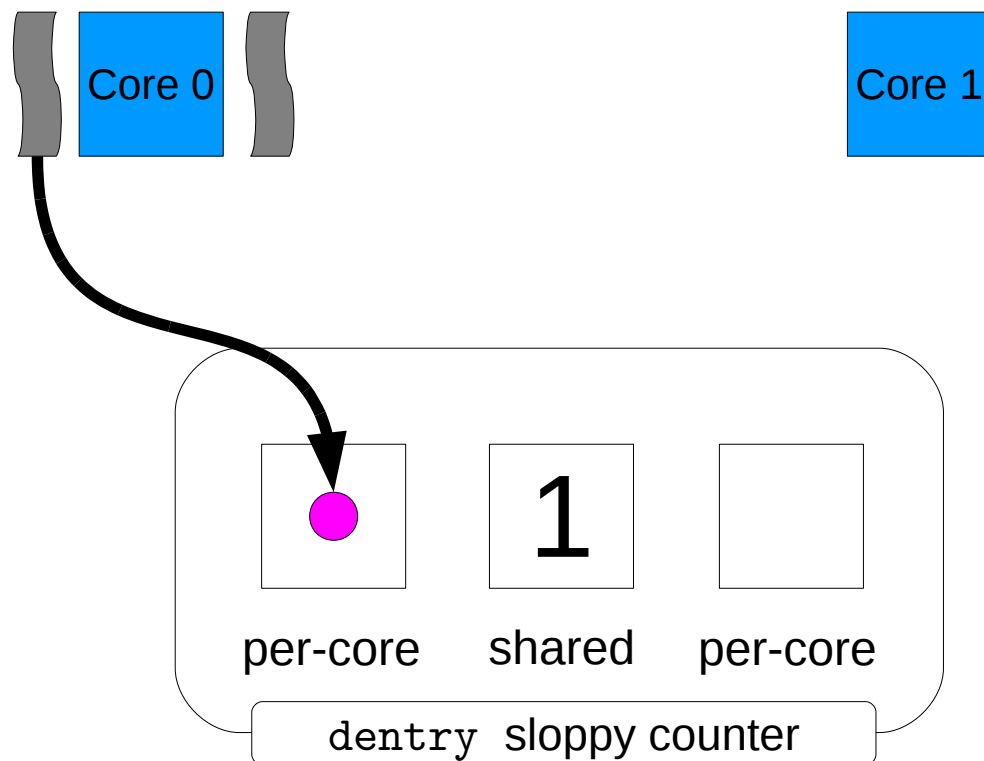
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



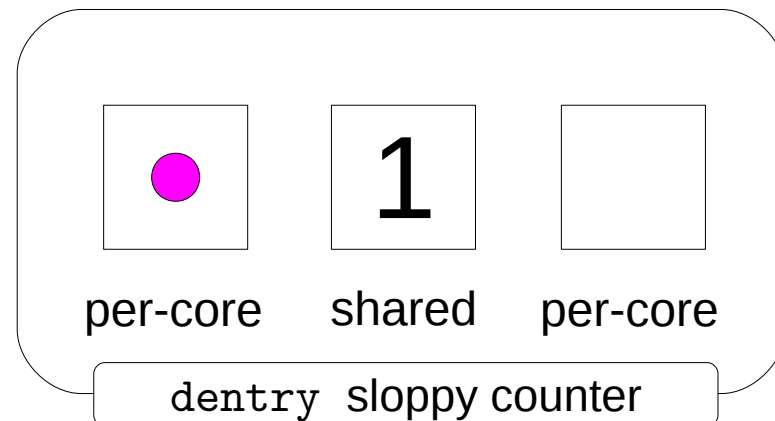
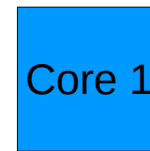
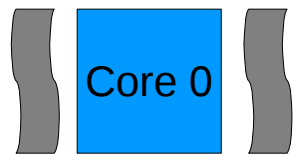
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



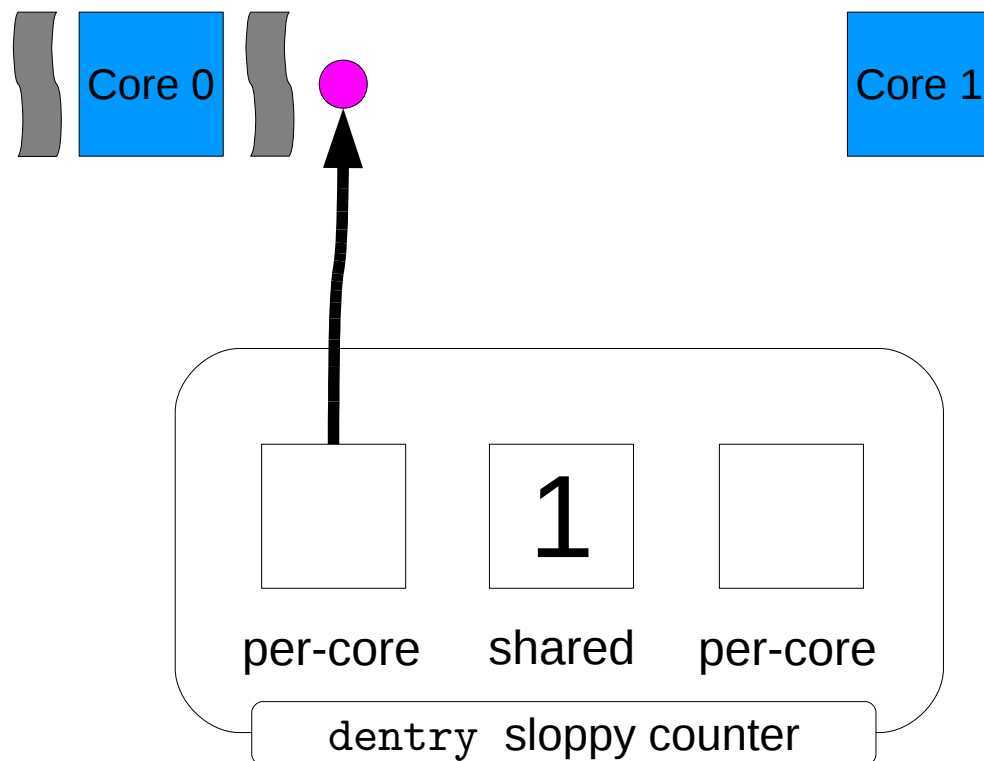
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



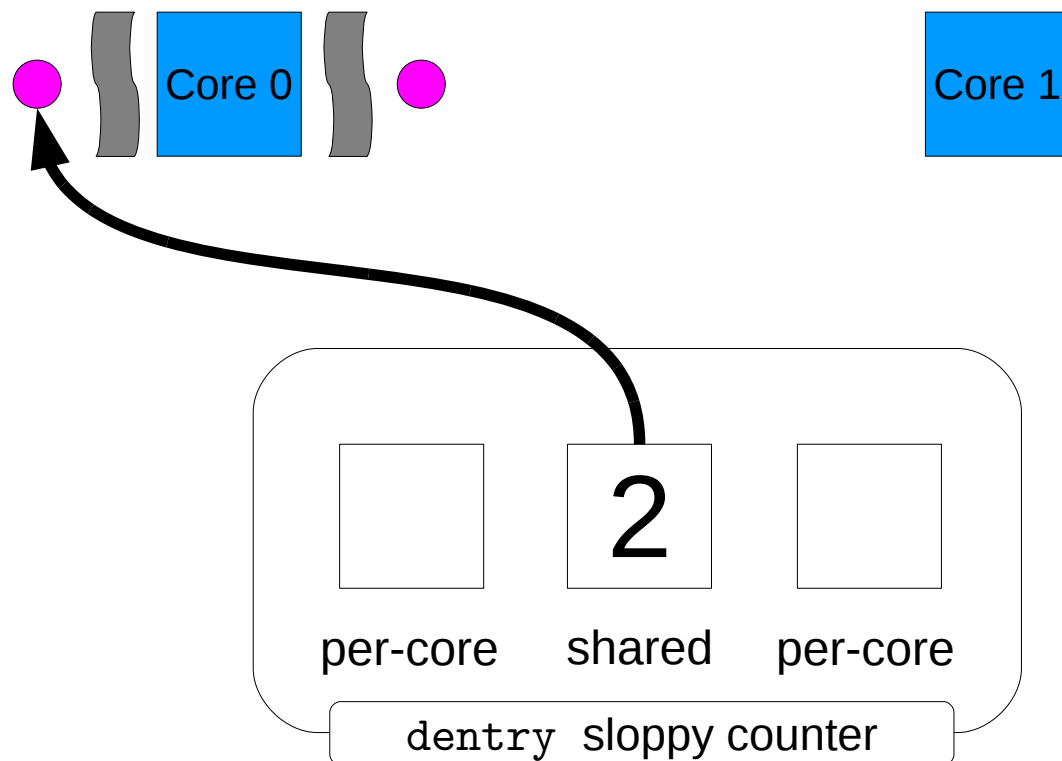
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



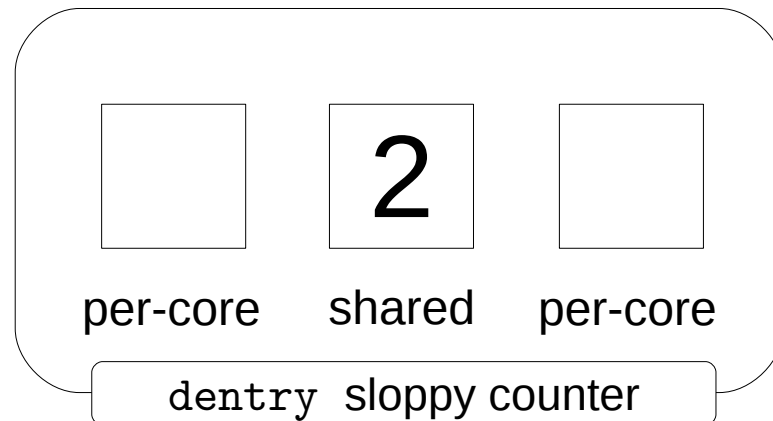
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



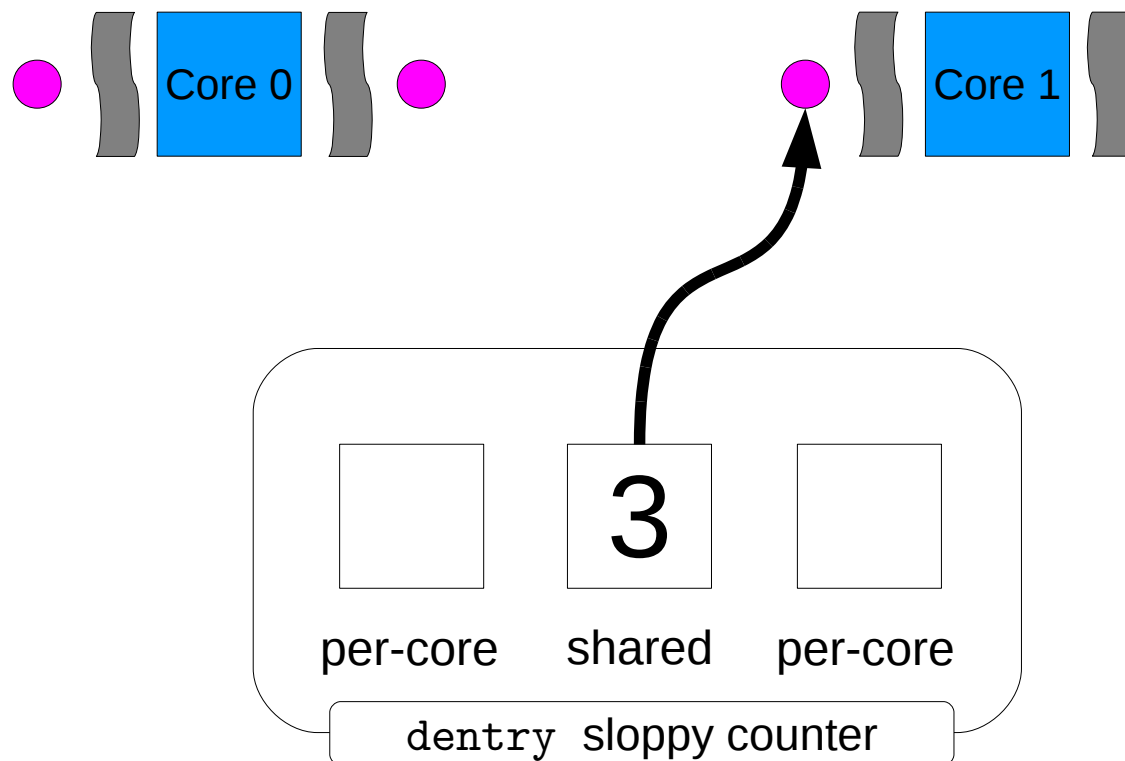
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



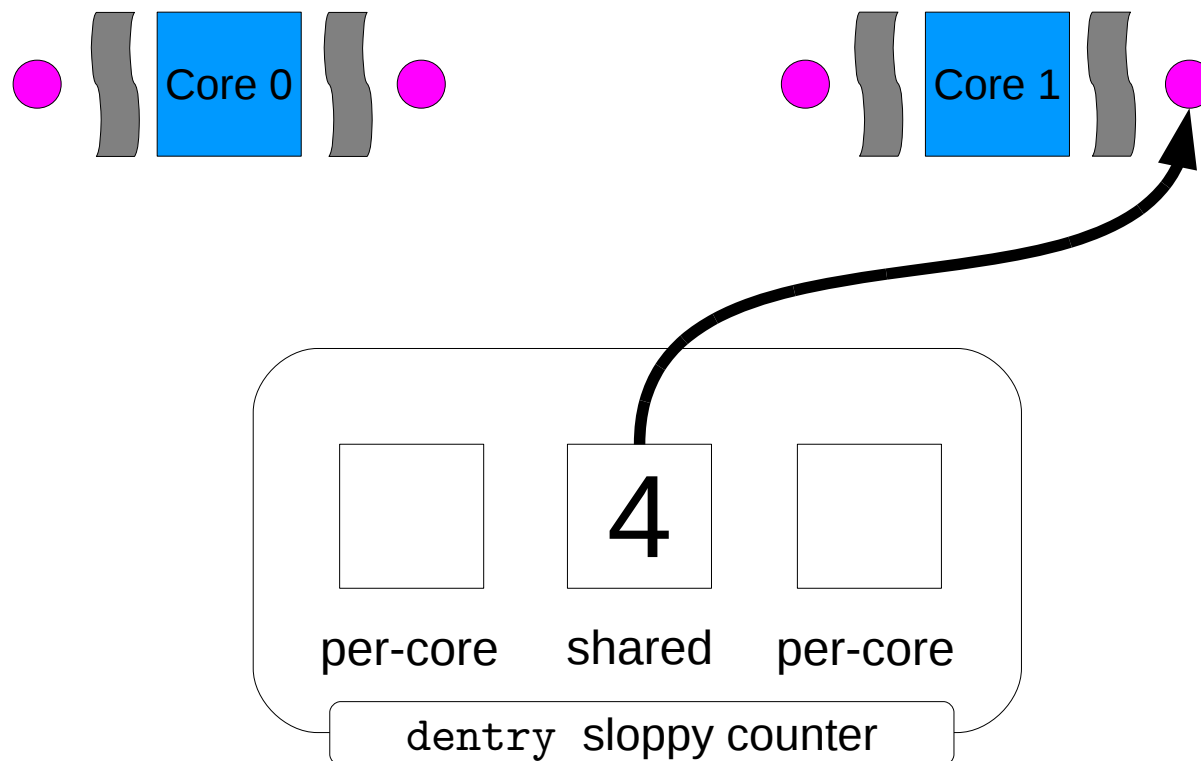
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



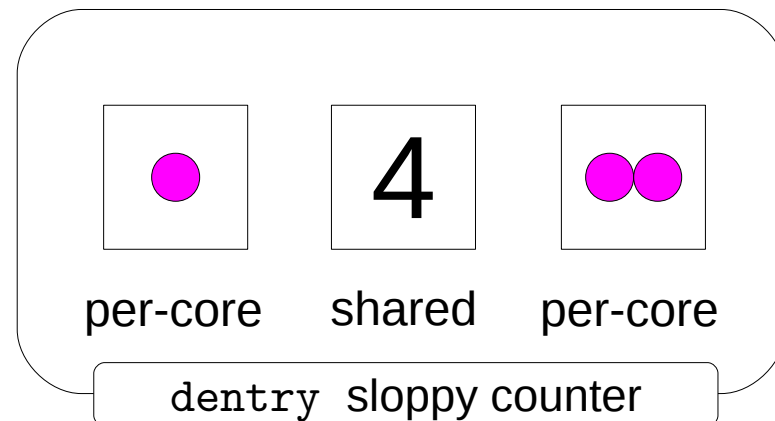
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



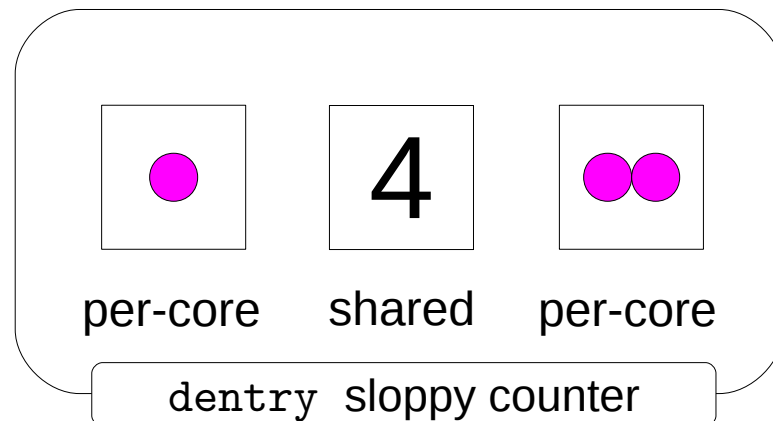
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



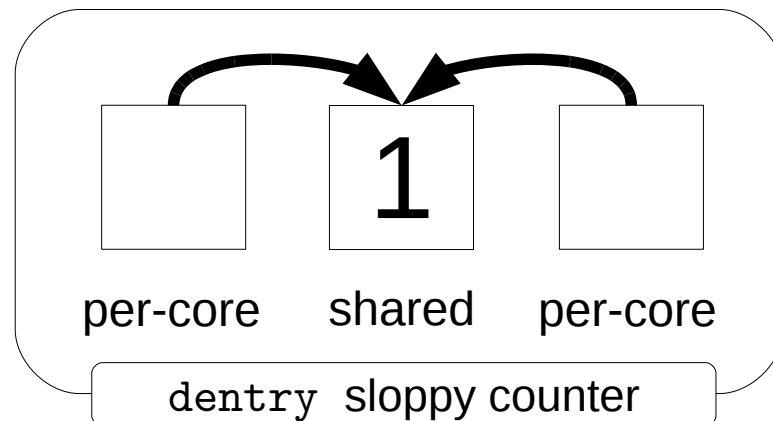
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



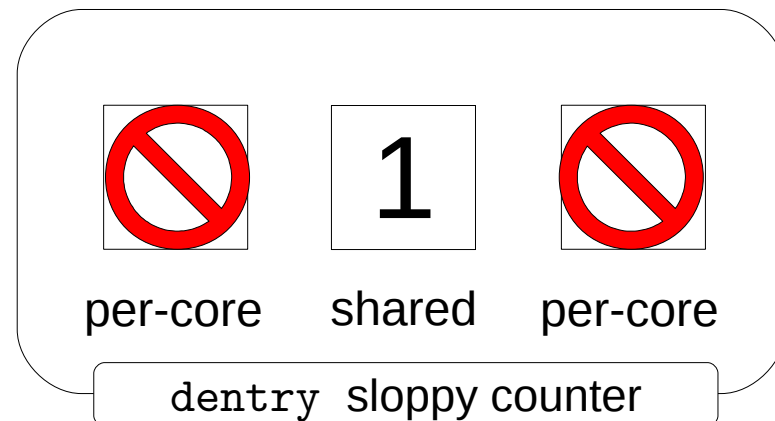
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



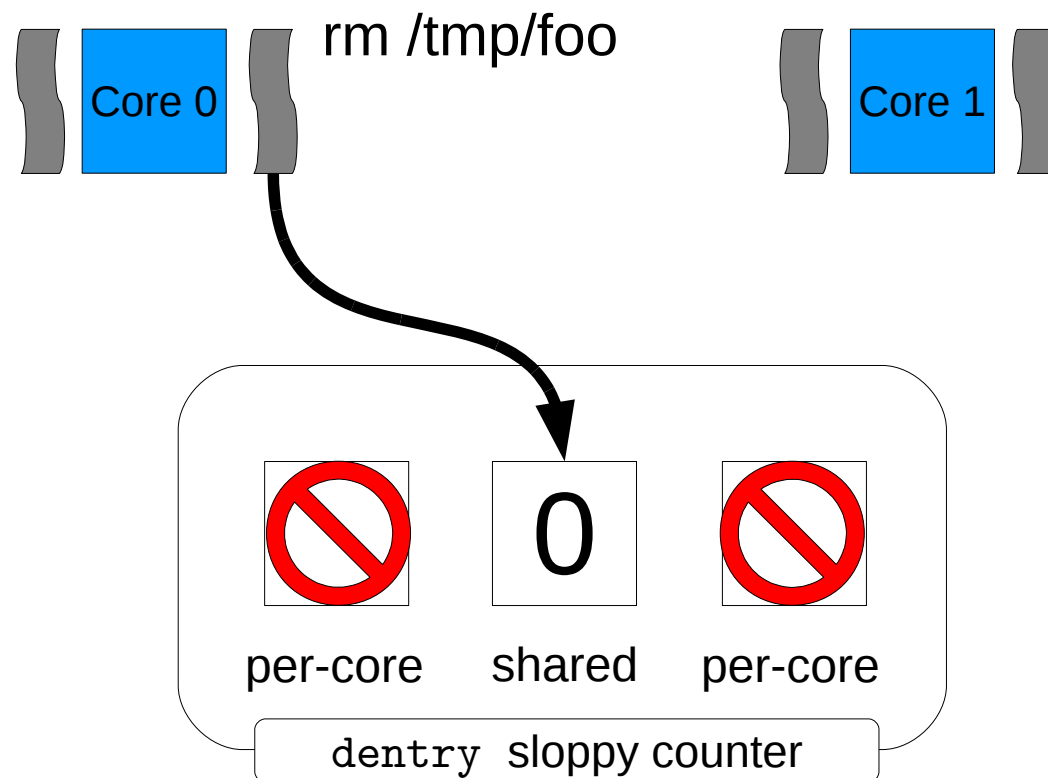
Solution: sloppy counters

- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



Solution: sloppy counters

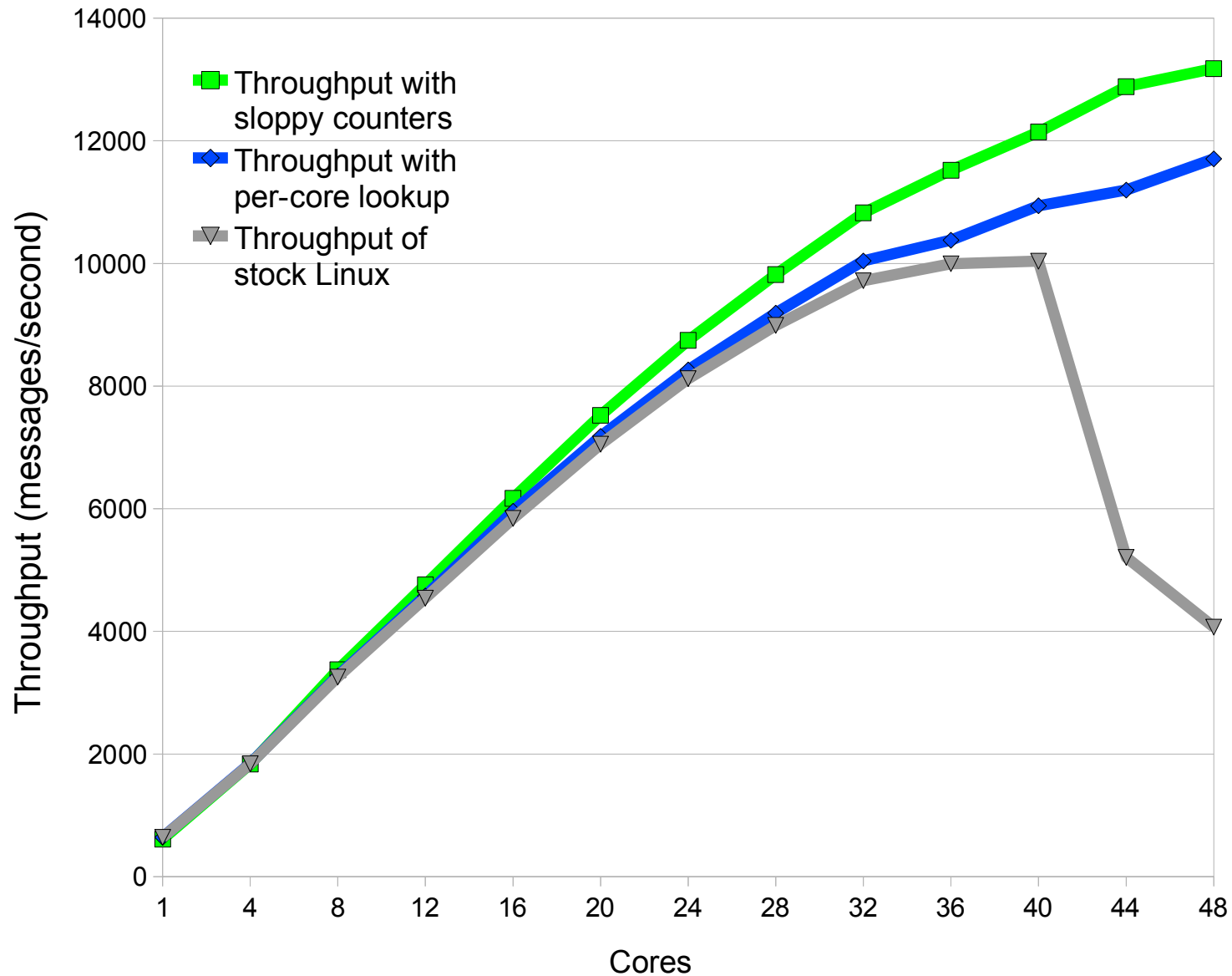
- Observation: kernel rarely needs true value of ref count
 - Each core holds a few “spare” references



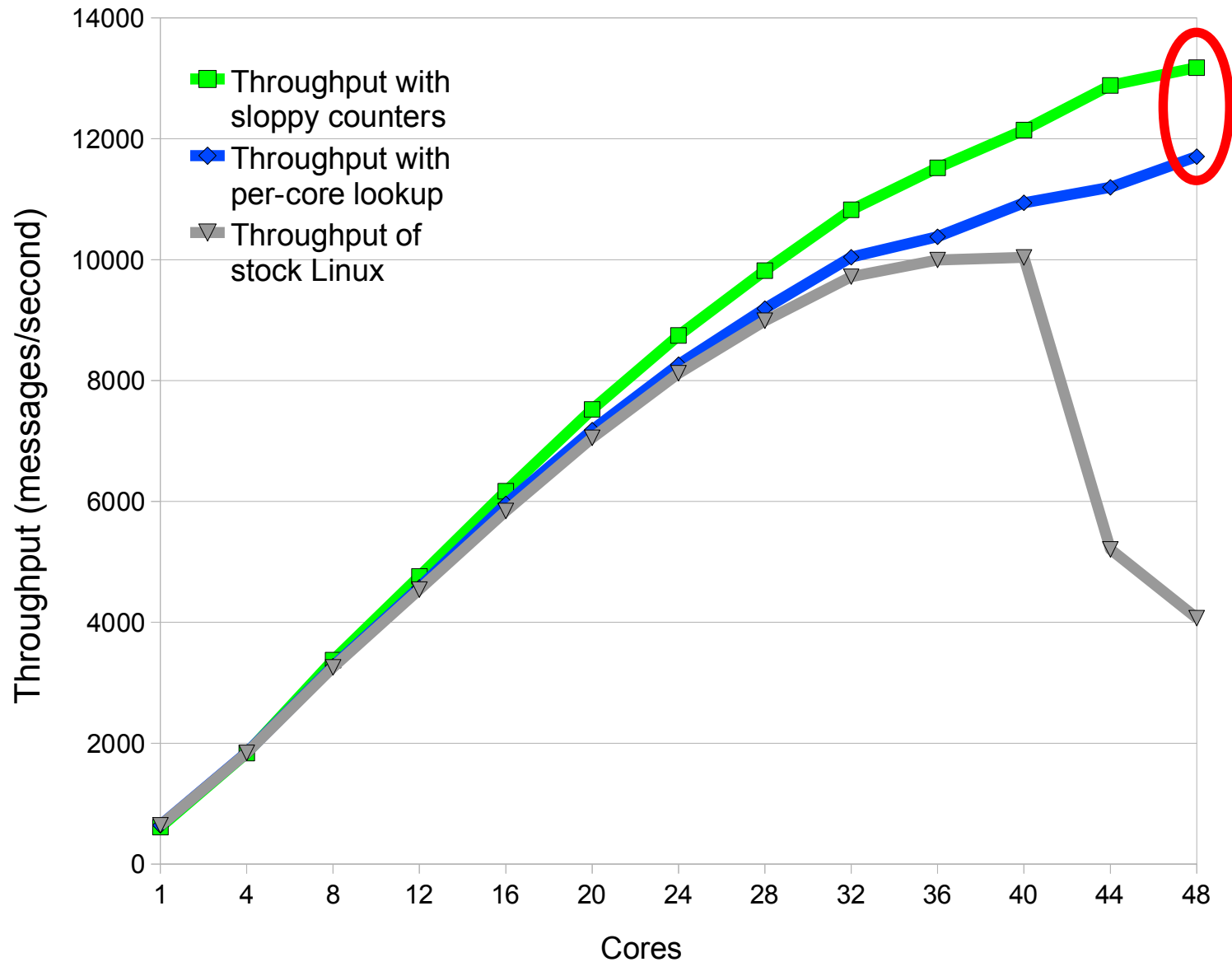
Properties of sloppy counters

- Simple to start using:
 - Change data structure
 - `atomic_inc` \rightarrow `sloppy_inc`
- Scale well: no cache misses in common case
- Memory usage: $O(N)$ space
- Related to: SNZI [Ellen 07] and distributed counters [Appavoo 07]

Sloppy counters: more scalability



Sloppy counters: more scalability



Summary of changes

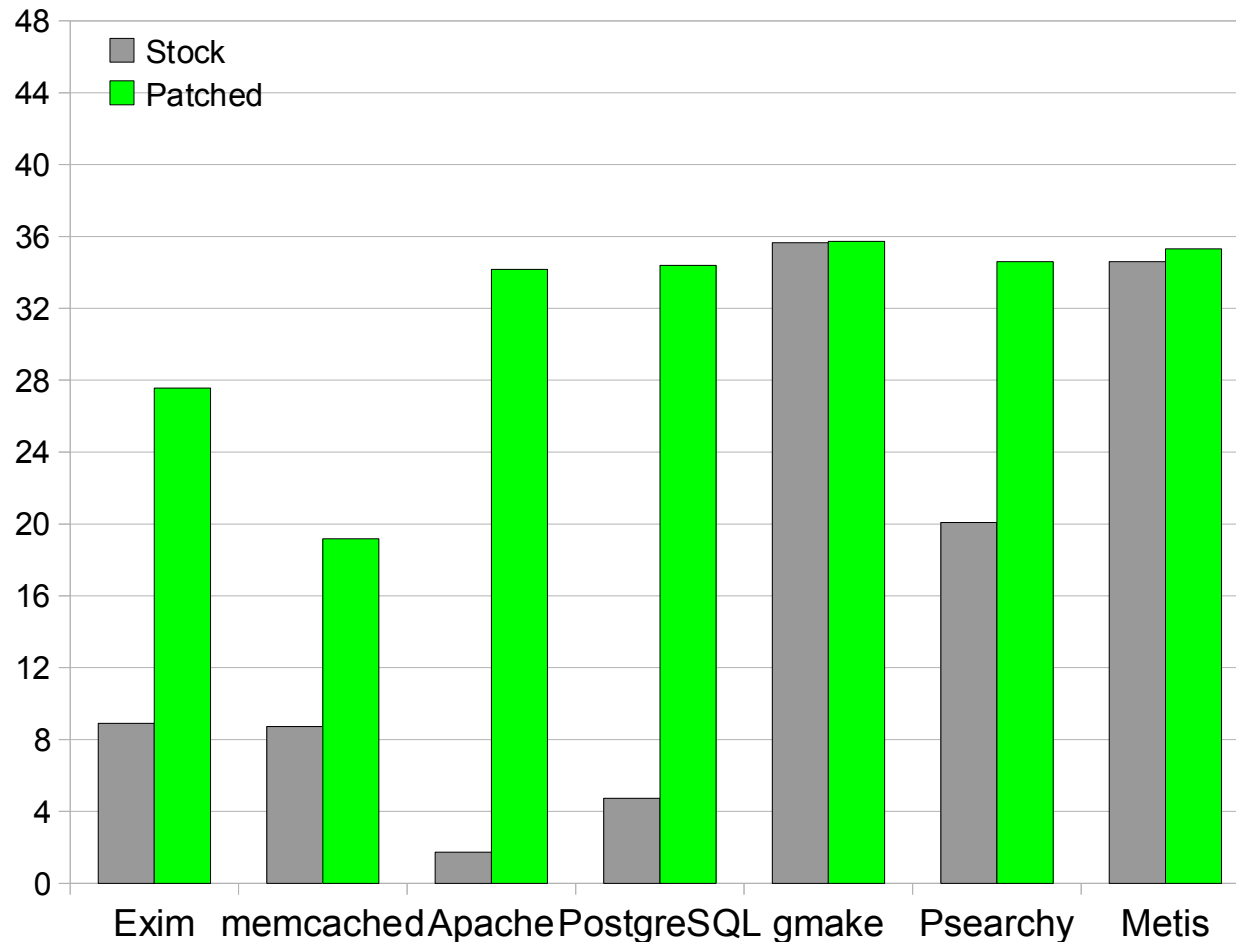
	memcached	Apache	Exim	PostgreSQL	gmake	Psearchy	Metis
Mount tables		X	X				
Open file table		X	X				
Sloppy counters	X	X	X				
inode allocation	X	X					
Lock-free dentry lookup		X	X				
Super pages							X
DMA buffer allocation	X	X					
Network stack false sharing	X	X		X			
Parallel accept		X					
Application modifications				X		X	X

- 3002 lines of changes to the kernel
- 60 lines of changes to the applications

Handful of known techniques [Cantrill 08]

- Lock-free algorithms
- Per-core data structures
- Fine-grained locking
- Cache-alignment
- Sloppy counters

Better scaling with our modifications



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

- Most of the scalability is due to the Linux community's efforts

Current 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

Limitations

- Results limited to 48 cores and small set of applications
- In-memory FS instead of disk
- 48-core AMD machine \neq single 48-core chip

What about on N cores?

- Looming problems
 - fork/virtual memory book-keeping
 - Page allocator
 - File system
 - Concurrent modifications to address space
- Can we fix with small changes?
 - We don't know
- Future work: study Linux at the level of cache lines shared

Related work

- Linux and Solaris scalability studies [Yan 09,10] [Veal 07] [Tseng 07] [Jia 08] ...
- Scalable multiprocessor Unix variants
 - Flash, IBM, SGI, Sun, ...
 - 100s of CPUs
- Linux scalability improvements
 - RCU, NUMA awareness, ...
- Our contribution:
 - In-depth analysis of kernel intensive applications

Conclusion

- Linux has scalability problems
- They are easy to fix or avoid up to 48 cores

`http://pdos.csail.mit.edu/mosbench`

