An Analysis of Linux Scalability to Many Cores

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

OSDI 2010

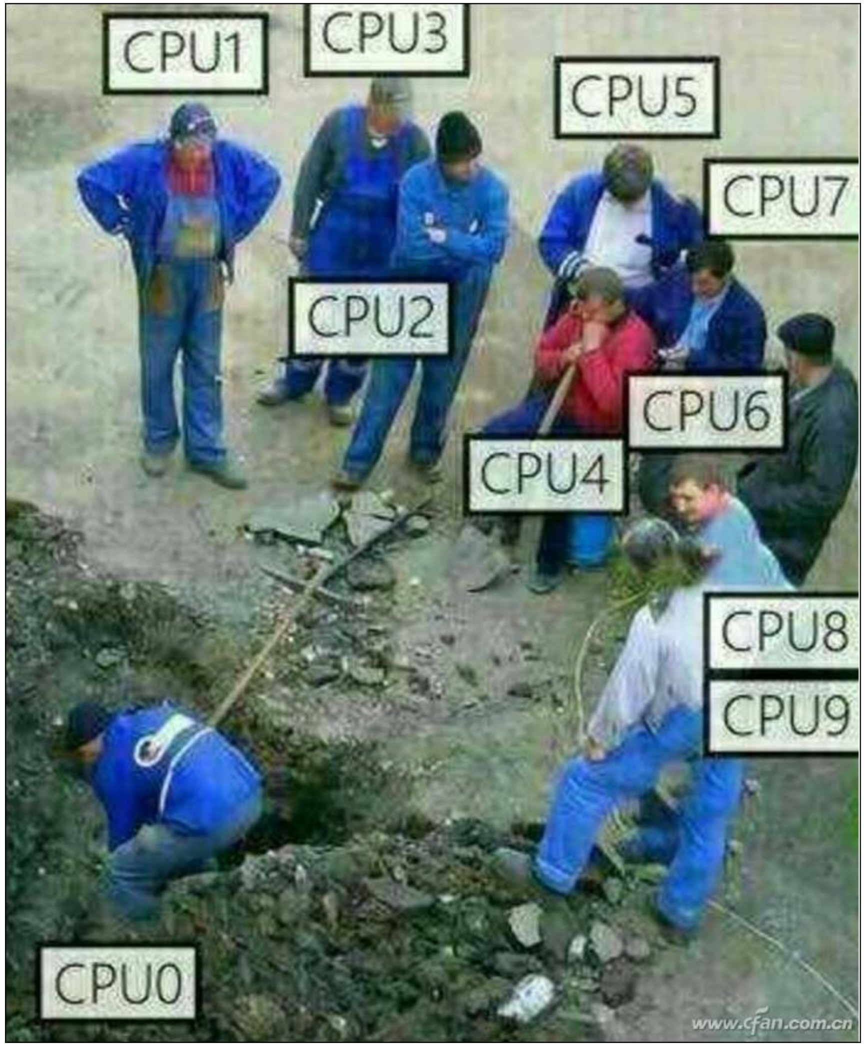
Presenter: Yifan Li

VS

Dual Cores

TEN Cores !!!

Uh Oh


CPU1
CPU3
CPU5
CPU7
CPU2
CPU6
CPU4
CPU8
CPU9
CPU0
www.cfan.com.cn

Uh Oh

We have many cores,
but they’re not working together!

We need to modify our {OS, applications}
to scale to many cores.

An Analysis of Linux Scalability to Many Cores

Background

Author Introduction

(First Author)
Silas Boyd-Wickizer
Now: CTO at Valora

(Last Author)
Nickolai Zeldovich
Professor at MIT

Affiliation: MIT CSAIL/PDOS

Author Introduction

Austin T. Clements

Aleksey Pesterev (Now at Philo)

Robert Morris
(Professor, MIT)
Morris Worm

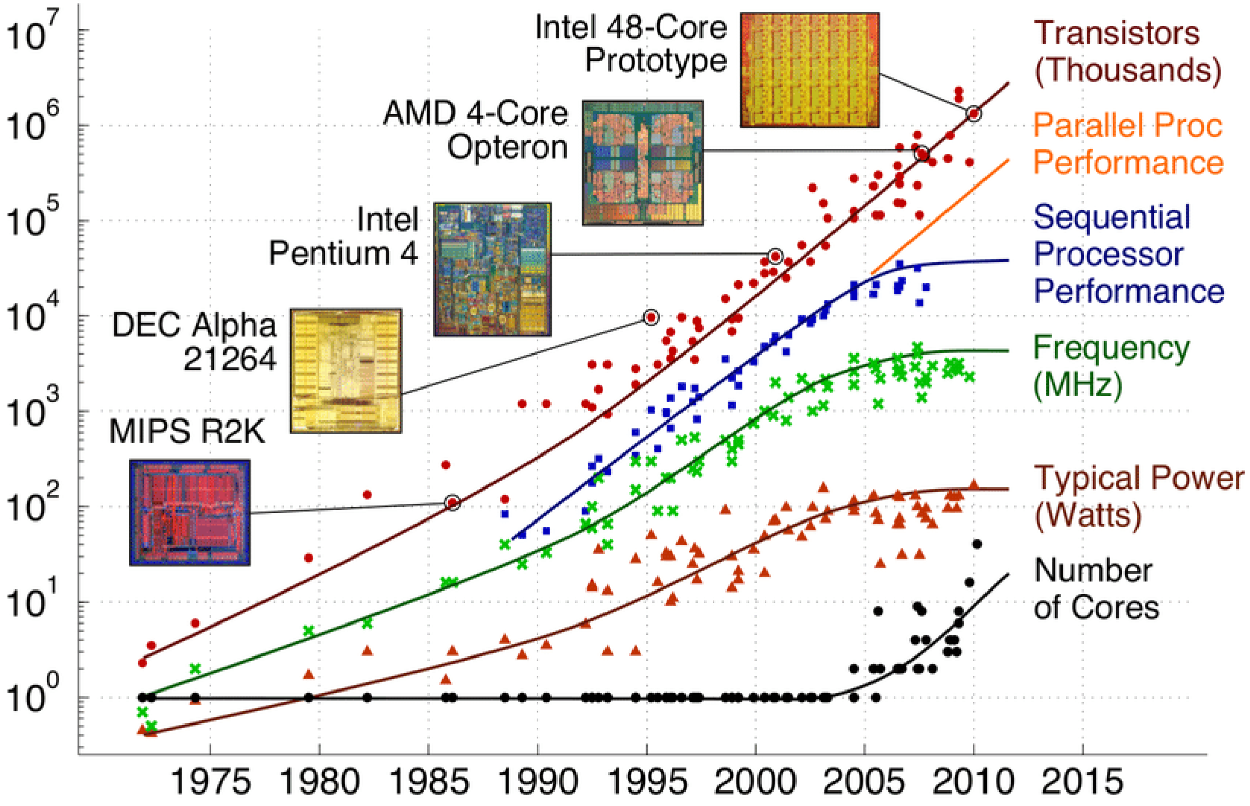
Yandong Mao (Now at Databricks)

M. Frans Kaashoek
(Professor, MIT)
Author of Exokernel

Affiliation: MIT CSAIL/PDOS

Background

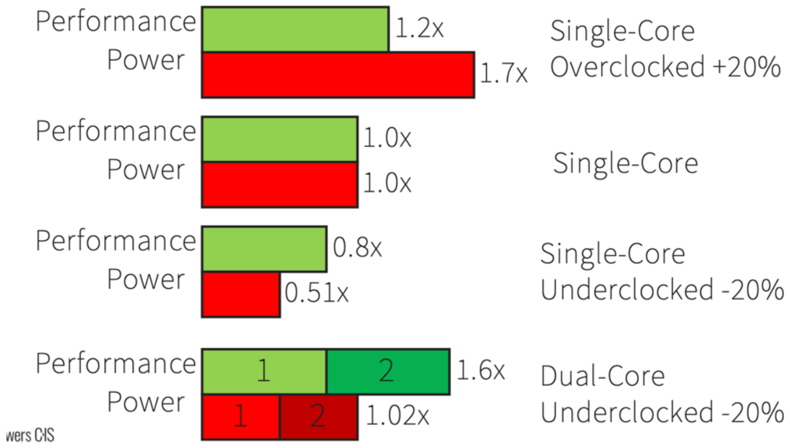
Multicore CPUs emerge around 2005,
why?

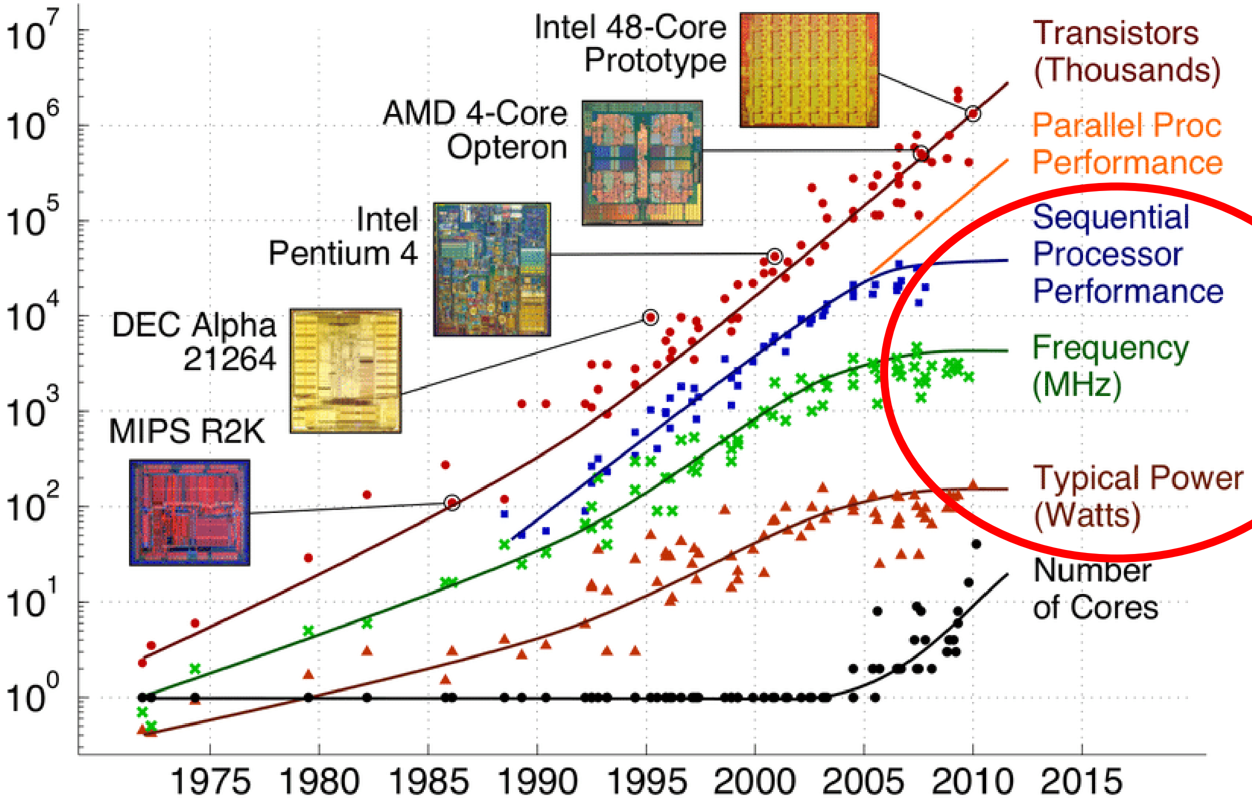


Reference: https://www.researchgate.net/figure/Evolution-of-multi-core-processors-1_fig1_281534326

Background

Multicore CPUs emerge around 2005, as clock frequency hits the wall.

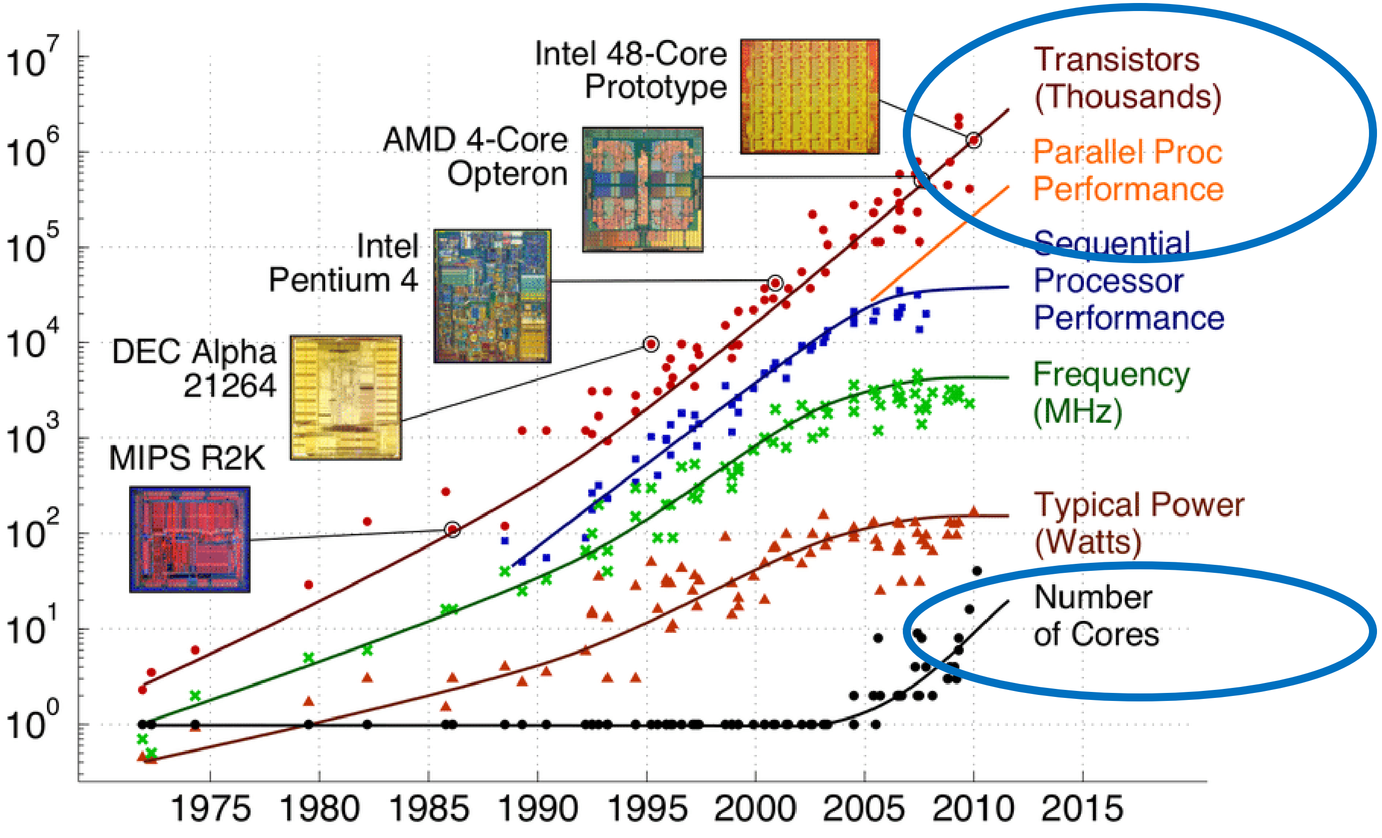




Reference: https://www.cs.cornell.edu/courses/cs3410/2025sp/lectures/22-multicore-notes.pdf

Background

Multicore CPUs emerge around 2005,
as clock frequency hits the wall.

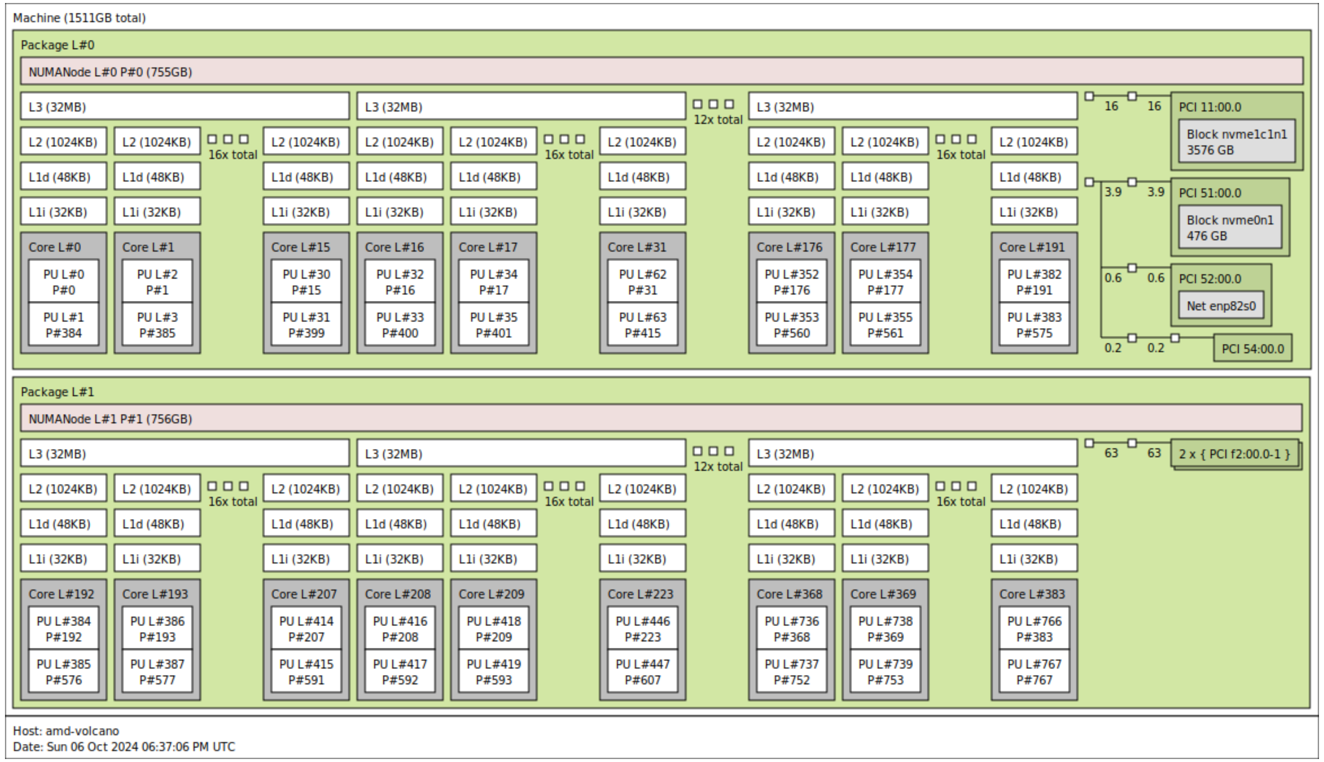


Reference: https://www.cs.cornell.edu/courses/cs3410/2025sp/lectures/22-multicore-notes.pdf

Background

Core counts have skyrocketed since 2020

EPYC 9965 packs **192 cores** on a single die!



EPYC 9965 Topology w/ 2 CPUs.

Scalability and Amdahl’s law

We do not get 192x speedup for using 192 cores.

Scalability:

The ability to handle more works / fulfills work faster as CPU core count increases.

Scalability and Amdahl's law

Amdahl's law:

$$SpeedUp = \frac{T_{all}}{T_{Serial} + \frac{T_{Parallel}}{N}}$$

Motivation

Motivation: Scalability problems

Amdahl's law:

$$SpeedUp = \frac{T_{all}}{T_{Serial} + \frac{T_{Parallel}}{N}}$$

Scalability is **limited** by sequential part,

And **worsen** by contention on resources.

Motivation: Scalability problems

Amdahl's law:

$$SpeedUp = \frac{T_{all}}{T_{Serial} + \frac{T_{Parallel}}{N}}$$

Scalability is **limited** by sequential part,

And **worsen** by contention on resources.

Discussion: Any examples?

Scalability: Spinlocks

I can finish in 20 minutes

Total Time:
20 Minutes

Scalability: Spinlocks

I can finish in 20 minutes

Let me do a part!

Total Time:
10 Minutes (CPU0)+
5 Minutes(Transition)+
10 Minutes(CPU6)=
25 Minutes!

Scalability: Spinlocks

Watch out!

You are doing wrong!

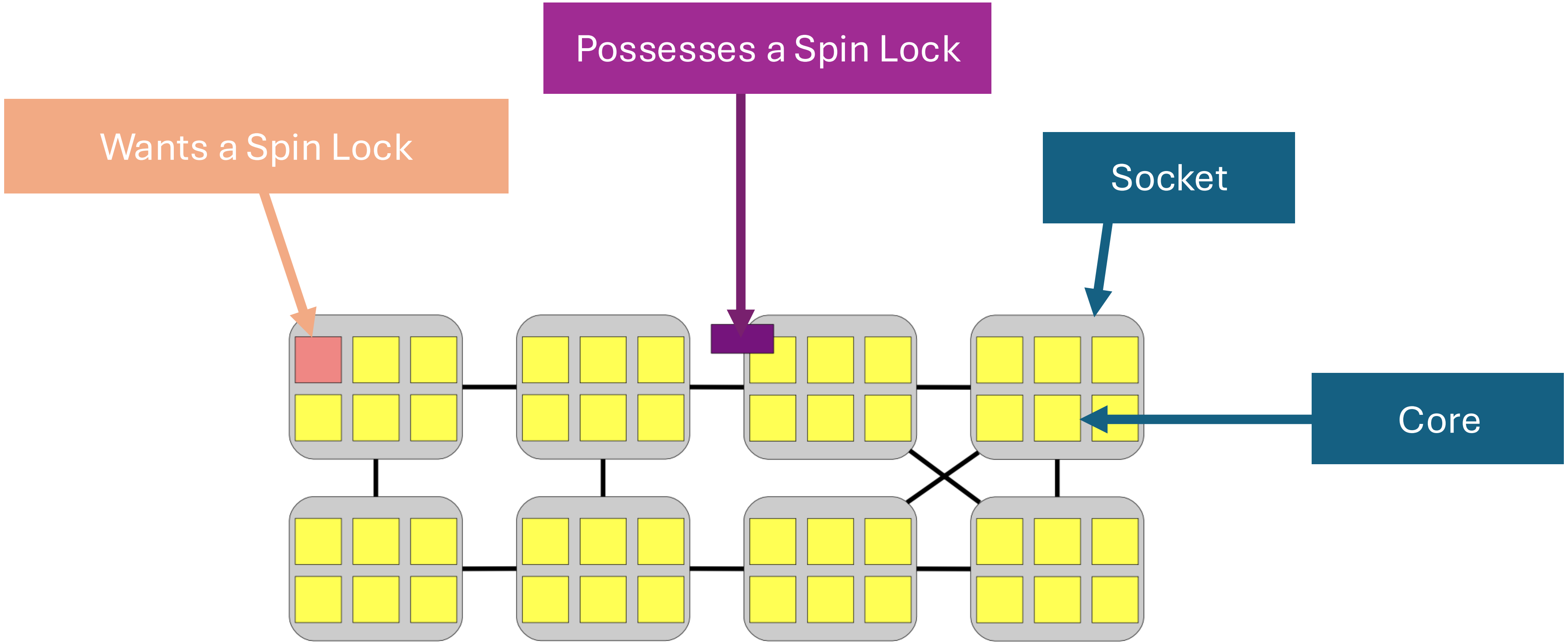
This way!

I can finish in 20 minutes

Total Time:
20 Minutes (CPU0)+
10 Minutes(Arguing)=
30 Minutes!

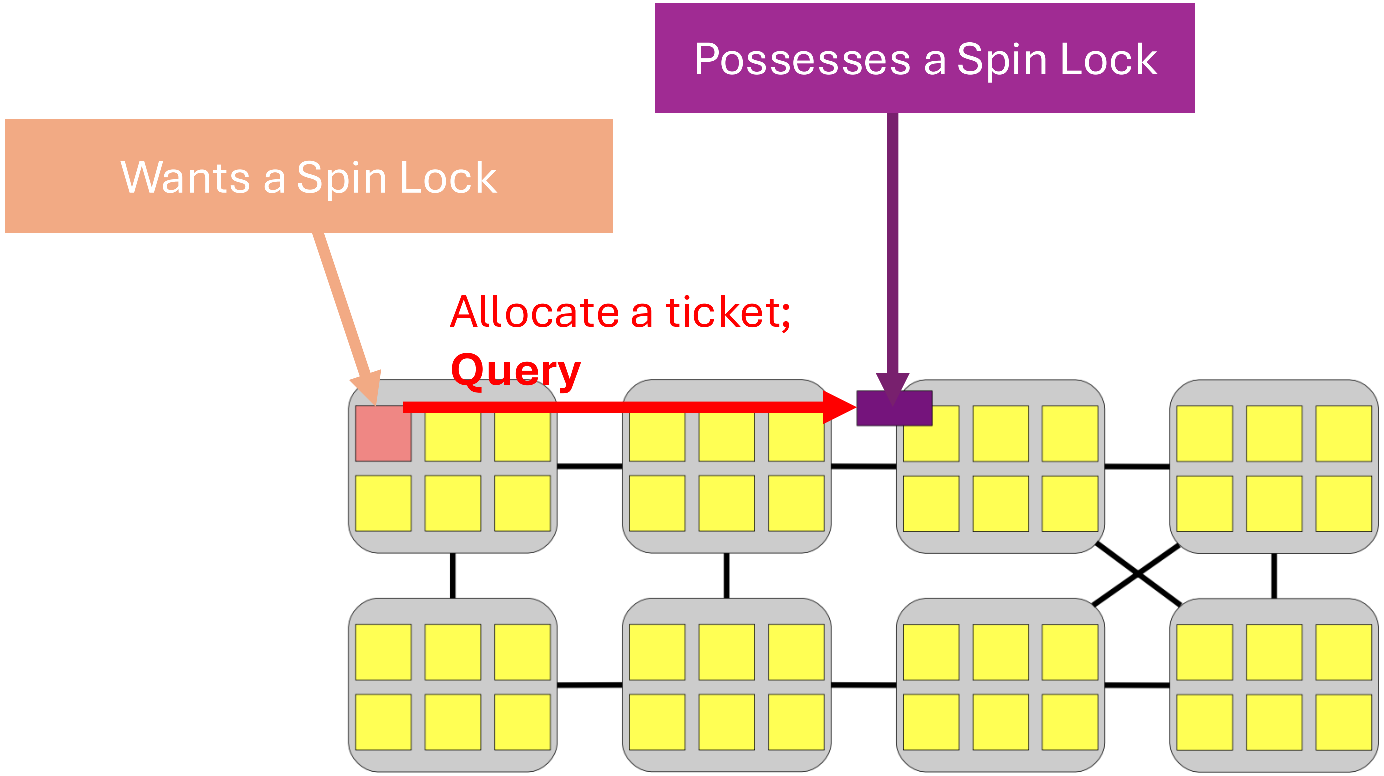
This is (basically) what happens to Linux Spinlock Design!

Scalability: Spinlocks



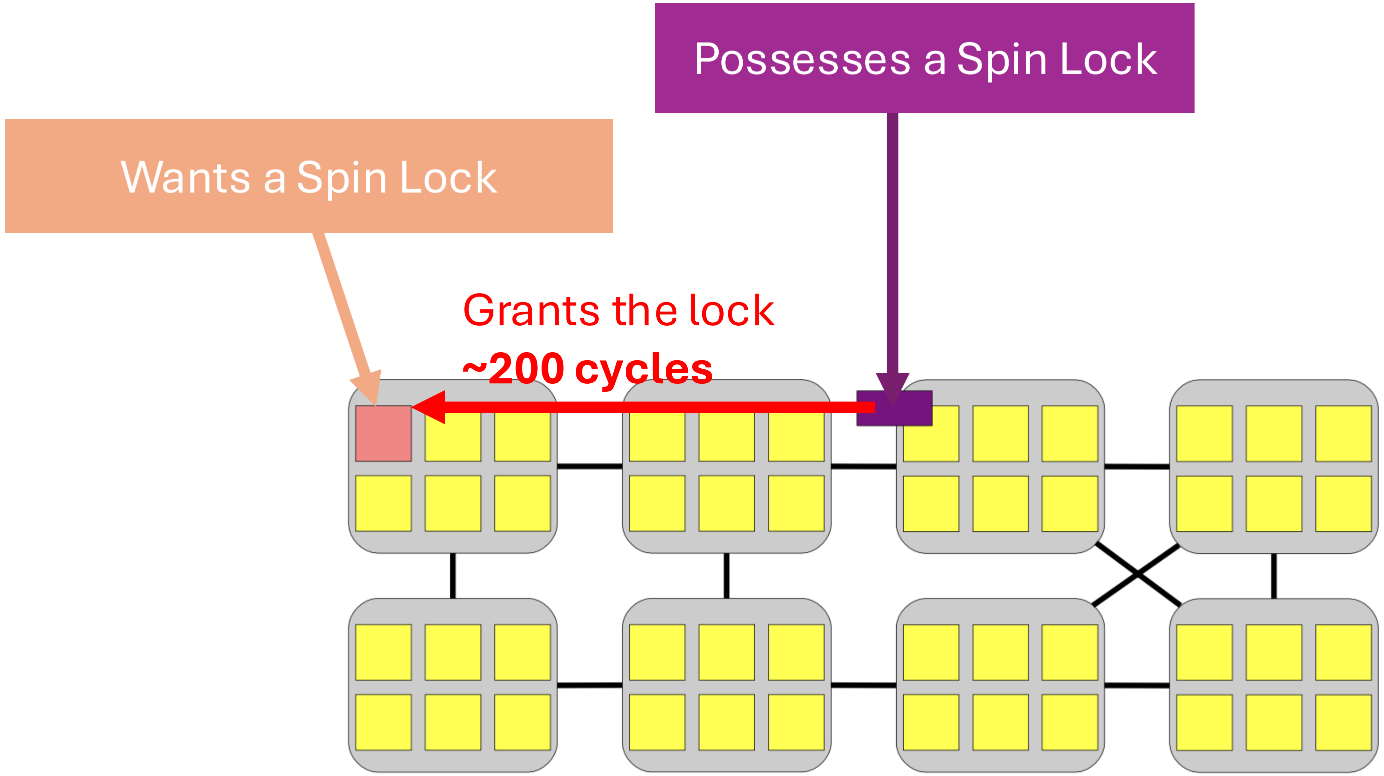
Scalability: Spinlocks

```
spin_lock(&vfsmount_lock);
mnt = hash_get(mnts, path);
spin_unlock(&vfsmount_lock);
```



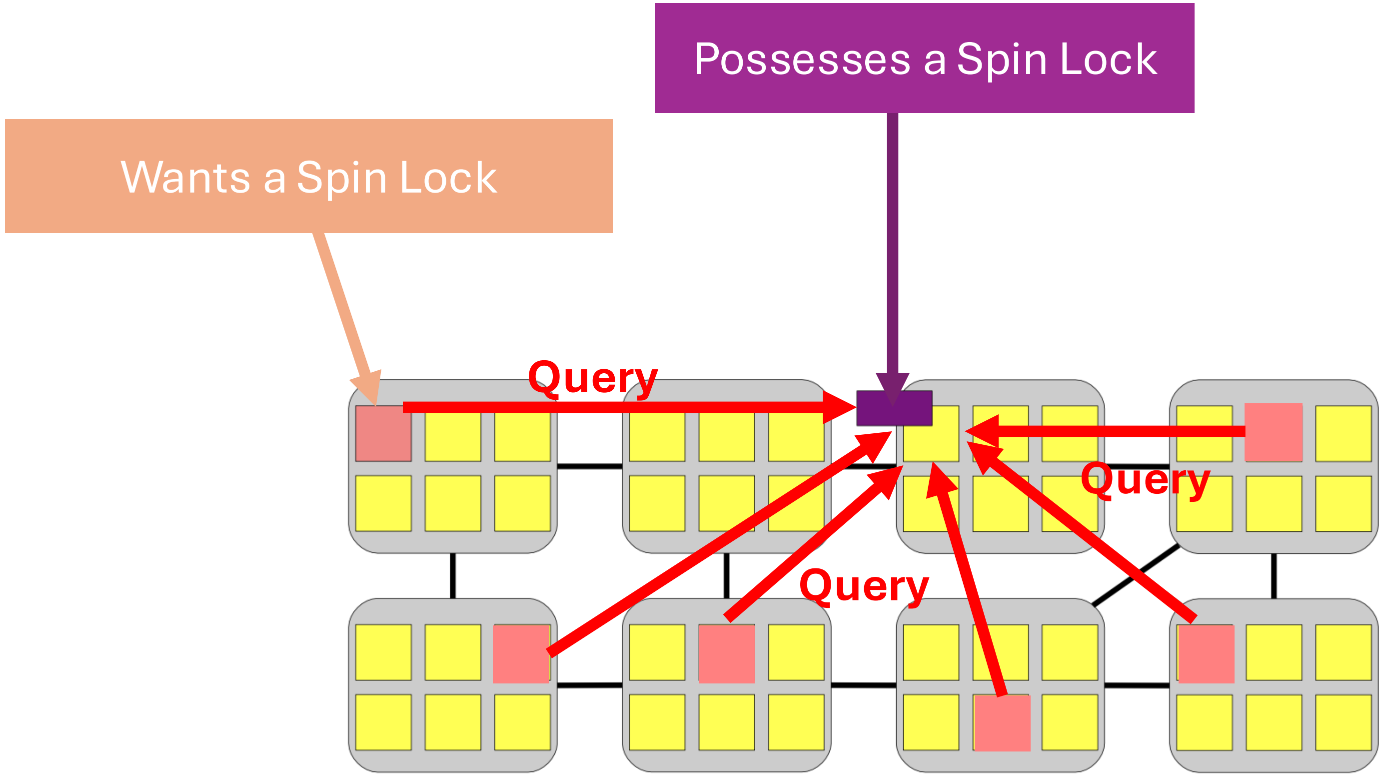
Scalability: Spinlocks

```
spin_lock(&vfsmount_lock);
mnt = hash_get(mnts, path);
spin_unlock(&vfsmount_lock);
```



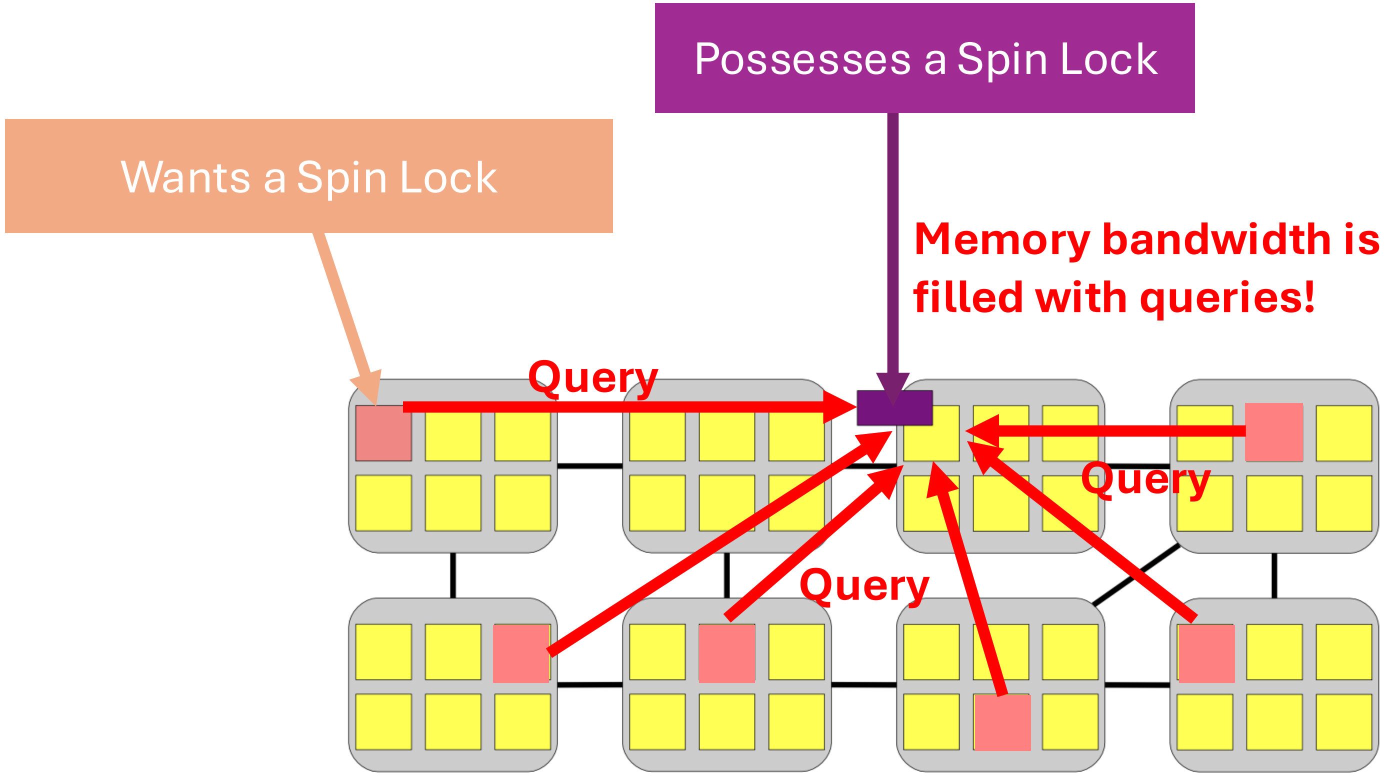
Scalability: Spinlocks

```
spin_lock(&vfsmount_lock);
mnt = hash_get(mnts, path);
spin_unlock(&vfsmount_lock);
```



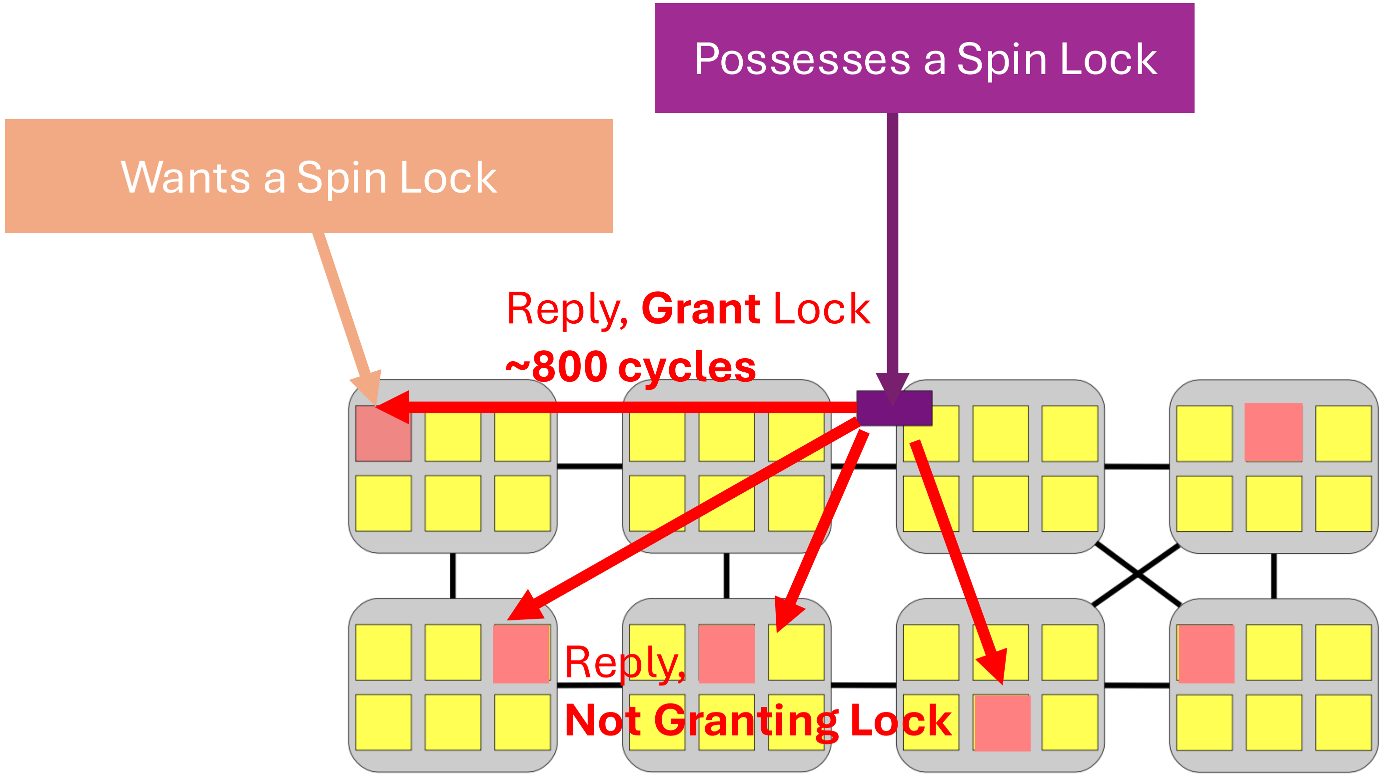
Scalability: Spinlocks

```
spin_lock(&vfsmount_lock);
mnt = hash_get(mnts, path);
spin_unlock(&vfsmount_lock);
```



Scalability: Spinlocks

```
t = atomic_inc(lock>next_ticket);
while (t != lock->current_ticket)
 /* Spin */
```



Motivation: Scalability problems

Scalability is **limited** by sequential part,

And **worsen** by contention on resources: locks, atomics

Motivation: Kernel Scalability

Scalability is **limited** by sequential part,

And **worsen** by contention on resources: locks, atomics

These bottlenecks exist in Linux Kernel!
e.g. TLB, filesystem, I/O handling…

And **applications spend a lot of time in the kernel.**

Motivation: Kernel Scalability

Application	**Single Core** **Kernel Time Percentage**
Mail Server	69%
Object Cache	80%
Web Server	60%
Database	1.5% (82% at 48 cores)
Parallel Build	7.6%
File Indexer	1.9% (23% at 48 cores)
MapReduce	3% (16% at 48 cores)

Motivation: Kernel Scalability

Many studies have been trying to investigate this problem.

Discussion:

- Will the common monolithic kernel work well?
- What kind of kernel design is the best fit?

Motivation: Kernel Scalability

Many studies have been trying to investigate this problem.

Some come up with new OS design:

Corey, Barrelfish, fos……

Motivation: Kernel Scalability

Many studies have been trying to investigate this problem.

Some come up with new OS design:

Corey: applications should control sharing

- An **exo-kernel** like design
- Memory address space (sharing) is **controlled by applications**
- Kernel avoids unnecessary sharing, provides interfaces for explicit sharing
- Some cores may be dedicated to kernel functions
- A proof-of-concept system

Motivation: Kernel Scalability

"There is a sense in the community that traditional kernel designs ***won't scale well*** *on multicore processors: that applications will spend an* ***increasing*** *fraction of their time* ***in the kernel*** *as the number of cores increases."*

This work focuses on:

- What's the bottleneck for (applications on) **current Linux OS**?
- How serious?
- Can we **remedy** them?

Methods

Methods

1. Run experiments on stock Linux, vary core count;
2. Identify bottlenecks for multicore execution;
3. Fix the bottlenecks; Goto 1.

Methods

1. Run experiments on stock Linux, vary core count;
2. Identify bottlenecks for multicore execution;
3. Fix the bottlenecks; Goto 1.

Contributions

- **MOS**Bench, a set of 7 applications for testing parallel performance.
- 16 Patches (3k loc) for Linux kernel;
- Scale 7 real applications efficiently to 48 cores.

MOSBench

Set 1 - Applications not scaling well on Linux

- **Memcached: Object cache**. Launches one instance per core to avoid contention on the global hash table.
- **Apache: Web server.** Uses one instance, one process per core, multiple threads.
- **Metis: MapReduce Library**. Combined with an application that generates inverted indices.

MOSBench

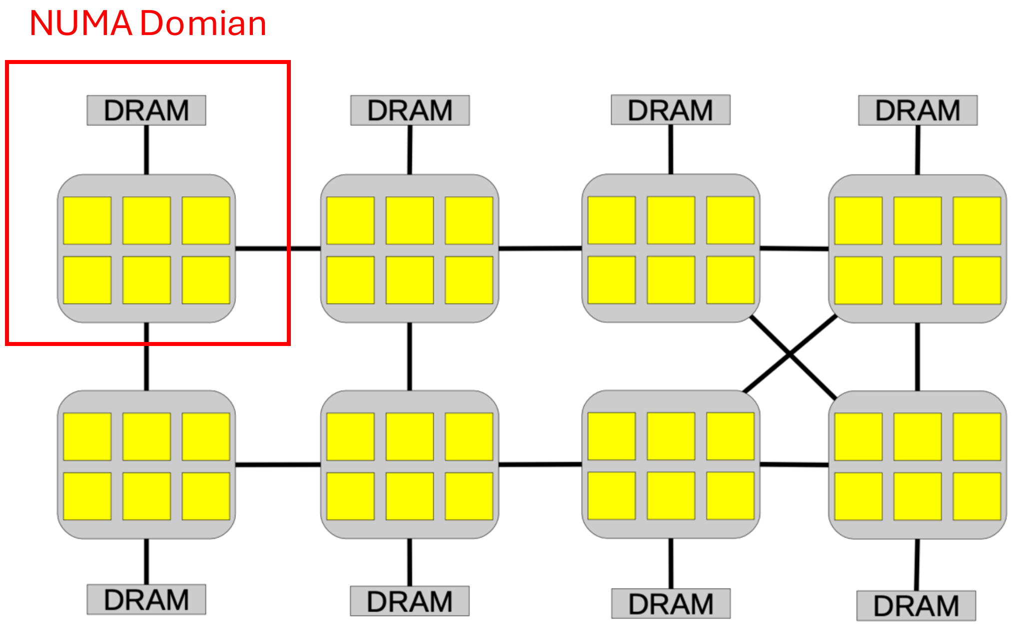
Set 2 - Applications designed for parallel execution and kernel-intensive

- **Exim: Mail server**. A single master process is started and forks a new process for each connection.
- **PostgreSQL: Database server.** One process per connection.
- **Gmake: Parallel build tool.** Used to build Linux kernel for benchmark, creates many processes.
- **Psearchy: File indexer.** An indexer is run on each core, which shares a working queue of input files.

Setup

Hardware:

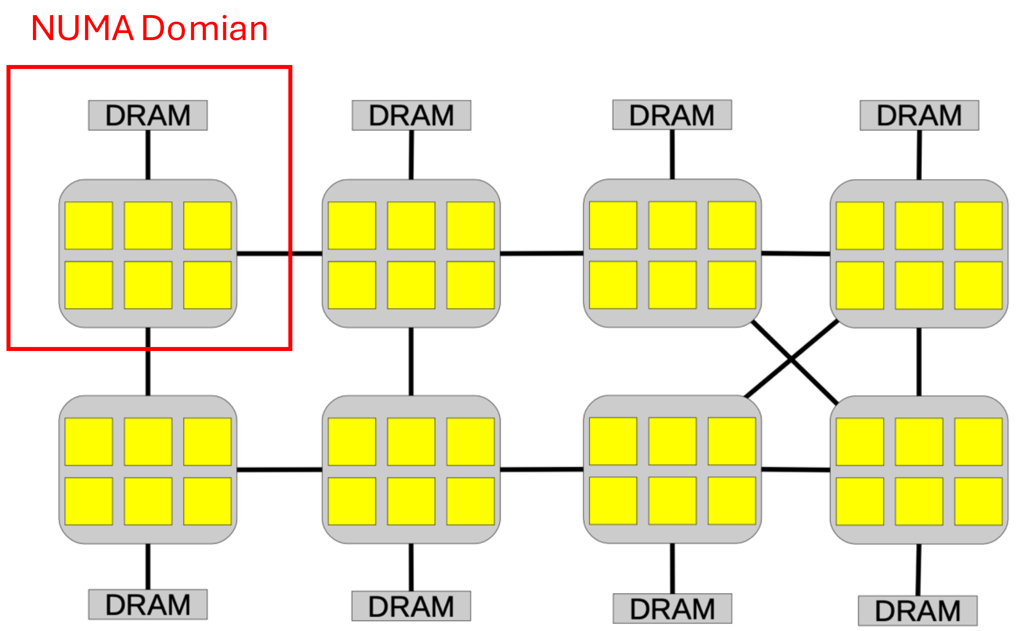
- 8 * (6 core AMD M4985 CPU)
- "Weird" topology
- Non Unified Memory Access
- RAM disk to avoid disk bottleneck



Setup

Hardware:

- 8 * (6 core AMD M4985 CPU)
- “Weird” topology
- Non Unified Memory Access
- RAM disk to avoid disk bottleneck

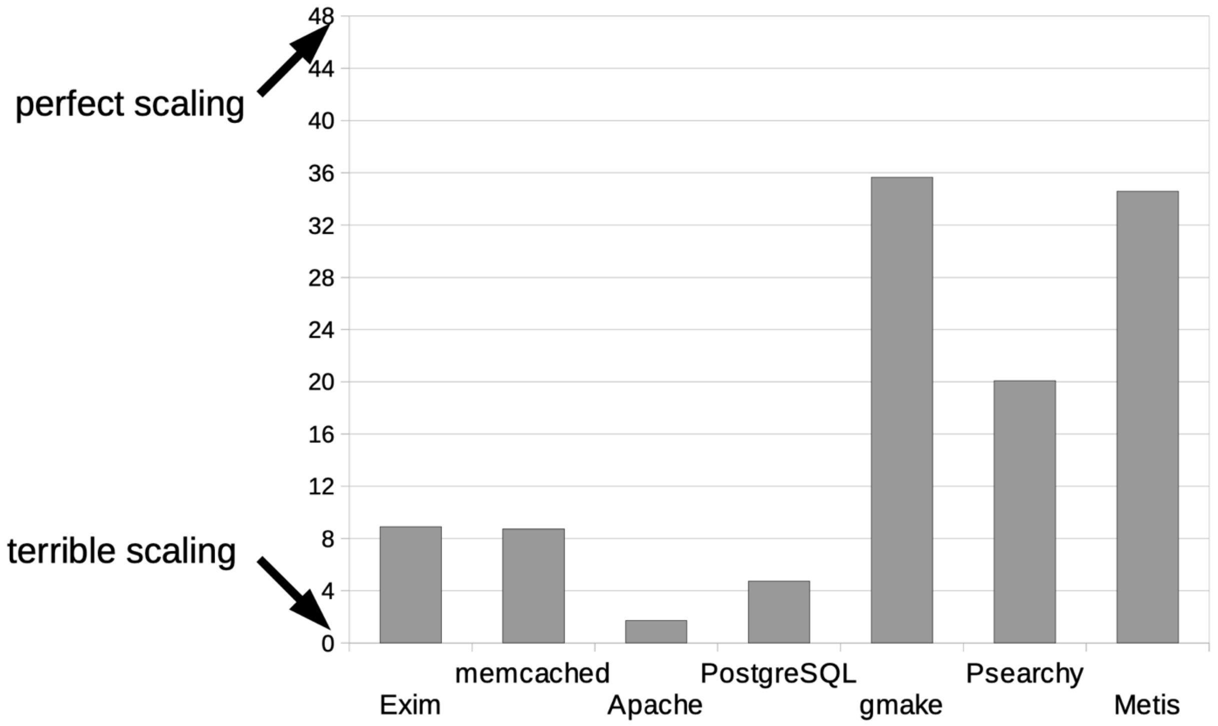


Software:

- Latest Linux kernel (2.6.35-rc5)
- 7 commonly-used server software

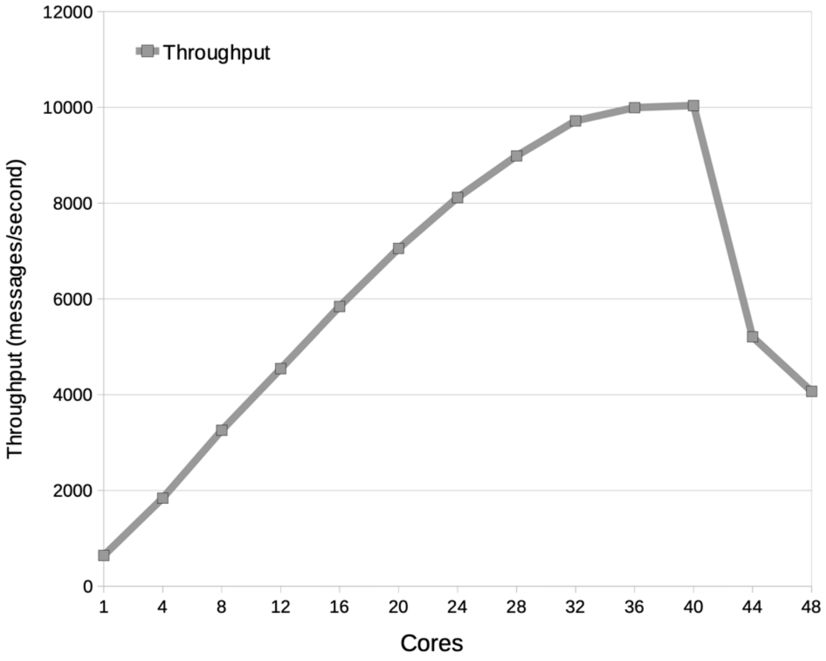
Case studies

Starting Point: Poor Scaling



Speedup achieved using 48 cores

Exim



Performance Drop

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

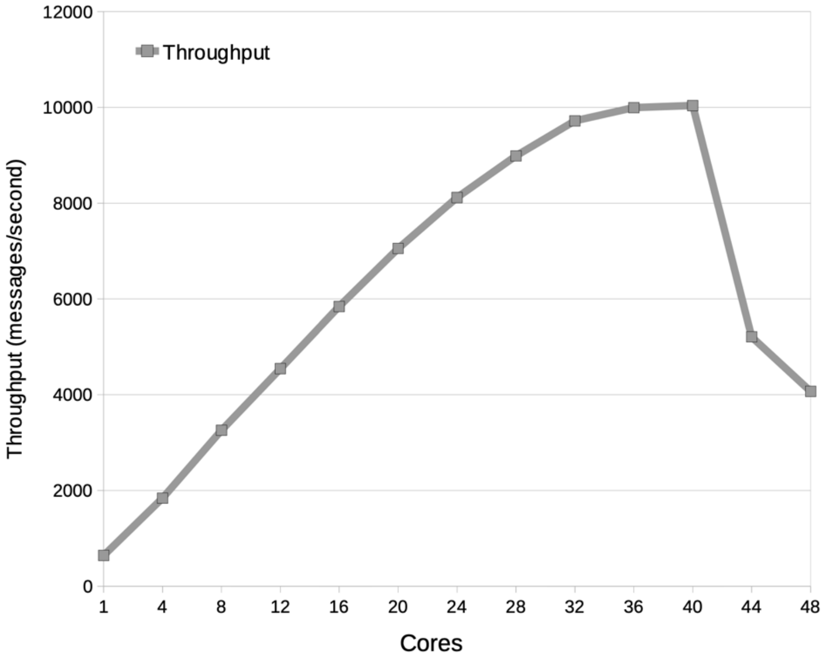
48 cores:
4000 msg/sec

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

Profiling Result

Reference: https://www.usenix.org/legacy/events/osdi10/tech/slides/boyd-wickizer.pdf

Exim



Performance Drop

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

Profiling Result

Exim Bottleneck

```
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 Code

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

Profiling Result

Exim Bottleneck: Reading Mount Table

```
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 Code

This is a critical path of `sys_open`;

Hashing itself is cheap;

Exim Bottleneck: Reading Mount Table

```
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;
}
```

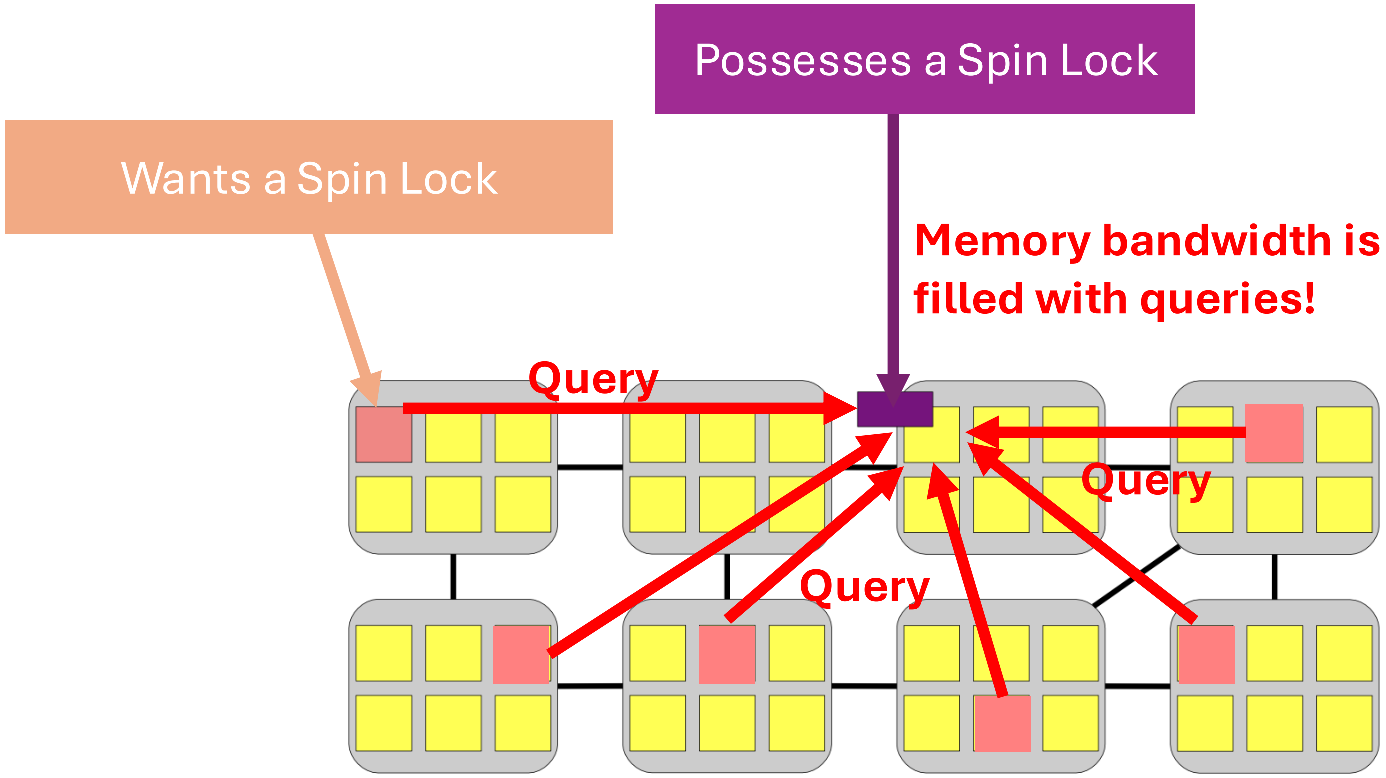
Bottleneck Code

This is a critical path of `sys_open`;

Hashing itself is cheap;

Spinlock is consuming much time!

Exim Bottleneck: Reading Mount Table




```
spin_lock(&vfsmount_lock);
mnt = hash_get(mnts, path);
spin_unlock(&vfsmount_lock);
```

Exim Solution: Mount Caches

```
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;
}
```

Bottleneck Code

Implement Per-core mount caches;

Exim Solution: Mount Caches

```
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;
}
```

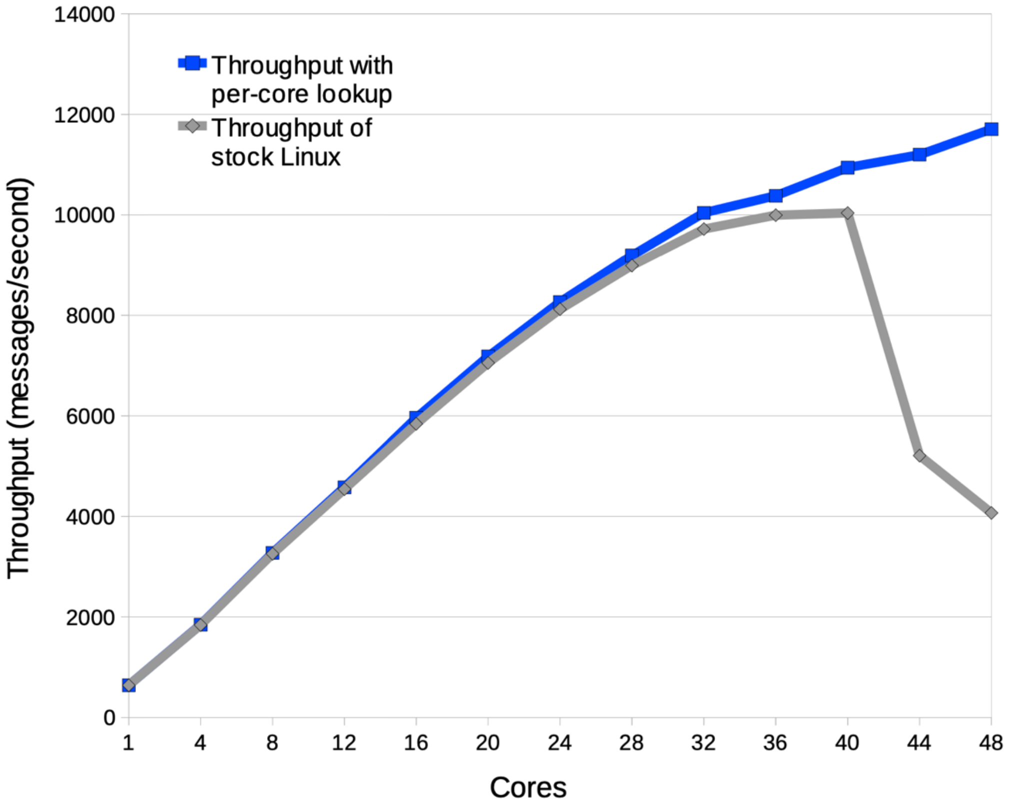
Bottleneck Code

Implement Per-core mount caches;

Depending Observation: mount table is rarely modified;

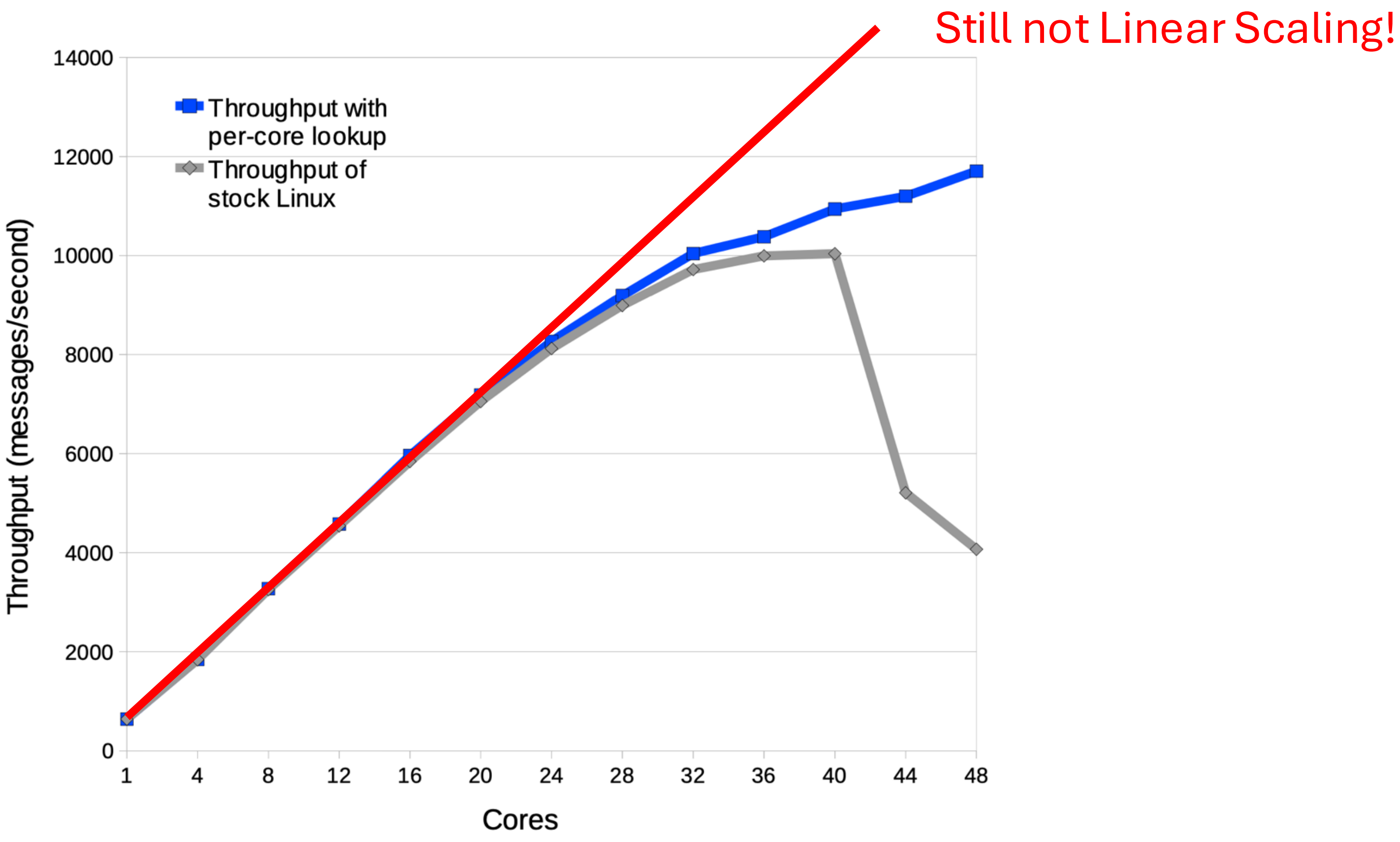
When modified, invalidate all cache.

Exim Performance Improvement



Performance

Exim Performance Improvement



Performance

Exim Bottleneck: Reference Counting

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

Profiling result w/ mount cache

Exim Bottleneck: Reference Counting

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

Profiling result w/ mount cache

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

Bottleneck Code

Exim Bottleneck: Reference Counting

Reference Counting indicates whether kernel can free an object;

Here `dentry` is file name cache.

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

Bottleneck Code

Exim Bottleneck: Reference Counting

Reference Counting indicates whether kernel can free an object;

Here `dentry` is file name cache.

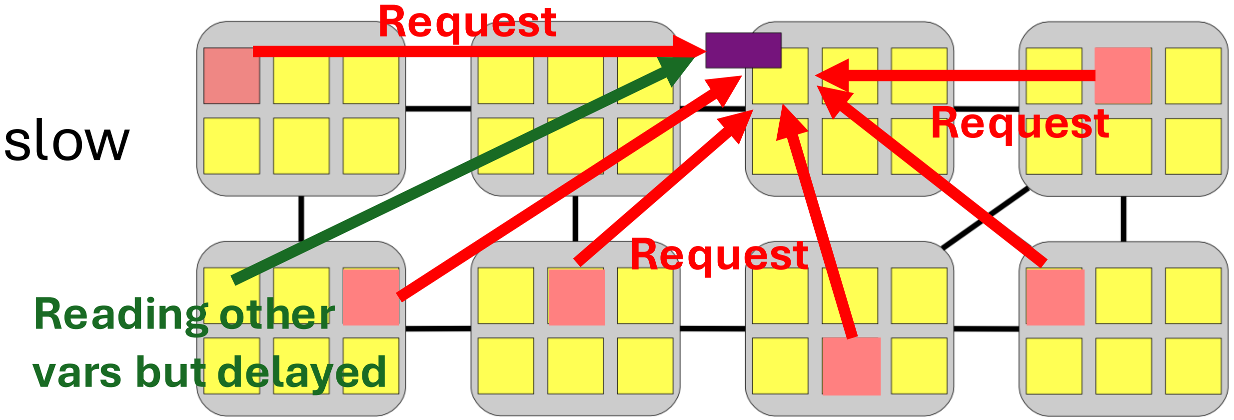
Vars are locked to a certain cache line with atomic operations --

Reading a var from memory is slow due to cache mechanism;

Interconnect is congested.

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

Bottleneck Code



Reference Counting Solution: Sloppy Counters

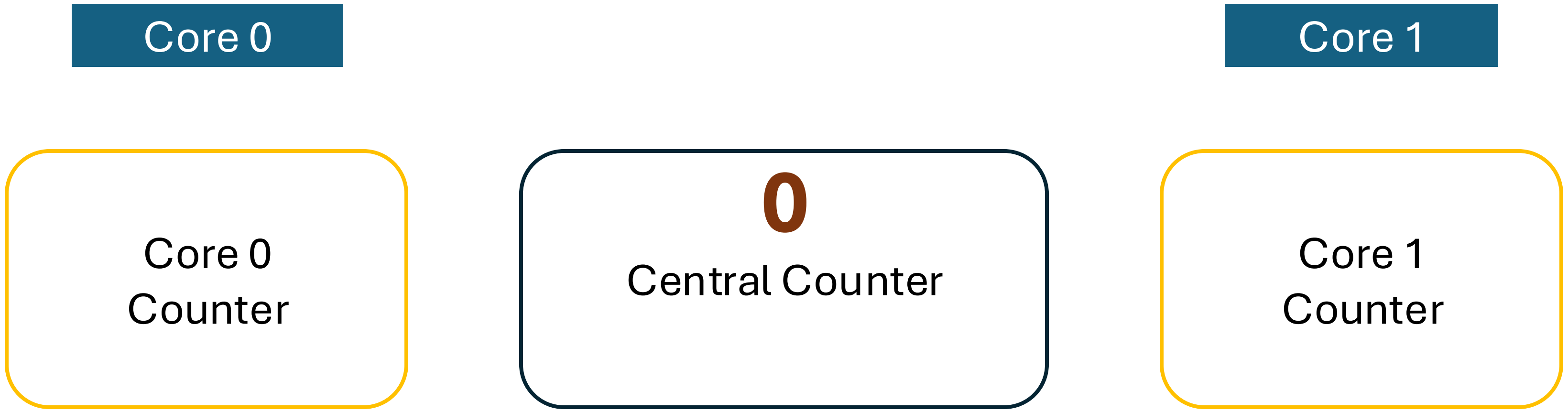
Observation:
The true and precise value of reference count is typically not needed.

Thus, we can use a “loose” counter,
Each core holds a few “spare” references.

Solution: Sloppy Counters

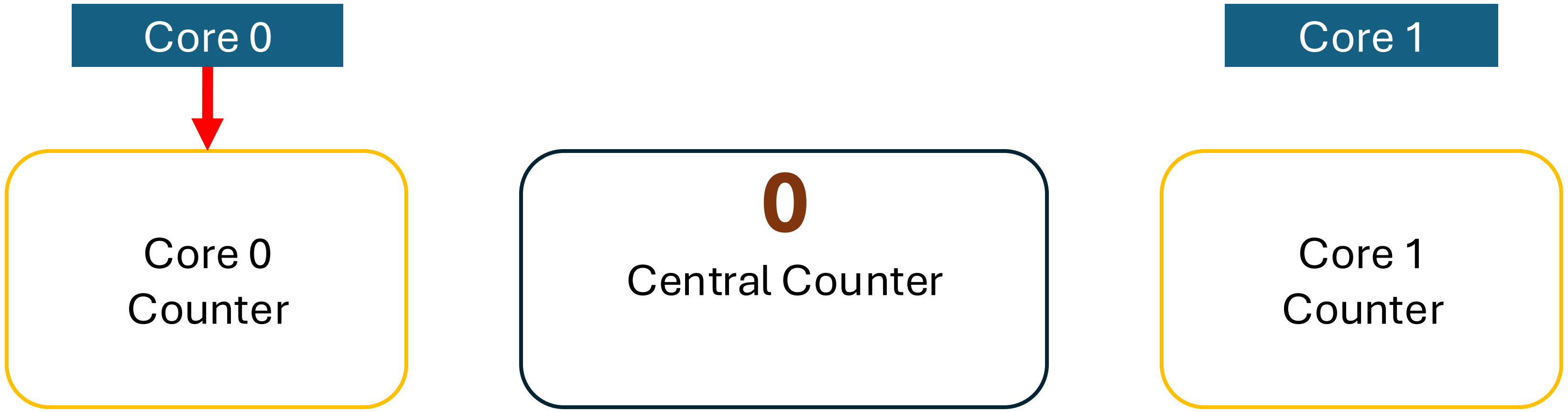
A sloppy counter represents one logical counter as

- a single shared central counter, and
- a set of per-core counts of spare references



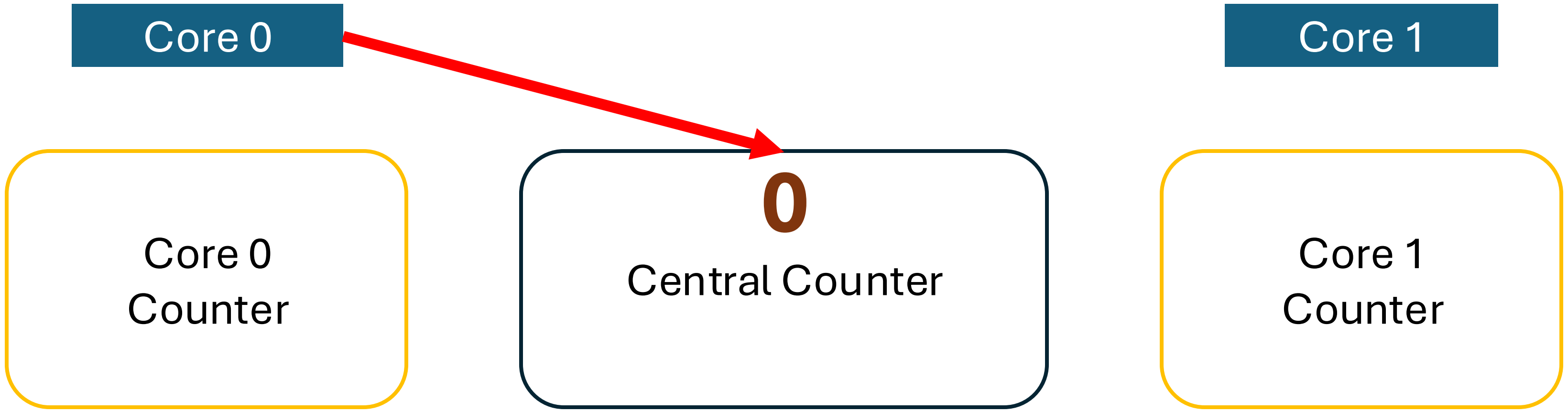
Solution: Sloppy Counters

When a core wants a reference, it first look at local counter for spare references.



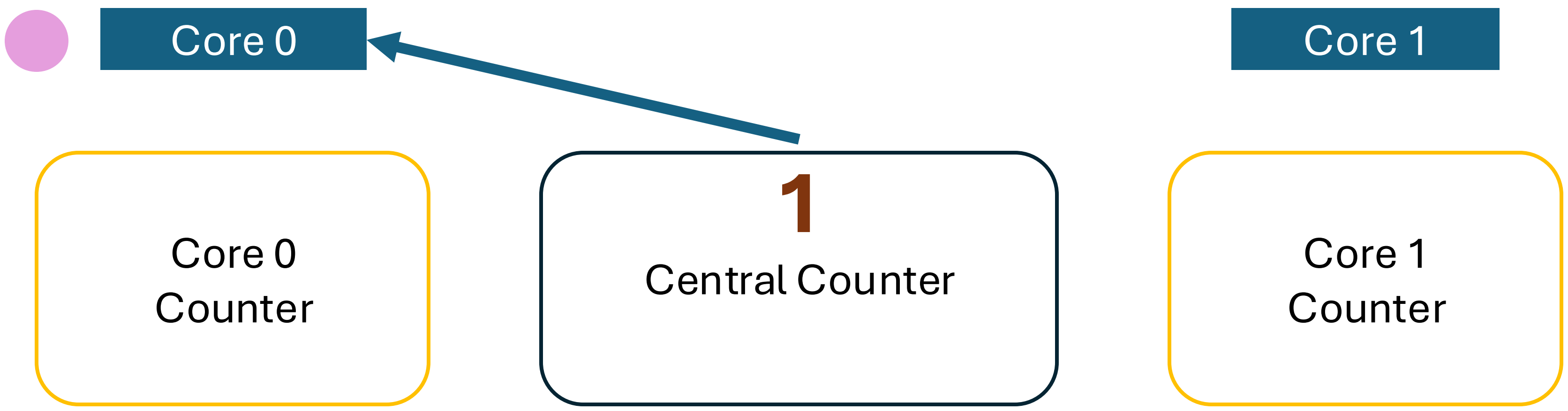
Solution: Sloppy Counters

When a core wants a reference, it first look at local counter for spare references.
If and only if there's no spare ones at local counter, it'll go to central counters.



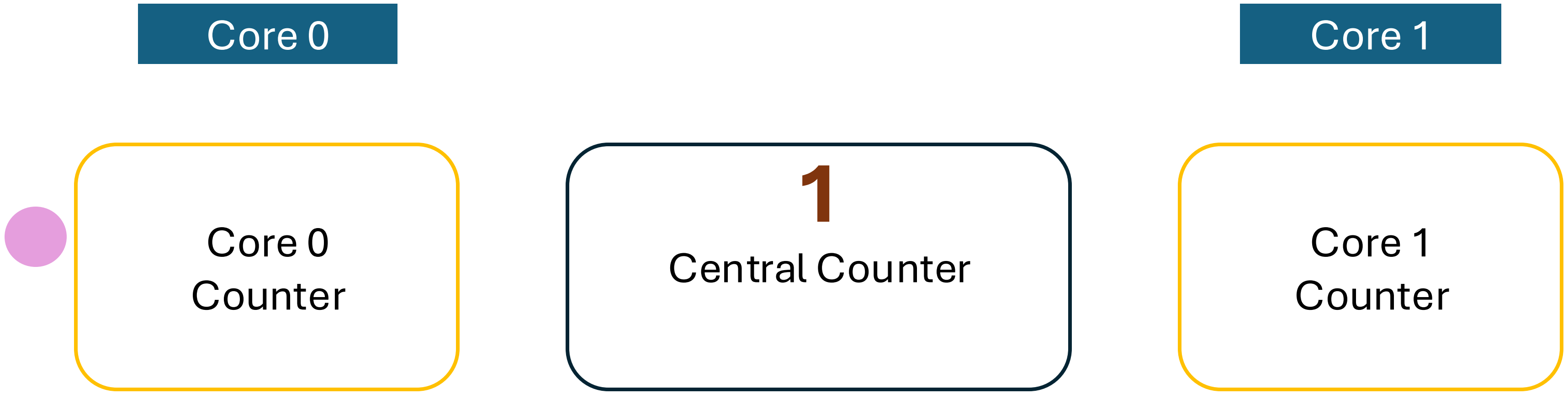
Solution: Sloppy Counters

When a core wants a reference, it first look at local counter for spare references.
If and only if there's no spare ones at local counter, it'll go to central counters.



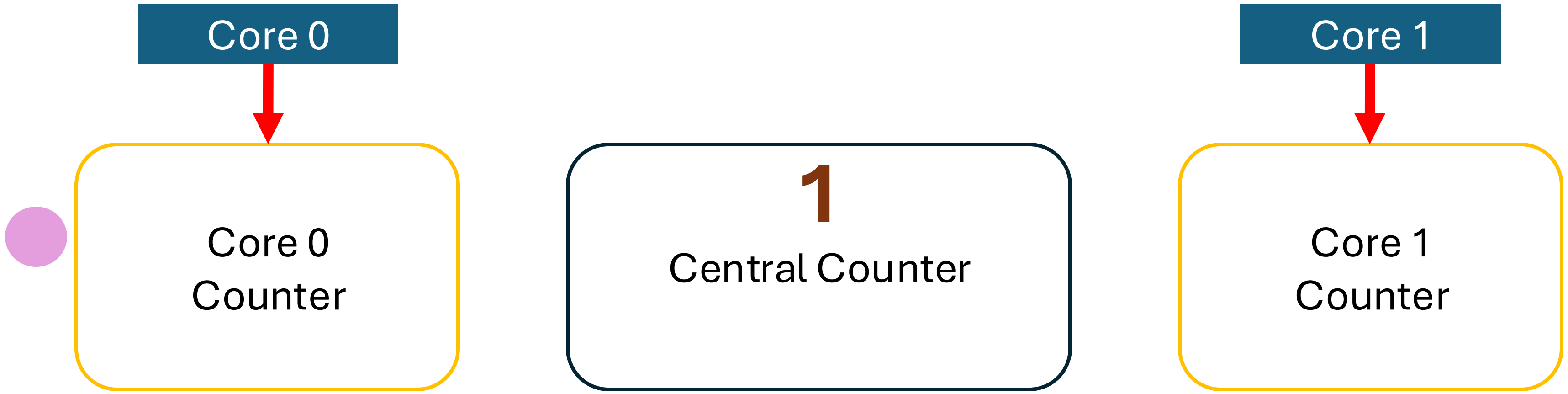
Solution: Sloppy Counters

When a core releases a reference, it'll go back to the local counter.



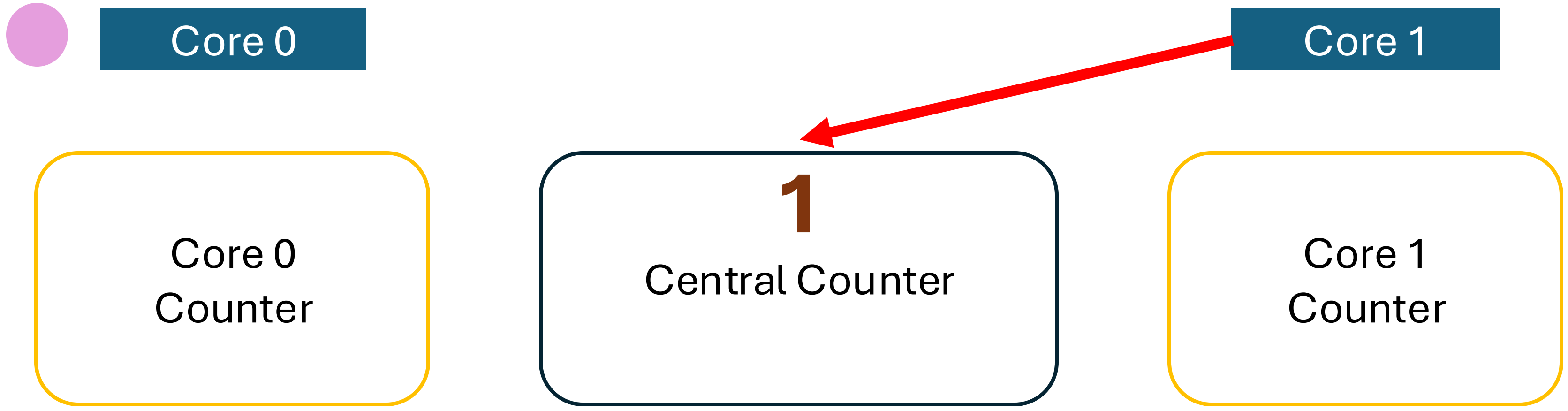
Solution: Sloppy Counters

When a core releases a reference, it'll go back to the local counter.

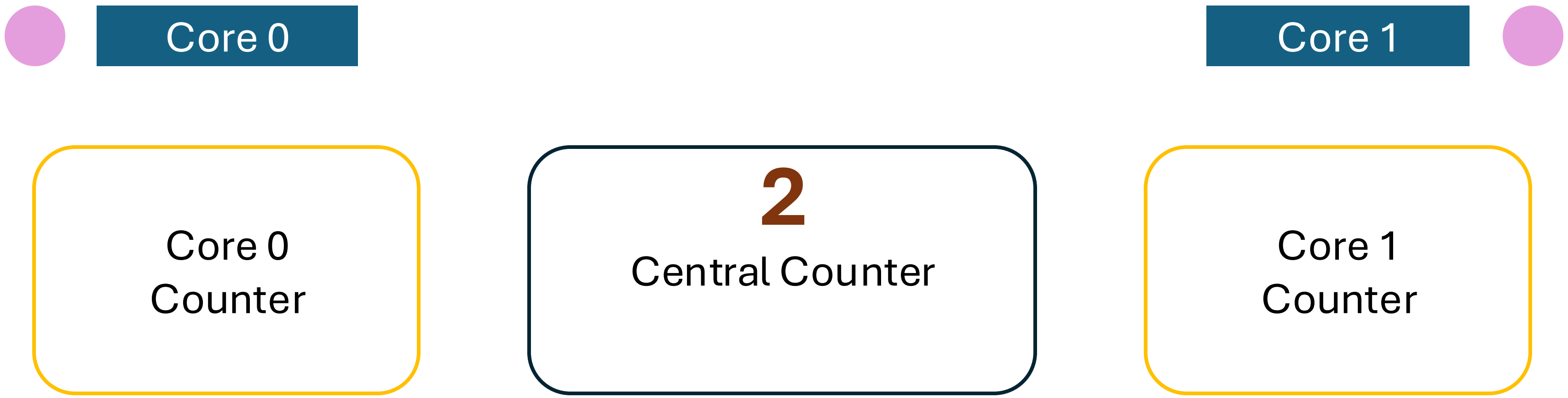


Solution: Sloppy Counters

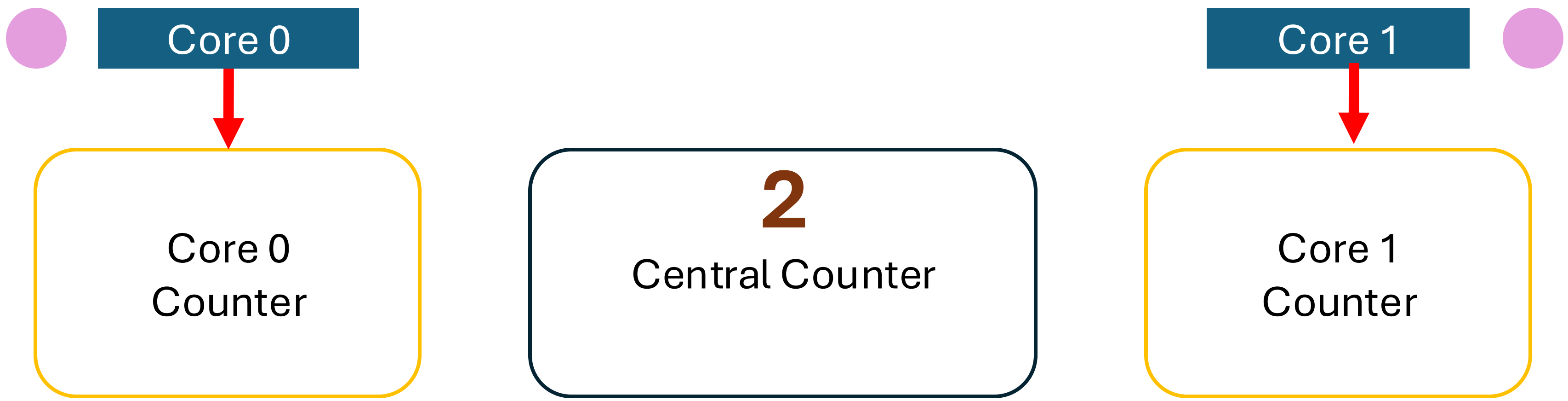
When a core releases a reference, it'll go back to the local counter.



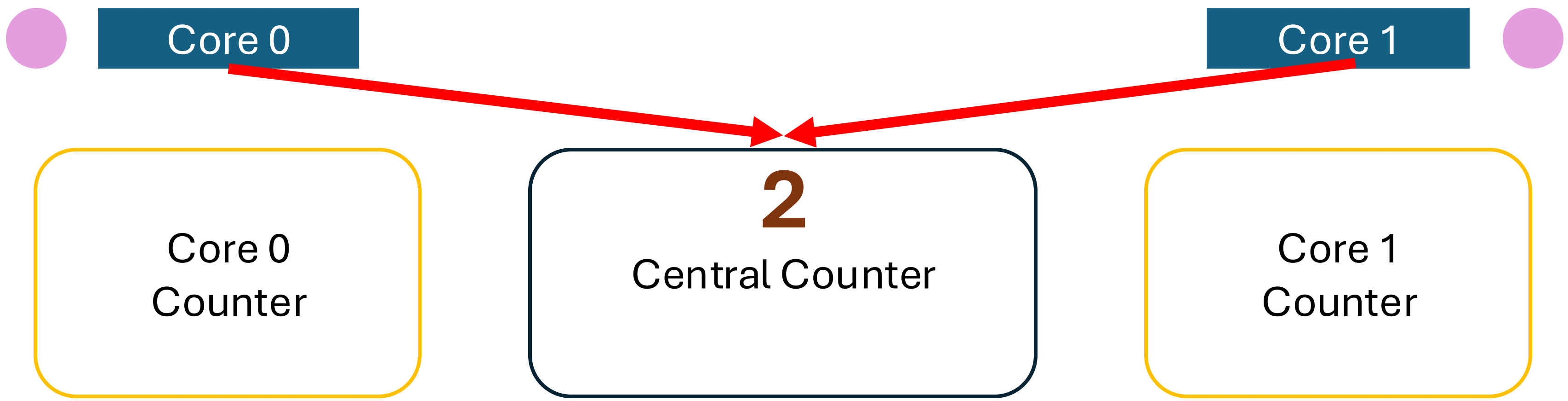
Solution: Sloppy Counters



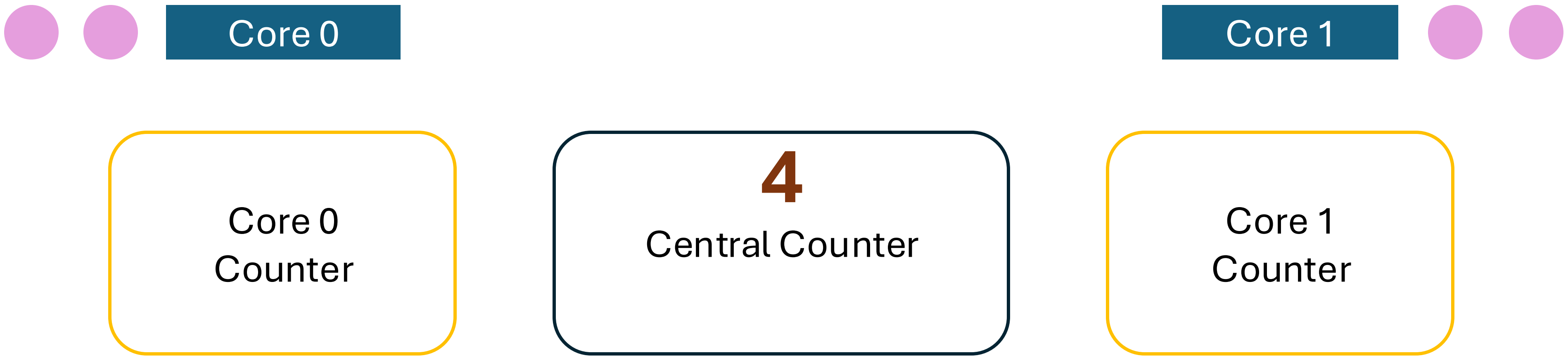
Solution: Sloppy Counters



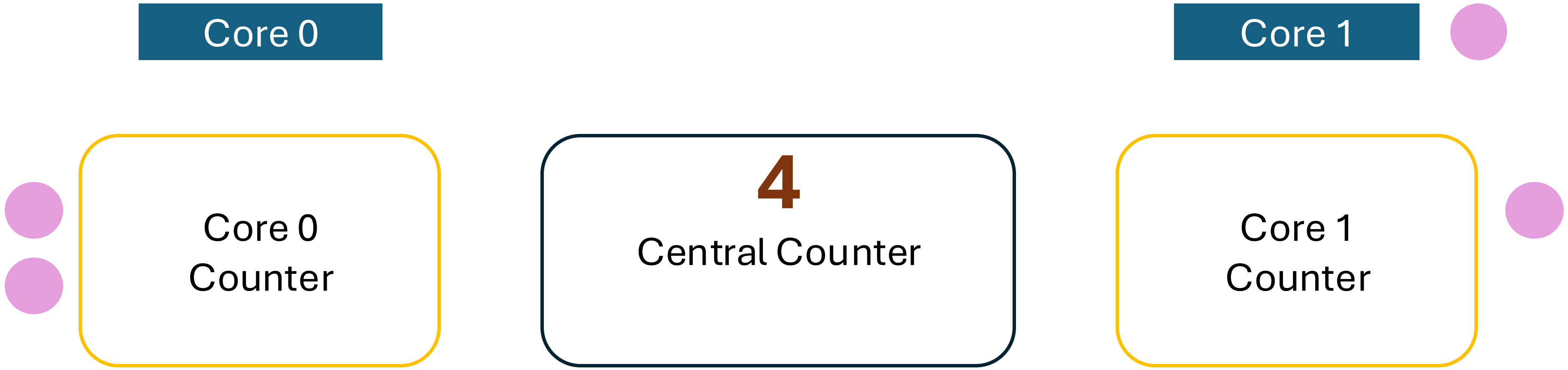
Solution: Sloppy Counters



Solution: Sloppy Counters

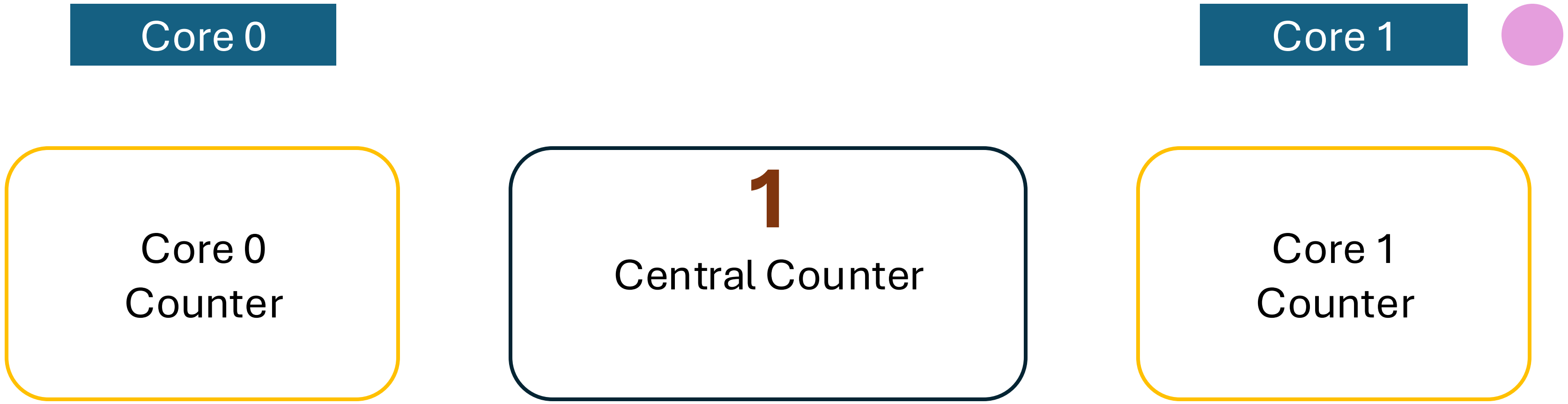


Solution: Sloppy Counters



Solution: Sloppy Counters

Under certain circumstances, the local spare references are released

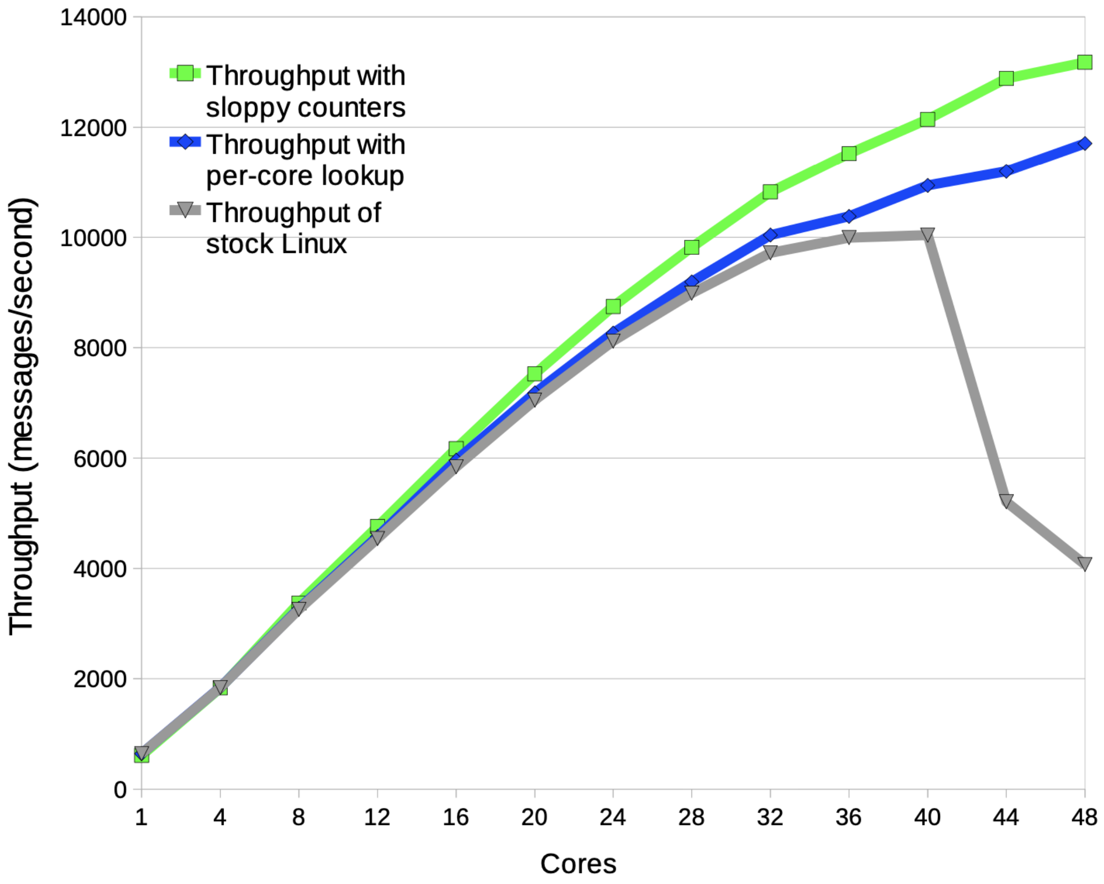


Solution: Sloppy Counters

Advantages of Sloppy Counters include:

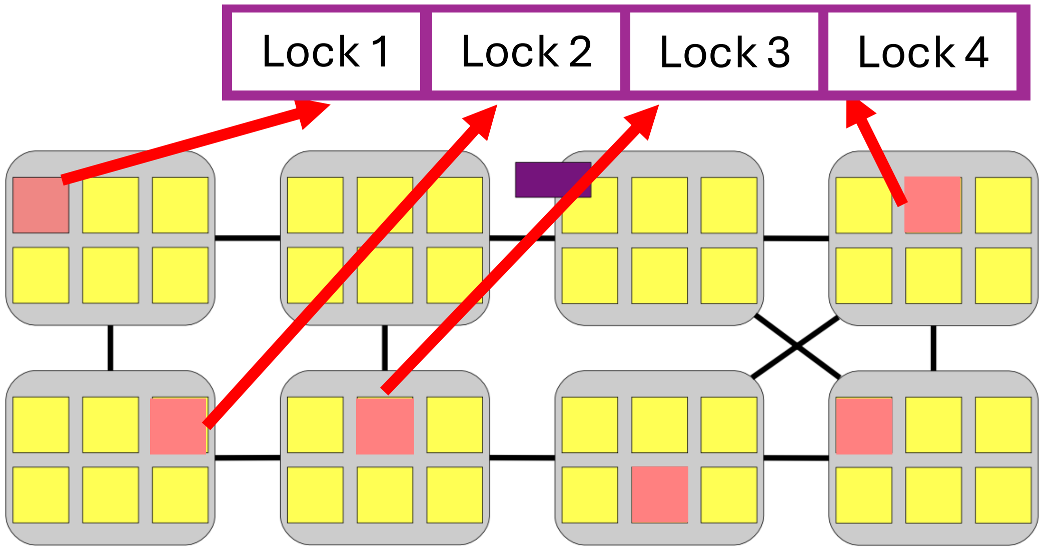
- Simple to use: No need to change application code
- Scale well: No cache misses in common case
- Acceptable memory usage: O(N)

Solution: Sloppy Counters



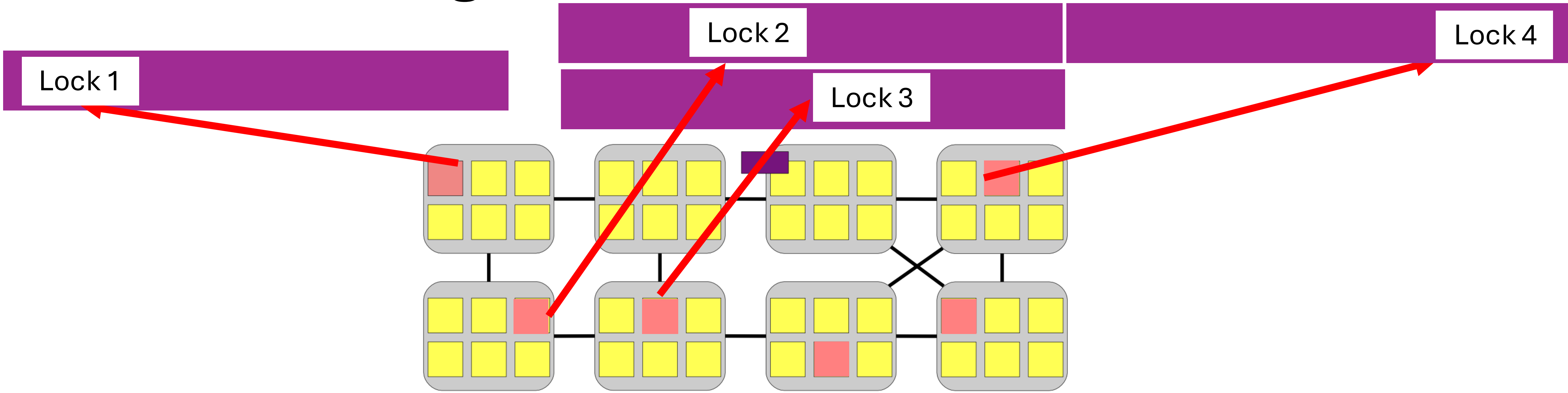
Performance

Corner case: False sharing



The cores are requiring different vars;
These vars happen to **fall into the same cache line**

False sharing solution



Simply split to **different cache lines.**

Discussion

- What’s the common reason behind those bottlenecks?
- Will there be a common solution?

Results and Conclusions

	Memcached	Apache	Exim	PostgreSQL	GMake	P-Searchy	Metis
Mount Table *Per core Caching*		X	X				
File Table *Per core Caching*		X	X				
Sloppy Counter	X	X	X				
inode allocation *Avoid locks*	X	X					
Lock-free dentry *Avoid locks*		X	X				
Super Page *Fewer locks*							X
DMA buffer *Allocate local memory*	X	X					
Network Stack *Avoid false sharing*	X	X		X			
Parallel Accept *Per core Socket queue*		X					
App Modification				X		X	X

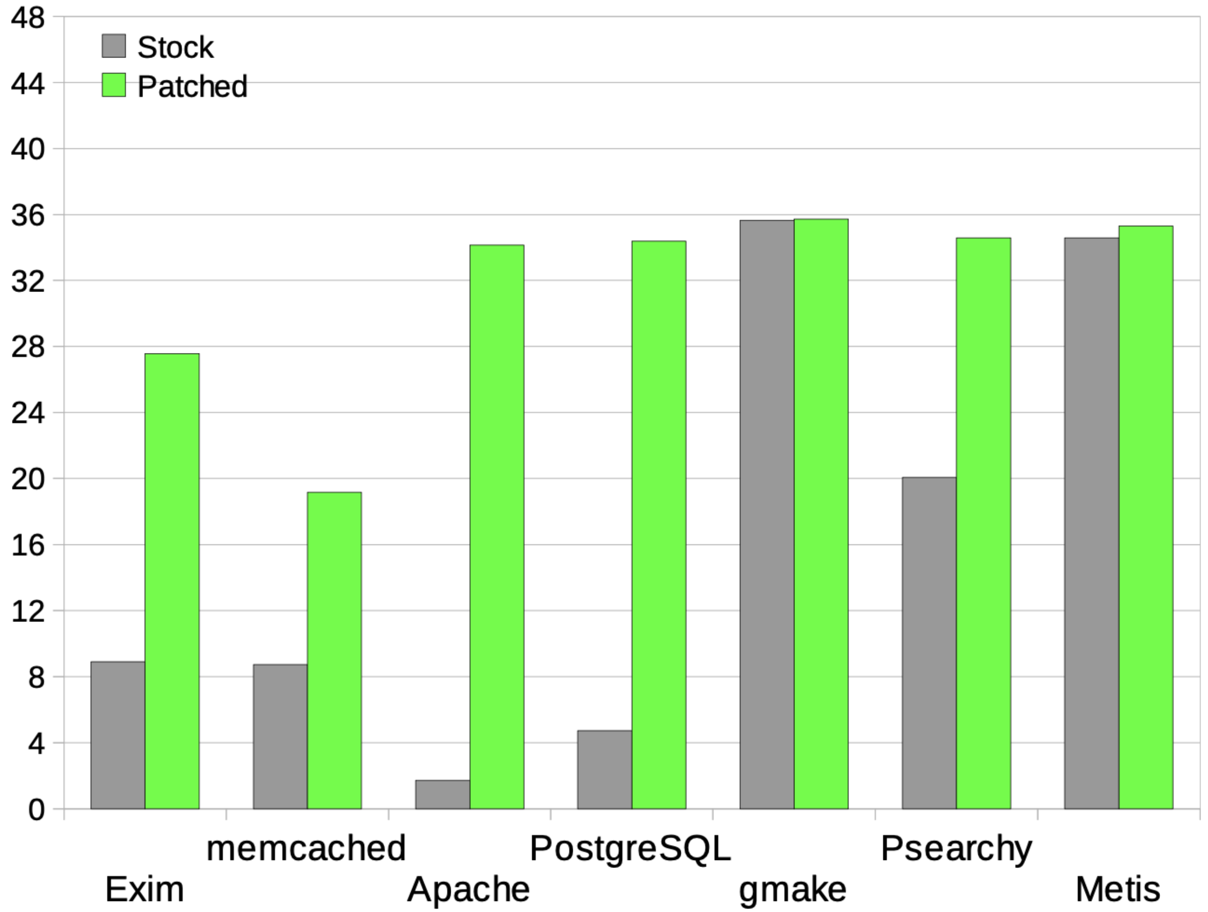
	Memcached	Apache	Exim	PostgreSQL	GMake	P-Searchy	Metis
Mount Table *Per core Caching*		X	X				
File Table *Per core Caching*		X	X				
Sloppy Counter	X	X	X				
inode allocation *Avoid locks*	X	X					
Lock-free dentry *Avoid locks*		X	X				
Super Page *Fewer locks*							X
DMA buffer *Allocate local memory*	X	X					
Network Stack *Avoid false sharing*	X	X		X			
Parallel Accept *Per core Socket queue*		X					
App Modification				X		X	X

Uses a “Generation Counter” to check for modifications during comparison

	Memcached	Apache	Exim	PostgreSQL	GMake	P-Searchy	Metis
Mount Table *Per core Caching*		X	X				
File Table *Per core Caching*		X	X				
Sloppy Counter	X	X	X				
inode allocation *Avoid locks*	X	X					
Lock-free dentry *Avoid locks*		X	X				
Super Page *Fewer locks*							X
DMA buffer *Allocate local memory*	X	X					
Network Stack *Avoid false sharing*	X	X		X			
Parallel Accept *Per core Socket queue*		X					
App Modification				X		X	X

Programs NIC to direct packets to different queues

Performance after changes



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

Performance after changes

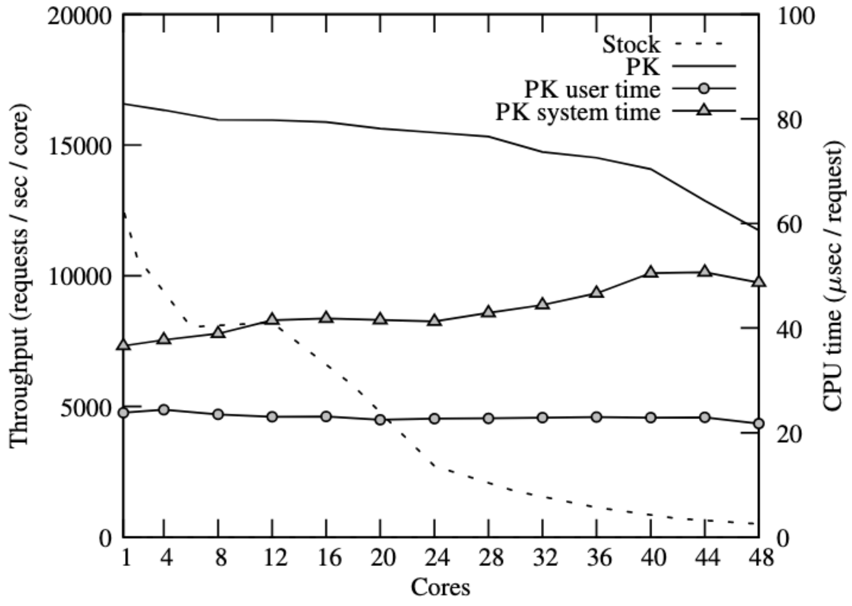


Figure 6: Apache throughput and runtime breakdown.

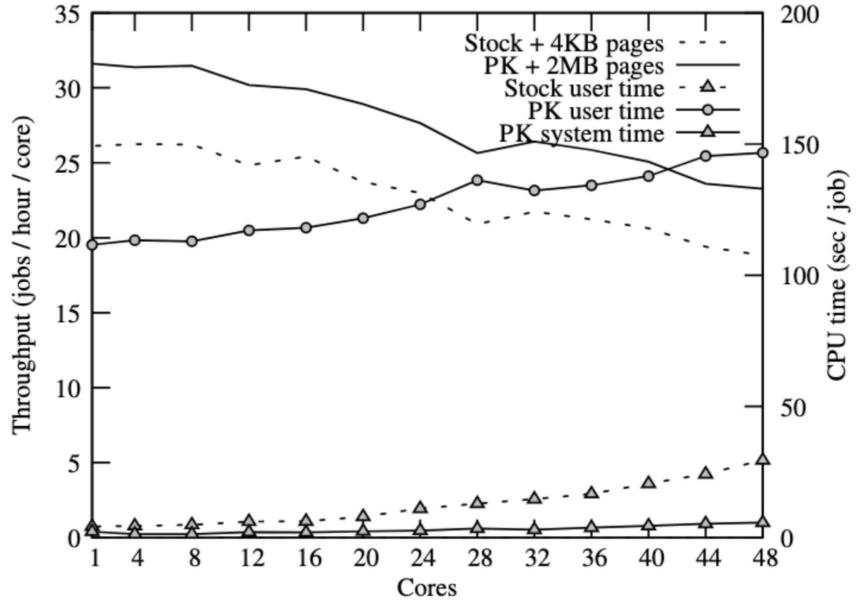


Figure 11: Metis throughput and runtime breakdown.

Conclusion

- Current Linux (2010) is capable for scaling server software, up to 48 cores.
- Some necessary parallel programming techniques need to be applied to kernel / applications.

Discussion

Limitations

- Ignore File System (using RAM disks)
- Limited to 48 cores, a few applications

Limitations

- Ignore File System (using RAM disks)
- Limited to 48 cores, a few applications

- Many applications could be disk/memory bounded!
- How are cores binded and selected (when not using all of them)?
- Will different topology affect the results?
- Will the solutions scale (in theory)?

Next Steps?