Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System

Ben Gamsa, Orran Krieger, Jonathan Appavoo, Michael Stumm

[Department of Electrical and Computer Engineering, University of Toronto – 1999]

Josh Dorothy
CS533, Winter 2010
Why Tornado? What’s the problem?

- Traditional multiprocessor systems are designed around a set of architecture expectations

- But those expectations have changed!
Introduction

What’s the solution?

- We can no longer ignore locality

- Expensive/prohibitive to retrofit existing ‘traditional’ OSs

- Enter Tornado
Cycles per update of a counter log scale

Gamsa et al., Figure 1
Tornado is a shared memory multiprocessor operating system designed to maximize spatial and temporal locality.

To do this, it uses an object oriented approach.

Three additional optimizations:
- Clustered objects
- Protected Procedure Call (PPC)
- Locking/garbage collection optimizations
One: Object-Oriented Structure

- Principle: requests to different virtual resources should be handled independently
  - E.g. – no shared locks

- Tornado handles different processes with logically different objects

- This means that each process has no sources of contention
Key memory management object relationships in Tornado

Gamsa et al., Figure 3
Tornado can map independent resources to independent objects, but what about widely-shared ones (e.g. the FCM)?

Replication, distribution, partitioning of objects: the clustered object
A clustered object is an abstraction of multiple component objects.

Each component object is a representative of some subset of processors.

The number of representatives depends on the degree of clustering.
Clustered Objects (continued)

Several benefits to clustered objects:

1) Allows multiprocessor optimizations, like replication or partitioning of locks and data structures

2) Abstracts the object reference for programmers

3) Enables incremental optimization
Clustered Objects: Implementation

- Key: per-processor translation tables

- Each CPU’s table contains a pointer to the representative responsible for handling an object on that processor

- Each CPU’s table is located at the same virtual memory location – one call to an object will point to different locations for different processors!
It’s immediately known which reps will be needed and where, so we need miss handling.

Each object has a miss handler object, which is called by a global miss handler.

If needed, the miss handler installs the proper pointer in the translation table and the call is restarted with that pointer.
For the clustered object approach to be effective, it needs processor-local memory allocation.

Pools of free memory are divided into clusters to increase locality.

To avoid false sharing, a separate per-processor pool of small blocks is allocated.
Synchronization

Two kinds of locking issues in most systems:

1) **Locking** – concurrency control with respect to data

2) **Existence guarantees** – ensuring the data structure containing the data is not deallocated prematurely
Synchronization – Locking

- Key problem: overhead

- Solution: encapsulate locks in clustered objects

- Reduces lock scope and contention

- Partitioning locks among representatives reduces scope/contention even further
Synchronization – Existence Guarantees

- Traditional locking encourages holding locks for long periods of time (e.g. handling entire page fault)

- Tornado: data structures’ existence guaranteed by the garbage collection scheme

- Simplifies locking protocol, sometimes even eliminating locks
Garbage Collection

- Key: distinguish between temporary references and persistent references

- Temporary: clustered references held privately (e.g. thread stack); destroyed when thread terminates

- Persistent: shared memory, can persist beyond lifetime of a thread
Garbage Collection

Collection done in three phases:

1) Ensure all persistent references removed
2) Ensure all temporary references removed
3) Destroy object itself
Step 2:

- **Uniprocessor case:**
  - Number of active calls to server from external client thread stored in counter
  - When counter is zero, we know there are no more temporary references to any object on that processor

- **Multiprocessor case:**
  - As above, but we need to track other processors too
  - Set: union of processors with objects that reference target object
  - Step through each CPU in set, waiting for counter to be zero
Tornado: microkernel OS that relies on client/server communication

Need to extend locality and concurrency to this communication as well to achieve desired performance

Solution: protected procedure call
Three: Protected Procedure Call (PPC)

- A call from a client object to server object is basically a clustered object call that crosses protection domains

- Key advantages:
  1) Client requests always serviced on their own CPU
  2) Clients and servers share the CPU similar to handoff scheduling
  3) There are as many threads of control in the server as there are client requests
Three: Protected Procedure Call (PPC)

- Client state that would normally be maintained by the server is stored with the client instead, reducing overhead.

- PPCs are generated via stubs based on the public interface of the clustered object.

- After passing a security check, they are made directly to local rep of the target object, providing same locality as IPC.
Three: PPC Implementation

- A PPC is really just a trap and several queue manipulations
- A worker thread is removed from the set of ready workers and the caller thread is enqueued on the worker
- Registers are used to pass parameters
Performance Results

Test machine:

- 16 processor NUMAchine prototype
- SimOS simulator
Component Results
Average number of cycles required for n threads – Tornado on NUMAchine

Gamsa et al., Fig. 8(a)
Average number of cycles required for n threads – Tornado on SimOS

Gamsa et al., Fig. 8(c)
Microbenchmarks
Slowdown relative to uniprocessor case – Tornado on NUMAchine

Gamsa et al., Fig. 9(a)
Slowdown relative to uniprocessor case – Tornado on SimOS

Gamsa et al., Fig. 9(c)
Microbenchmarks – System Comparison
Multithreaded Tests – System Comparison, page faults

Gamsa et al., Fig. 10(a)
Multithreaded Tests – System Comparison, fstats

Gamsa et al., Fig. 10(a)
Multithreaded Tests – System Comparison, threads creations

Gamsa et al., Fig. 10(a)
Summary of Results

- Tornado is highly scalable
- Tornado’s scalability compares very favorably to the other tested systems (100x slowdown on 16 processors)
Conclusion

- Object-oriented nature allows exploitation of locality and concurrency

- Tornado’s abstractions incur overhead, but these are low compared to performance advantages

- Tornado scales extremely well and achieves high performance on shared-memory multiprocessors