Scalability of modern Linux kernels

September 2010

Andi Kleen, Tim Chen

LinuxCon Japan

Agenda

Presentation is about Linux kernel scalability

On "single image" systems

Not applications or clusters

Presentation is about scalability, not performance

Assumes basic knowledge of multi-threaded programming

What is scalability?

More CPU cores added to the system:

- System handles more operations

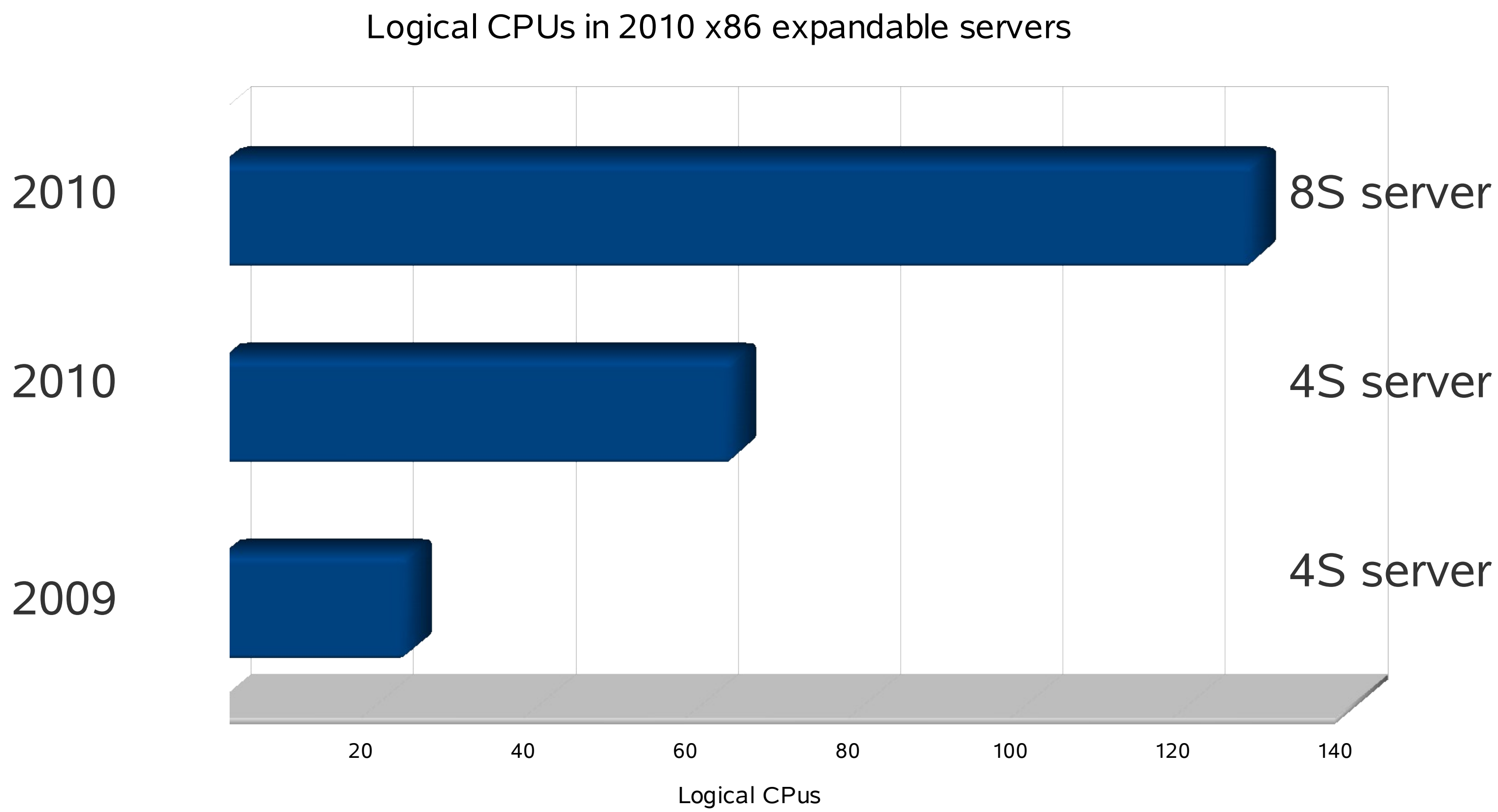
More memory added to the system

- System runs faster

More threads running on a system to use all CPUs:

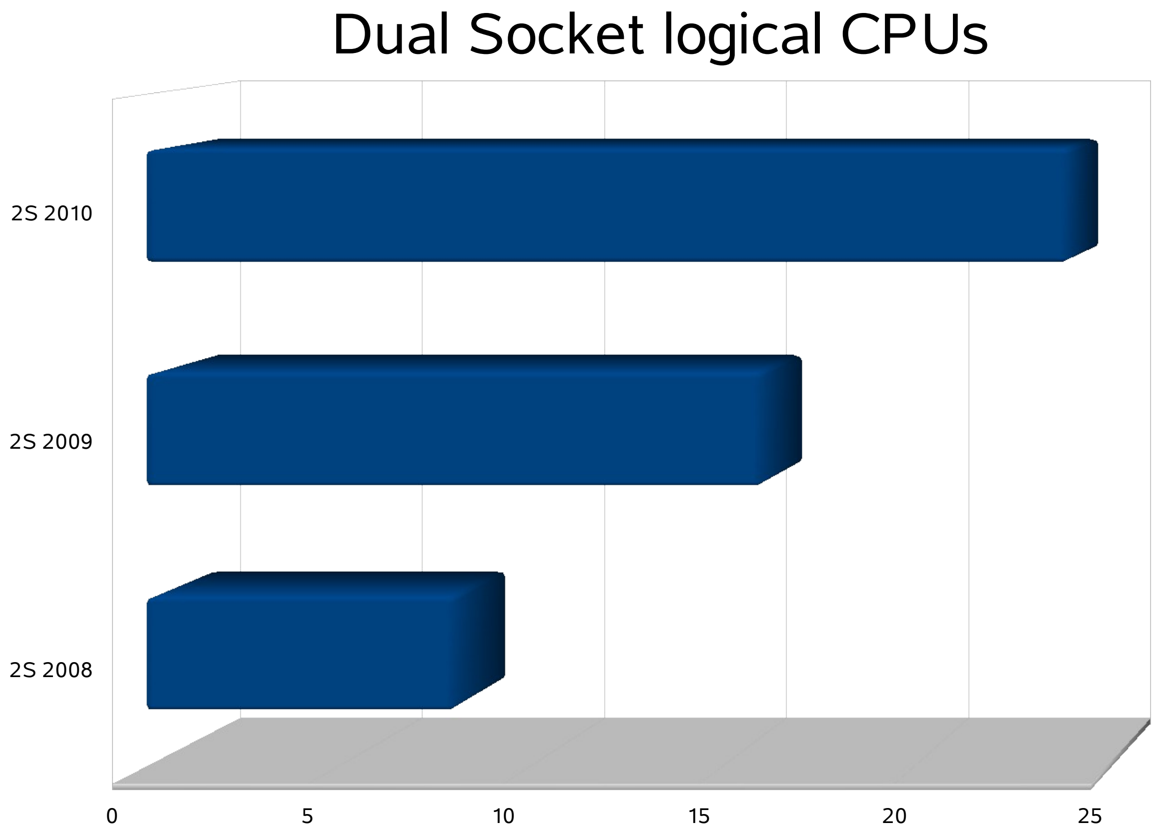
- System does more useful work

The need for software scalability



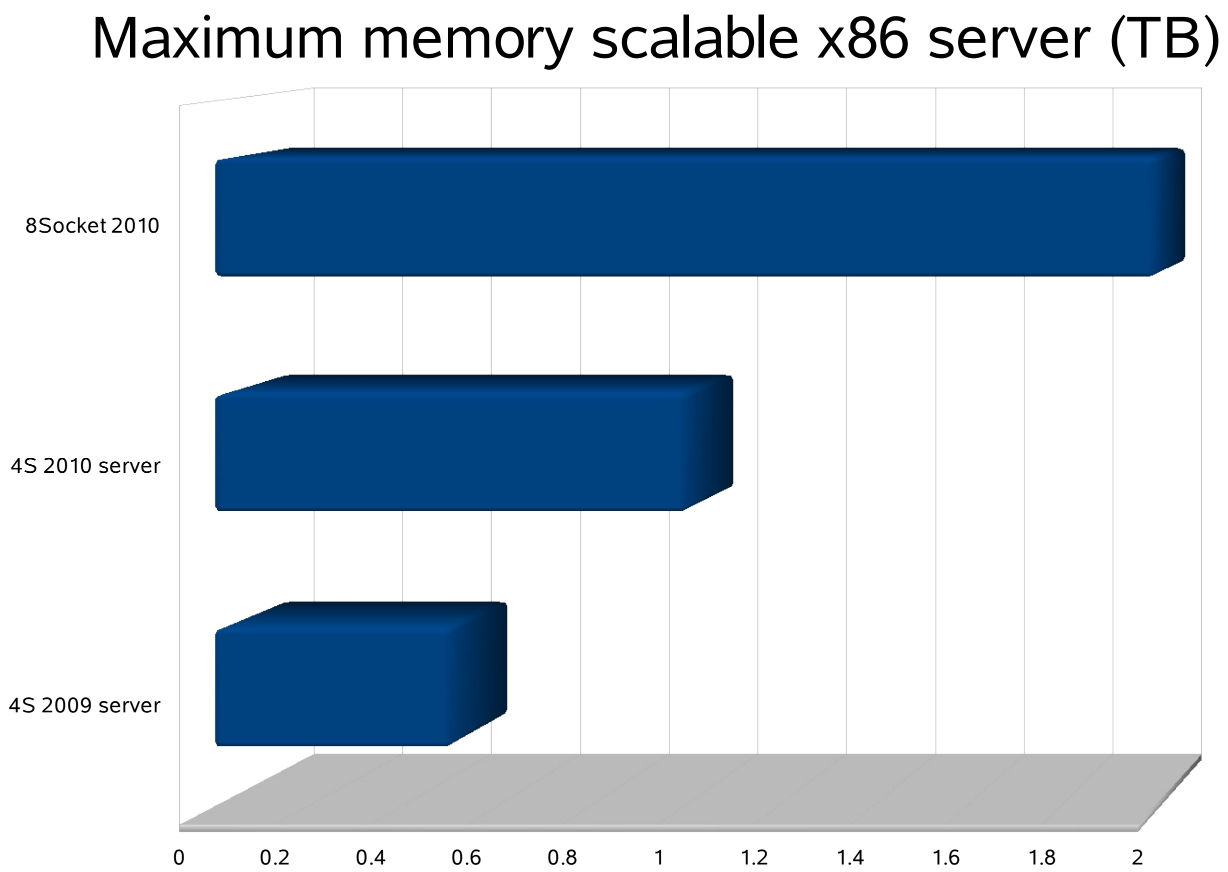
These are standard commercial servers, not super computers

2 socket server CPU trends



Scalability is not just a high-end problem

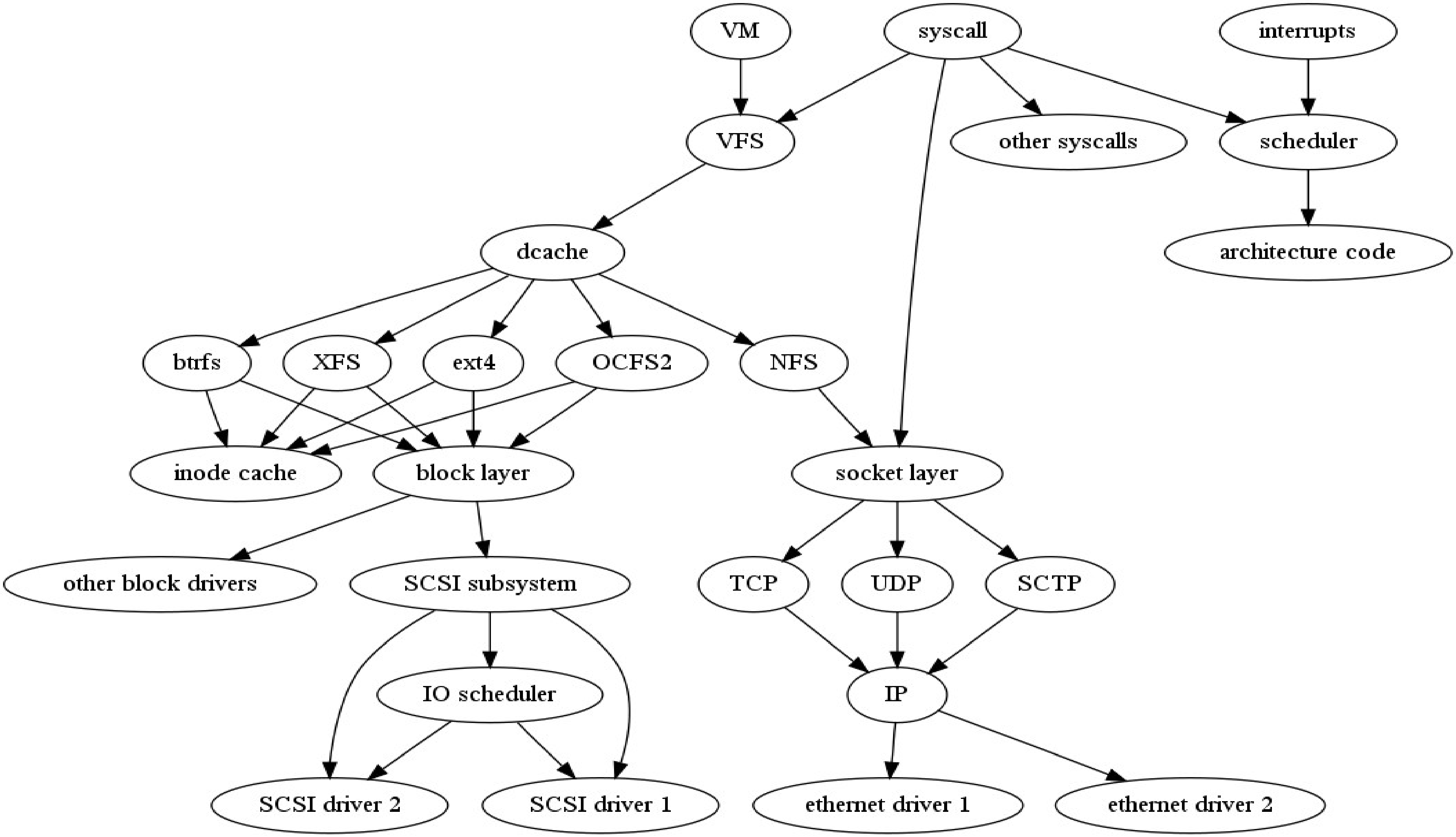
Memory size trends



Memory size scalability is important too

1TB systems available for less than 50000 EUR today

Some subsystems (incomplete)



So does the Linux kernel scale?

Yes it does!

It runs on the largest HPC systems deployed today

- Core of the system is extremely scalable

But actual results depends on the workload

Continuous improvements needed to get better

To how many CPUs does it scale?

It depends how you use it

There is no single number

Depends on: workload, hardware

Kernel scalability crash course crash course

Kernel is a big library essentially

No big data sets, but a lot of parallel operations

- Provides services to application

Key is accessing shared data structures in parallel

Needs fine-grained locking

Scalability disadvantages

More locks is not always faster

- Each atomic operation has a cost

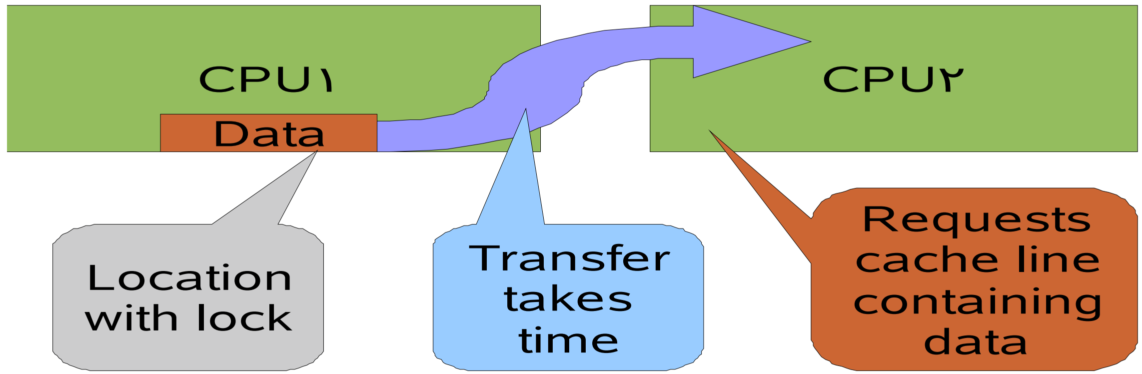
Scalability is hard

- But we're getting better tools

Scalability makes code more complicated

- Trade off against code maintainability
- Some changes are not worth doing

Latencies



- Sharing data that changes is costly
  - This affects locks or reference counts
- It's (usually) about inter inter-core latencies
  - Think of the system as a network
- Contention versus lock bouncing

- Unfair memory a problem on NUMA

Lock data not code

Code locks versus data locks

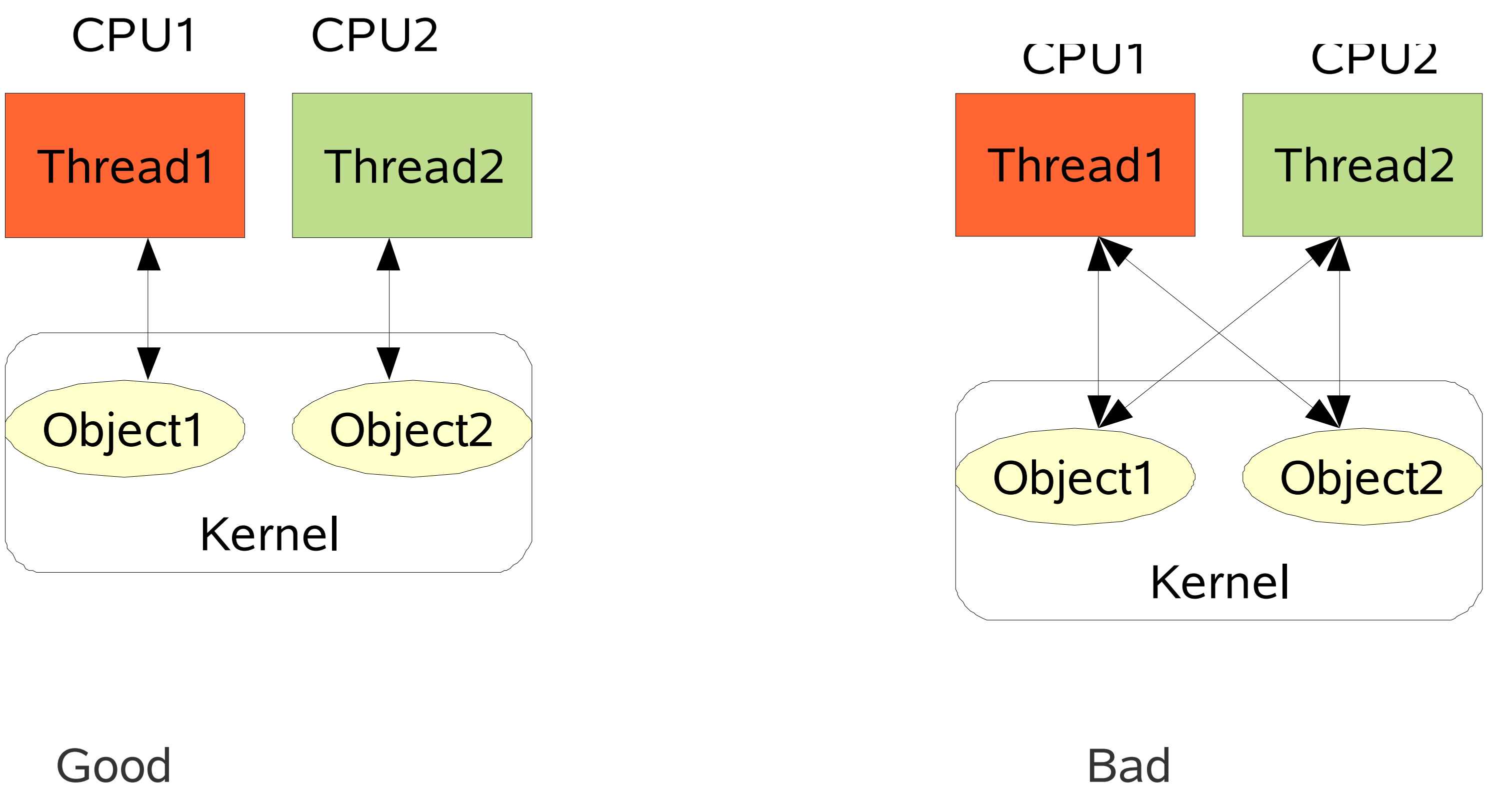
Modern kernels have few code locks

- But still a few critical ones

Reference counts avoid locking

- But still atomic operation

Data Localization



Objects have locks and other state that would need transferring between CPUs

Objects

Objects can be

- network device
- SCSI host
- file
- address space
- socket
- ...

Fix: spread workload to multiple objects

- Kernel improvements in this area ongoing
- But will always be limits

Case study: global lock: dcache_lock

directory cache caches file names in a hash table

dcache_lock is a global lock that protects the directory cache when we update changes to dcache (file or directory create, delete)

- Not for pure reading!

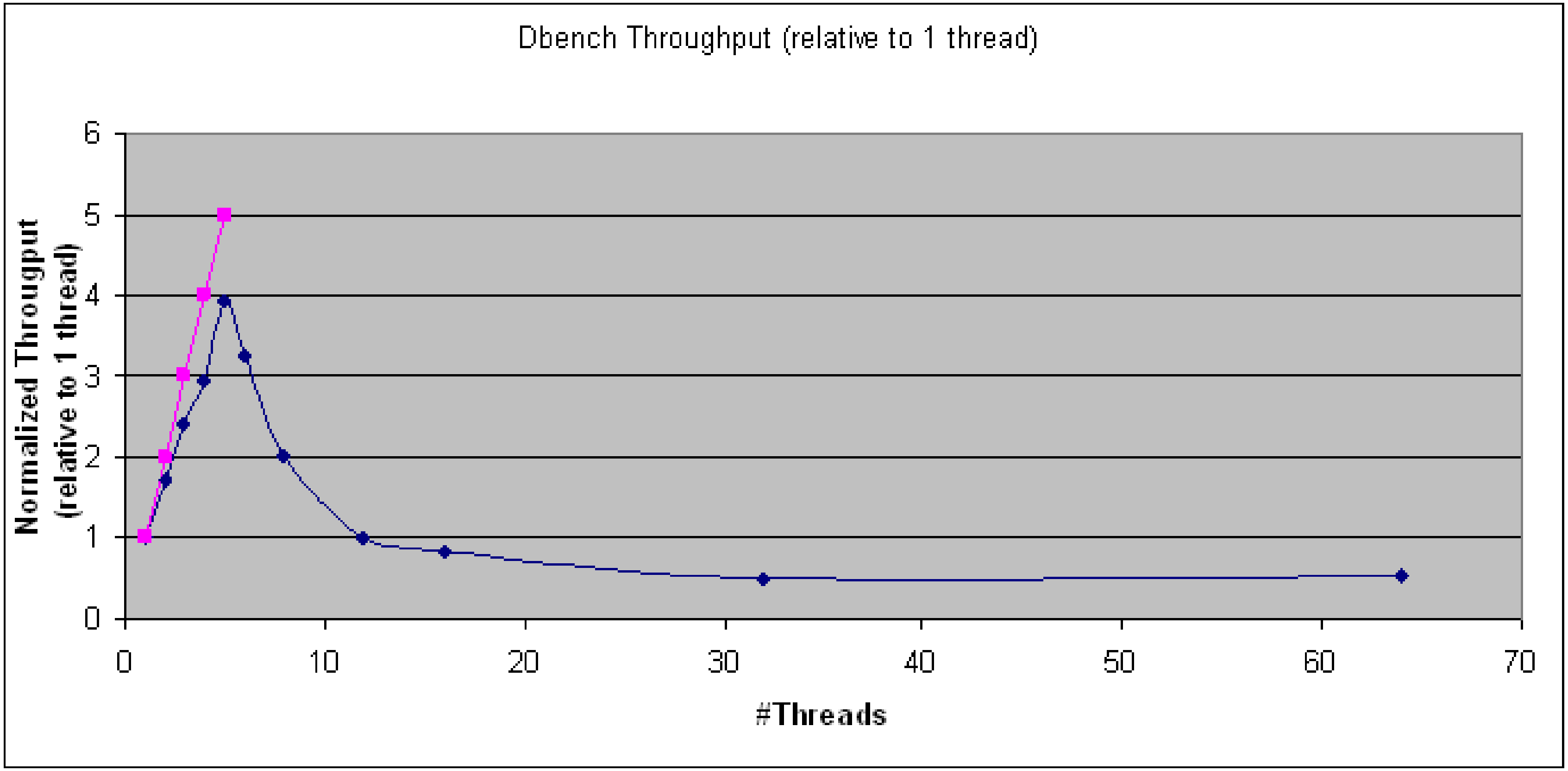
Use Dbench to emulate multiple clients stressing the file system, each doing create, delete, read, write for files.

Disclaimer: dbench not a good benchmark in general to optimize for

- But serves as a load generator here

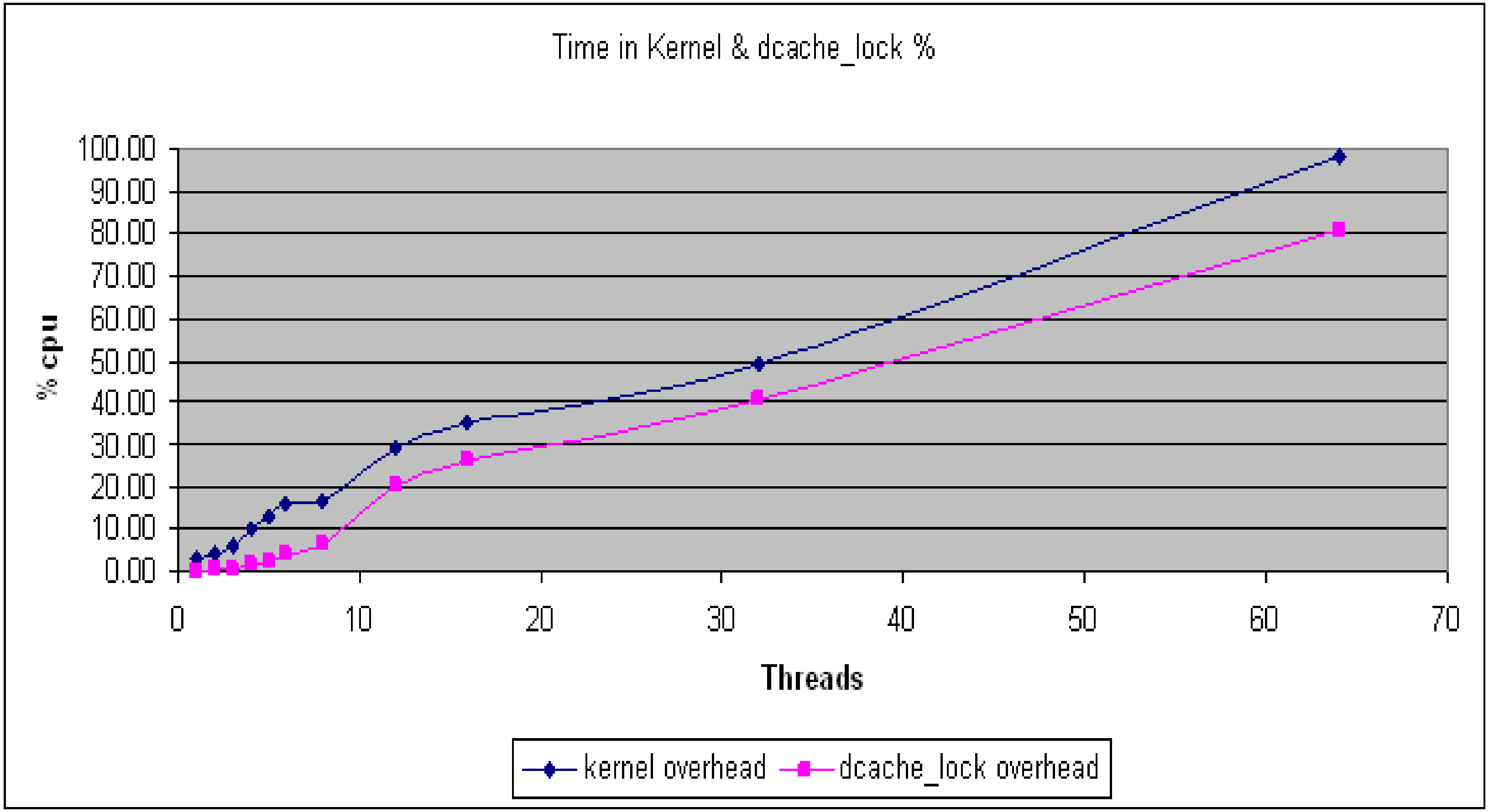
Thanks to Tim Chen for the data

dbench throughput



Workload runs in tmpfs

dcache_lock overhead



Improving the dcache

Requires large changes to get rid of the inode/dcache code locks

Problem large code locks protected a lot of different things

Large patchkit available to fix the VFS locking (N.Piggin)

- This fixes the dcache_lock and inode_lock
- Makes common case faster too due to less reference counting

File systems

- Data IO is (usually) parallel
  - Especially when you preallocate
- Often metadata locking in file system per mount point
  - If a problem use multiple file systems
- Synchronization of writes per file descriptor
- FS performance depends on the application

Filesystems: ext4

ext4 better than ext3 in scalability

- Extents and new algorithms help

Some metadata synchronization, per directory

- For data O_DIRECT is best

Journal threads can be a bottleneck

- Scalability problem in journal locking fixed recently

Filesystems: XFS

- XFS more fine grained locking, good at scalability
  - Can access "allocation groups" in parallel
  - Good parallelism in a file
- Good at large IO, bad at lots of small files (but is improving)
- Ongoing improvements

Filesystems: btrfs

Still rather new and under development

Not much focus on scalability yet

Some locking issues in trees.

Virtual Memory subsystem

In general scales reasonably well with different processes

Some problems with free page management inside NUMA nodes

- zone->lock can be a problem

Scalability to very large memory sizes still work in progress

But has been done in special setups (HPC, large pages)

Address spaces

Single locks protect a process address space
- mmap_sem protects tree of address space mappings

Problems with parallel page faults, parallel brk/mmap in a threaded program
- All threads hit the single address space

Workaround: do less mmaps/unmaps in application
- Such as: tune malloc mapping thresholds

Networking basics

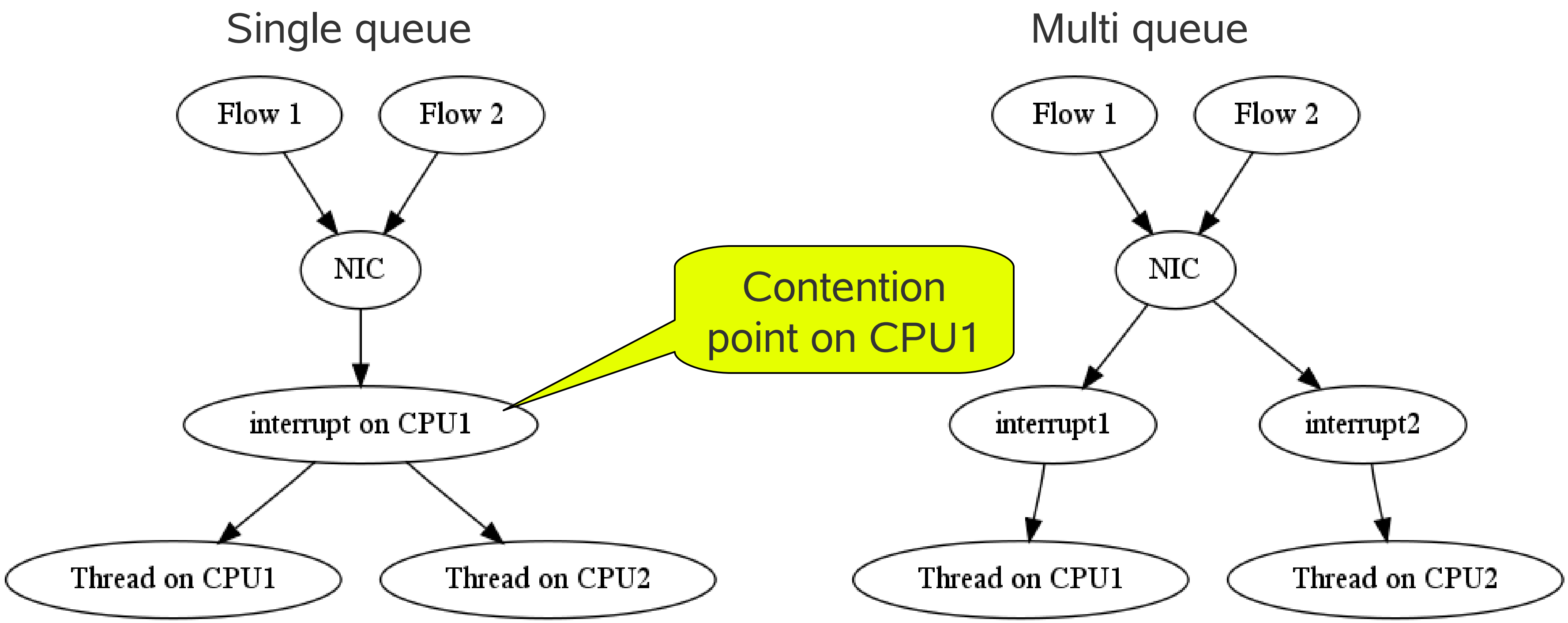
Basic TCP/IP network stack very scalable

No serious locking problems on a global scale

Object locks can be still a problem

Networking multi-queue

Goal: spread network connections to multiple CPUs



Multi queue development in kernel still ongoing
Currently still needs manual tuning through sysfs
NUMA locality can be critical and needs manual tuning too
Older kernels missing multi queue

Scheduler scalability

In principle, scalable: major run queues per CPU

Often algorithmic problems, many regressions on workloads

Real time scheduler not scalable on newer kernel

- Attempts “global” real time fairness
- Some workarounds possible using cpusets

Analysis

System time (is there a problem?)

Scaling tests with increasing thread counts

- Watching system time

Whole system profilers:

- oprofile, perf to analyze kernel behaviour
- Often need callgraphs enabled to see lock caller
- Profiling can be done on short steady states (1min)

Tracers:

- systemtap, ftrace to understand behavior

Summary

Kernel already scales well today

- But work needed to handle more workloads and more cores

Kernel scalability cannot be threated like a black box

- Some areas to be avoided on large systems
- Application tuning can help today to avoid bottlenecks

Questions?

Backup

Kernel scalability history

- 2.0 big kernel lock for everything
- 2.2 big kernel lock for most of kernel
  - Interrupts running independently with own locks
  - First usage on larger systems (16 CPUs)
- 2.4 more fine grained locking, still common global locks
- 2.6 serious tuning, ongoing
  - Redesigned subsystems for scalability
    - multi queue CPU scheduler, multi flow networking, ...
  - Advanced lock-less tuning (Read-Copy-Update, others)
  - 2.6.37: Big Kernel Lock will be (nearly) gone
  - A few problematic code locks left

Read-Copy-Update

Standard lock-less technique for scalability in the kernel

- When a lock is too costly

Uses "quiescent periods" to avoid freeing objects in operation

Allows scaling readers lock less at some cost to writers

- Helps for workloads that read more oft than writing

Writers generally still need locks

Makes code harder to understand

Enterprise distributions

RHEL5

- 2.6.18 based. Already several years old.
- Several known serious scalability issues:
  - VM, single queue networking

RHEL6/SLES11-SP1

- 2.6.32 based
- Many improvements, but still a lot of known issues
- Base of this presentation

Consumer distributions are more bleeding edge

- Fedora, OpenSUSE, ...
- Often scalability regressions, but also improvements

Legal Information

intel®

Leap ahead™