Tornado
Why Tornado?

Existing OS designs do not scale on shared memory multiprocessors!
Locality – Traditional System

Memory latency was low: fetch–decode–execute without stalling
Memory faster than CPU = okay to leave data in memory
Locality – Traditional System

Memory got faster, but CPUs got even faster than memory. Memory latency became high. Fast CPUs became worthless.
Locality – Traditional System

Memory access speed became critical
Cache

```
CPU --> cache --> Shared Bus --> Shared memory
```

- Fast memory close to CPU
- Locality of reference
Cache

We like repeated access to the same page of memory!
Memory Latency

CPU executes at speed of cache, not memory!
Locality

Locality:
- same memory location accessed frequently

Temporal Locality:
- same memory location repeatedly accessed within a small duration

Spatial Locality:
- memory location accessed by local CPU only
Caches

All CPUs accessing the same memory location
Shared Counter Example

Copy of counter exists in both processor’s cache

Shared by both CPU-1 & CPU-2
Shared Counter Example

CPU-1

Memory

CPU-2

0
Shared Counter Example

CPU-1

0

Memory

0

CPU-2
Shared Counter Example

- CPU-1: 1
- CPU-2: 1
- Memory: 1

Write in exclusive mode
Shared Counter Example
Shared Counter Example

1. Invalidate

2. From shared to exclusive
Shared Counter Example

Terrible Performance!
What Went Wrong?

Counter bounces between CPU caches leading to high cache miss rate
- CPU speed set by memory speed not cache

A better idea?
- Convert counter to an array
- Each CPU has its own counter
- Updates can be local
- Reads need all counters (add up)
- Depends on the commutativity of addition
Array-Based Counter Example

Array of counters, one for each CPU

Memory

CPU-1

CPU-2
Array-Based Counter Example
Array-Based Counter Example
Array-Based Counter Example

CPU-1
1

CPU-2
1

CPU
Add All Counters
(1 + 1)

Memory
1 1
Array-Based Counter Performance

Performance is no better!
Why Didn’t it Work?

Data is transferred between main memory and caches in fixed sized blocks, called *cache lines*

If two counters are in the same cache line, they can only be used by one processor at a time for writing

- It still has to bounce between CPU caches!
False Sharing

CPU-1

Memory

0,0

CPU-2
False Sharing
False Sharing

CPU-1
0,0

Sharing

CPU-2
0,0

Memory
0,0
False Sharing

CPU-1

1,0

Memory

1,0

CPU-2

Invalidate
False Sharing

CPU-1
1,0

Sharing

CPU-2
1,0

Memory
1,0
False Sharing

CPU-1

Invalidate

CPU-2

Memory
A Better Approach

Pad the array
- leave unused memory space between counters
- force them into independent cache lines

But doesn’t this waste memory ... cache space ... and TLB entries ... etc?

Isn’t this bad for locality?
Padded Array Example

Individual cache lines for each counter
Padded Array Example

Updates independent of each other
Padded Array Performance

Works better
Write Sharing With Caches

If all CPUs access the same memory location, sharing is very destructive, if they write
Shared Counter Example

Minimize read/write & write/write sharing
  - reduces cache coherence protocol activity!
  - i.e., cuts out unnecessary communication

Ideally we should remove all false sharing
What Has Changed?

In the old days locality was a good thing
Now locality seems to be a bad thing
- if you share data, you communicate
- communication is expensive!

Traditional OS implemented to have good locality
- on new architectures this causes cache interference
- more CPUs make the problem worse!

Ideally, CPUs should run from their own cache with no interference from others
Modularity

How can we minimize write sharing & false sharing? How can we structure a system so that CPUs don’t share the same memory?

Split data and encapsulate it
- each CPU should access a separate instance

The Tornado approach:
- Object oriented design
- Clustered objects
- Existence guarantees
Goal

Ideally:

- Cache hit rate should be high
- CPU should use data from its own cache
- CPU’s locality of reference should be good (it frequently accesses its cached data)
- Sharing is minimal
- Cache contents stay in cache and are not invalidated by other CPUs
Object Oriented Approach

Goals:
- Minimize access to shared data structures
- Minimize need for shared locks

Operating systems service application requests for virtual resources
- requests for different virtual resources must be handled independently
- resources are objects
- each instance of a resource is a separate object instance
- fine-grain locking: a lock per instance
Coarse vs. Fine Grain Locking

Process object: maintains the list of mapped memory regions in the process’s address space

On a page fault, it searches the process table to find the responsible region to forward page fault to

Whole table gets locked even though it only needs one region. Other threads get queued up
Coarse vs. Fine Grain Locking

Thread 1

Use individual locks

Thread 2

Region 1
Region 2
Region 3
...
Region n
Memory Management Objects

If file data not cached in memory, request new phy page frame

If file data is cached in memory, address of corresponding physical page frame is returned to Region which makes a call to the Hardware Address Translation object
Advantages of Object Orientation

Performance critical case of in-core page fault (TLB miss fault for a memory resident page)

- Objects invoked specific to faulting CPU
- Locks acquired & data accessed are internal to the object
- In contrast, many Operating Systems use a global page cache or single HAT layer
- These must be locked and both data and locks are sources of contention
Clustered Object Approach

Object Orientation is good, but some objects (resources) are still too widely shared

- Thread dispatch queue (ready list)
- Use of a single list causes memory and lock contention
- Solution: Partition the queue and give each processor its own private list

These private lists represent the thread dispatch queue

- together they form a clustered object
- each private list is a clustered object representative
**Clustered Objects**

- Actually made up of several component objects. Each rep handles calls from a subset of processors.
- Looks like a single object.
- All clients access a clustered object using a common clustered obj reference.
- Each call to a Clustered Obj automatically directed to appropriate local rep.
Clustered Objects

It looks like a shared counter

But actually made up of representative counters that each CPU can access independently

Shared counter Example
One rep for the entire system
Cached Obj Rep – Read mostly.
All processors share a single rep.

One rep for a cluster of Neighbouring processors
Region – Read mostly. On critical path for all page faults.

One rep per processor
FCM – maintains state of pages of a file cached in memory. Hash table for cache split across many reps
Clustered Objects
Clustering Through Replication

One way to have shared data and scale well:
- Only allow reads!
- Replicate data
- All CPUs read their own copy from cache

What about writing?
- Challenge: how to maintain consistency among replicas?
- Fundamentally the same problem as cache coherence
- May be able to do better by taking advantage of the semantics of the object being replicated
- Example: shared counter
Advantages of Clustered Objects

Internally supports replication, partitioning and locks
- all essential for scalable performance on SMMP

Built on object oriented design
- No need to worry about location or organization of objects
- Just use clustered object references

Allows incremental implementation
- Change degree of clustering based on performance needs
- Initially one rep serving all requests
- Optimize if widely shared
Clustered Object Implementation

Each processor has its own translation tables

Clustered Obj 1 | Ptr
---|---
Clustered Obj 2 | Ptr
... | ...

Translation table

Clustered object reference is just a pointer to the local translation table

Pointer to rep responsible for handling method invocations for local processor
Clustered Objects

All clients access a clustered Object using a common clustered obj reference.

Each call to a Clustered Obj automatically directed to appropriate local rep.

Actually made up of several component objects. Each rep handles calls from a subset of processors.

Looks like a single object.
Clustered Object Implementation

Each ‘per processor’ copy of the table is located in the same virtual address space.
One pointer into table will give rep responsible for handling invocation on each processor.
Clustered Object Implementation

We do not know which reps will be needed

Reps are created on demand, on first access

1. Each clustered object defines a miss handling object
2. All entries in the translation table are initialized to point to a *global* miss handling object
3. On first invocation, *global* miss handling object called
4. *Global* miss handling object saves state and calls clustered *object’s* miss handling object
5. *Object* miss handler:
   - if (rep exists)
     - pointer to rep installed in translation table
   - else
     - create rep, install pointer to it in translation table
Counter as a Clustered Object

CPU

Object Reference

Counter – Clustered Object

Rep 1

Rep 2
Counter as a Clustered Object
Counter as a Clustered Object

Updates are independent of each other
Counter as a Clustered Object
Counter as a Clustered Object

Add All Counters (1 + 1)
Synchronization

Locking
- concurrency control for managing accesses to shared data

Existence Guarantees
- Ensure the data structure containing a variable is not deallocated during an update
- Concurrent access to shared data structures requires existence guarantee
Locking

Lots of overhead
- Basic instruction overhead
- Extra cache coherence traffic due to write sharing of lock

Tornado: Encapsulate all locks within individual objects
- Reduces the scope of a lock
- Limits contention
- Split objects into multiple representatives to further reduce contention!
Existence Guarantees

Scheduler’s ready list

Thread 1: Traversing the list
Thread 2: Trying to delete element 2

Element 2

Garbage collected
Memory possibly reallocated

Thread Control Blocks
Existence Guarantees

How to stop a thread from deallocating an object being used by another?

Semi-automatic garbage collection scheme
Garbage Collection Implementation

- **Temporary**
  - Held privately by single thread
  - Destroyed when thread terminates

- **Persistent**
  - Stored in shared memory
  - Accessed by multiple threads
Clustered Object Destruction

Phase 1:
- Object makes sure all persistent references to it have been removed (lists, tables etc)

Phase 2:
- Object makes sure all temporary references to it have been removed
- Uniprocessor: Number of active operations maintained in per-processor counter. Count = 0 (none active)
- Multiprocessor: Clustered object knows which set of processors can access it. Counter must become zero on all processors (circulating token scheme)
IPC

Acts like a clustered object call that crosses from protection domain of client to that of server and back

Client requests are always serviced on their local processors
Processor sharing: like handoff scheduling
Performance

Test machine:
- 16 processor NUMAchine prototype
- SimOS Simulator
Component Results

- Performs quite well for memory allocation and miss handling
- Lots of variations in garbage collection and PPC tests
- Multiple data structures occasionally mapping to the same cache line on some processors

Avg no. of cycles Required for ‘n’ threads
Component Results

Avg cycles required on SimOS with 4-way associative cache

4-way associativity: A cache line from main memory can reside in any one of the 4 slots
Conclusion

Tornado is highly scalable

First generation:
- Hurricane (coarse grain approach to scalability)

Second generation:
- Tornado (University of Toronto)

Third generation:
- K42 (IBM & University of Toronto)
References

- Tornado: Maximizing Locality and Concurrency in a shared-memory multiprocessor operating system by Benjamin Gamsa, PhD thesis, University of Toronto, 1999
- PowerPoint diagrams, earlier slides from Anusha Muthia