Multiprocessor, Scalability

Haibo Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/haibo_chen

Some slides adjusted from. Elsa L Gunter (UIUC), Jonathan Walpole (PSU)
Paul McKenney (IBM) Tom Hart (University of Toronto)
We are in Multi-core era
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, ...)

Locking

Why do kernels normally use locks?

Locks support a concurrent programming style based on mutual exclusion
  - Acquire lock on entry to critical sections
  - Release lock on exit
  - Block or spin if lock is held
  - Only one thread at a time executes the critical section

Locks prevent concurrent access and enable sequential reasoning about critical section code
Amdahl's Law

\[
\frac{1}{(1 - P)} + \frac{P}{S}
\]
Critical-section efficiency

\[
\text{Critical-section efficiency} = \frac{T_c}{T_c + T_a + T_r}
\]

Ignoring lock contention and cache conflicts in the critical section
Recap: Motivating example: file descriptors

Ideal FD performance graph

Actual FD performance
Recap: Why throughput drops?

Load fd_table data from L1 in 3 cycles.
Recap: Why throughput drops?

Now it takes 121 cycles!
Cause: Non-scalable locks

Non-scalable locks
   Such as spin locks
   Poor performance when highly contended

Many systems are using non-scalable locks

But they are dangerous
Why they are dangerous

Lead to performance collapse when adding a few more cores

Even tiny critical section will also lead to this performance collapse
Case study: ticket spinlock

Normal spinlock has extremely noticeable unfairness
   Even on a 8 core CPU

Ticket spinlock guarantees lock are granted to acquirers in order
   Used in Linux kernel

But it’s non scalable lock
Pseudo code for ticket lock

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

void spin_lock(spinlock_t *l) {
    int t = atomic_xadd(&l->next_ticket);
    while (t != lock->current_ticket)
        ; // spin
}

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

How do current_ticket and next_ticket work together to guarantee mutual exclusion?
Benchmark setup

Hardware
  48 core (8 x 6-core 2.4GHz AMD Opteron)

OS
  Linux 2.6.39

Warm file system cache
  Do not involve disk I/O during benchmarking
Benchmark: FOPS

Create a single file
Starts one process on each core
Each process repeatedly opens and closes the file

Lock in kernel
  per-entry lock in the FS name/inode cache
  when closing a file, acquire the lock, decrease ref cnt
Benchmark: FOPS
Benchmark: MEMPOP

Create one process on each core

Repeat the following

\texttt{mmap} 64KB of memory with \texttt{MAP\_POPULATE} flag

Tell the kernel to allocate page and insert into page table immediately

\texttt{munmap} the memory

Lock in kernel

protects the data structure mapping physical pages to virtual memory regions
Benchmark: MEMPOP

The graph shows the performance profile of the MEMPOP benchmark. The y-axis represents the number of mmaps per second (mmaps/ms), and the x-axis represents the number of cores. The graph indicates an initial peak at around 6 cores, followed by a decline as the number of cores increases, leveling off at higher core counts.
Benchmark: PFIND

Search file in a directory

- evenly divide directories in the search directory as per-core inputs
- search the file using find command on each subdirectory simultaneously
- Input directory is balanced

Lock in kernel

- lock protecting block buffer cache
Benchmark: PFIND
Benchmark: EXIM

A commonly used mail server on Unix system
  Single master process listening for incoming SMTP connections
  Fork a new process for each connection and handles the incoming message

Lock in kernel
  protects the data structure mapping physical pages to virtual memory regions (same as MEMOP)
Benchmark: EXIM

The graph shows the performance of EXIM with respect to the number of cores. The x-axis represents the number of cores, while the y-axis represents messages per second. The graph indicates an initial increase in messages/sec with an increase in cores, peaking around the 36 core mark, after which there is a sharp decline in performance.
Performance collapse

We expect to improve a systems performance by adding more CPU cores

But the performance actually becomes worse when we add more cores above the “threshold”

These are caused by critical section contention
# Critical section details

## Single core measurement

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>cycle/op</th>
<th>#acquire/op</th>
<th>cycles in critical section</th>
<th>% in critical section</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOPS</td>
<td>503</td>
<td>4</td>
<td>92</td>
<td>73%</td>
</tr>
<tr>
<td>MEMPOP</td>
<td>6852</td>
<td>4</td>
<td>121</td>
<td>7%</td>
</tr>
<tr>
<td>PFIND</td>
<td>2099M</td>
<td>70K</td>
<td>350</td>
<td>7%</td>
</tr>
<tr>
<td>EXIM</td>
<td>1156K</td>
<td>58</td>
<td>165</td>
<td>0.8%</td>
</tr>
</tbody>
</table>
Critical section length and collapse

#core to cause performance collapse

<table>
<thead>
<tr>
<th>Critical Section Percentage</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.80%</td>
<td>0.80%</td>
</tr>
<tr>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>73%</td>
<td>73%</td>
</tr>
</tbody>
</table>
Critical section length and collapse

#core to cause performance collapse

Higher percentage of serial section Collapses earlier

Critical section percentage
Why?

Why does the collapse start so early

Why does performance fall so far

Why does performance collapse so rapidly
Background on cache coherence

On multi-core processors, each core has its own private cache.

Need to keep the cache content in sync when some CPU modifies some memory.

Accomplished by cache coherence protocol.
Directory-based cache coherence
Directory-based cache coherence
Directory-based cache coherence

The diagram illustrates the architecture of a directory-based cache coherence system. It consists of multiple processors (P) connected through an interconnection network (Interconnection Network). Each processor is connected to a memory unit and a directory unit. The memory unit contains presence bits and dirty bits for tracking cache coherence.

- **Store**: Represents the process of updating the cache and memory coherence state.
- **Shared**: Indicates the shared state of the cache and memory.

The presence bits and dirty bit mechanisms ensure that cache coherence is maintained across the system, allowing for efficient and consistent data access.
Directory-based cache coherence

Store

Interconnection Network

Memory

Directory

presence bits

dirty bit

Modified
Directory-based cache coherence
Directory-based cache coherence

Diagram showing the interconnection network between memory and directory. The diagram includes symbols for load, probe message, modified, presence bits, and dirty bit.
Directory-based cache coherence
Directory-based cache coherence
A few notes cache coherence

There may be more than a single directory
   Especially for NUMA systems

Interconnect structure affects cache coherence performance

Directory is just one possible implementation
   Snooping is another commonly used approach
Intuition of the collapse

Key point: read with modified cache line have to get data back from the owner

  Coherence message are processed sequentially

Lock holder modifies cache holding the lock

Waiter is trying to read the lock

  They get value of the lock from the lock holder

More readers means the next lock holder need to wait more time to get the lock
Allocate a ticket
read current ticket and spin

```c
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;
}
```
Allocate a ticket
read current ticket and spin

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

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

```c
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```
Allocate a ticket
read current ticket and spin

```c
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;
}
```

cache coherence message
Allocate a ticket
read current ticket and spin

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

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

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

Lock shared by all core
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;
}
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;
}
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;
}

Only lock holder has lock in cache

Lock holder update ticket
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;
}
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!

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

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

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

Previous lock holder notifies next lock holder after sending out N/2 replies

All waiters read the lock
Markov Chain to model ticket lock

State $i$: $i$ cores holding or waiting for the lock

$a_i$: arrival rate of new cores when $i$ cores are contending

$s_{i+1}$: service rate when $i+1$ cores are contending
Arrival rate

Define

\[ a: \text{average time between consecutive lock acquisition on a single core} \]

Without contention, the rate of a single core trying to acquire the lock is \( \frac{1}{a} \)

With \( k \) cores contending, total of \( n \) cores, arrival rate is \( a_k = \frac{(n-k)}{a} \)
Service rate

Define

s: time spent in serial section

c: time taken by the directory to respond to a cache line request

Directory responds to each cache line request in turn
**Service rate**

$k$ requests, time for winner to get the cache line is $ck/2$

Time to process the serial section and transfer the lock to the next holder: $s + ck/2$

- Increases with more cores contending

Service rate: $s_k = 1/(s+ck/2)$

- Decreases with more cores contending
Steady state

Define $P_0, \ldots, P_n$: steady state probabilities of lock being in states 0 through $n$

Steady state: $P_k a_k = P_{k+1} s_k$

Transition rates balances

With lots of math, we can calculate $P_i$ and expected number of cores waiting
Thus get the speed up with the ticket lock
Validating the model

Use the model to predict speed up with different length of critical section

Compare the predication with actual speed up control critical section length with micro benchmarks
400-cycle serial section

- Model, 3.2% serial
- Model, 0.4% serial
- Model, 0.2% serial
- Ticket, 3.2% serial
- Ticket, 0.4% serial
- Ticket, 0.2% serial

Speedup vs Cores
400-cycle serial section

Higher percentage of serial section Collapses earlier
Fixed serial section percentage (2%)
Fixed serial section percentage (2%)
Implications

The collapse of ticket lock is a property of their design

More cores contending

Effectively increase the length of the critical section
Thus increase the probability that another core start contending for the lock
Implications

Collapse only occurs for short critical sections

Service rate $s_k = 1/(s+ck/2)$

When $s$ is small, strongly influenced by $k$

The collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl’s law
Making ticket spinlock scalable

Common way: use proportional back-off

```c
void spin_lock(spinlock_t *l) {
    int t = atomic_xadd(&l->next_ticket);
    while (t != lock->current_ticket) {
        // wait more time with each failure
    }
}
```

Why this would work? (answer this later in questions)
Using scalable locks

Many existing scalable locks

Main idea is to avoid contending on a single cache line

Example

MCS (John M. Mellor-Crummey and Michael L. Scott)

K42
General idea of MCS lock

mcs_lock

NULL
General idea of MCS lock

Use compare and swap to change mcs_lock point to self node

Check previous node
NULL in this case, no need to wait
General idea of MCS lock

previous node is not NULL

Current waiting is 1

Wait until lock holder set waiting to 0
General idea of MCS lock

- Waiting: 0
  - Next: mcs_lock

- Previous node is not NULL
  - Current waiting is 1
  - Wait until lock holder sets waiting to 0

Diagram:
- Node with waiting: 0 and next: NULL
  - Connected to mcs_lock node
  - mcs_lock node has previous node

Summary:
- The lock system maintains a list of waiting nodes.
- Each node has a waiting count and a next node pointer.
- When a node waits, it sets its waiting count.
- The lock holder sets the waiting count to 0 when it's time to release the lock.
General idea of MCS lock

- Previous node is not NULL
- Current waiting is 1
- Wait until lock holder set waiting to 0
Using scalable locks: FOPS
Using scalable locks: MEMPOP
Using scalable locks: PFIND

![Graph showing throughput vs cores for ticket lock and MCS lock](image-url)
Using scalable locks: EXIM

- Ticket lock
- MCS lock

Throughput (messages/sec) vs. Cores
Questions

Why does the performance of ticket locks collapse with a small number of cores?
   You should be able to answer this now

Why back off can make ticket lock more scalable?
   Decrease the contention on the lock cache line
Problem with back-off

Hard to choose the back off time is important and difficult

May penalize non contended case

May confuse smart hardware which detects busy loops to enter power saving mode
Questions

Problems with the benchmark
   Only one real world application
   Collapses near #40 cores

Why the Linux kernel does not use the scalable locks?
   They perform worse with small number of cores and no contention
   The ultimate solution is to avoid contention instead of using scalable locks
Non-scalable locks are dangerous
   Short critical section may lead to performance collapse
   Caused by contention on lock cache line

Scalable locks is a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel
Thanks

• Next class
  – Concurrency bugs, race detection