

Multi-threaded Performance Pitfalls

Ciaran McHale

License

Copyright © 2008 Ciaran McHale.

Permission is hereby granted, free of charge, to any person obtaining a copy of this training course and associated documentation files (the "Training Course"), to deal in the Training Course without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Training Course, and to permit persons to whom the Training Course is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Training Course.

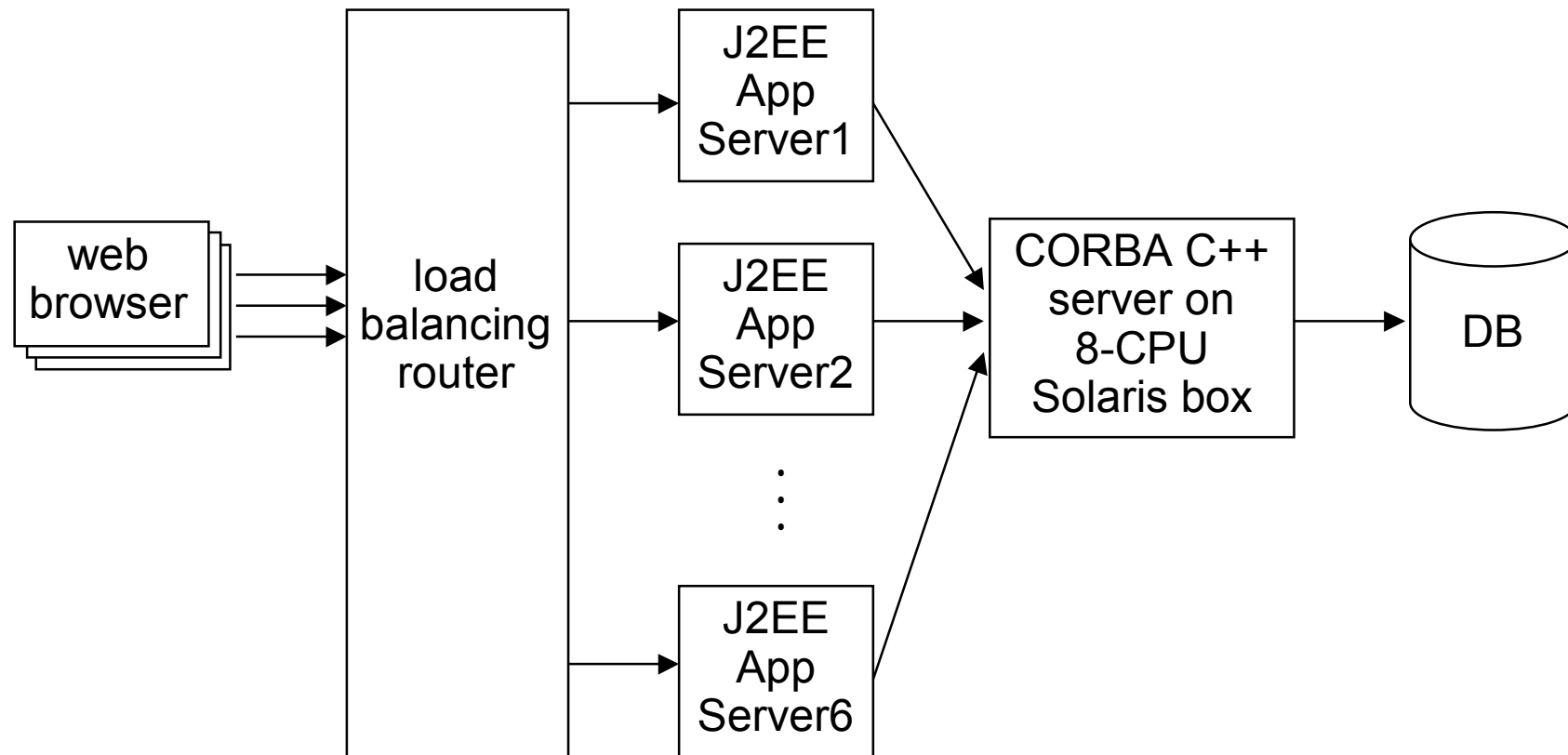
THE TRAINING COURSE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE TRAINING COURSE OR THE USE OR OTHER DEALINGS IN THE TRAINING COURSE.

Purpose of this presentation

- Some issues in multi-threading are counter-intuitive
- Ignorance of these issues can result in poor performance
 - Performance can actually get *worse* when you add more CPUs
- This presentation explains the counter-intuitive issues

1. A case study

Architectural diagram



Architectural notes

- The customer felt J2EE was slower than CORBA/C++
- So, the architecture had:
 - Multiple J2EE App Servers acting as clients to...
 - Just one CORBA/C++ server that ran on an 8-CPU Solaris box
- The customer assumed the CORBA/C++ server “should be able to cope with the load”

Strange problems were observed

- Throughput of the CORBA server *decreased* as the number of CPUs increased
 - It ran fastest on 1 CPU
 - It ran slower but “fast enough” with moderate load on 4 CPUs (development machines)
 - It ran very slowly on 8 CPUs (production machine)
- The CORBA server ran faster if a thread pool limit was imposed
- Under a high load in production:
 - Most requests were processed in < 0.3 second
 - But some took up to a minute to be processed
 - A few took up to 30 minutes to be processed
- This is *not* what you hope to see

2. Analysis of the problems

What went wrong?

- Investigation showed that scalability problems were caused by a combination of:
 - Cache consistency in multi-CPU machines
 - Unfair mutex wakeup semantics
- These issues are discussed in the following slides
- Another issue contributed (slightly) to scalability problems:
 - Bottlenecks in application code
 - A discussion of this is outside the scope of this presentation

Cache consistency

- RAM access is much slower than speed of CPU
 - Solution: high-speed cache memory sits between CPU and RAM
- Cache memory works great:
 - In a single-CPU machine
 - In a multi-CPU machine if the threads of a process are “bound” to a CPU
- Cache memory can backfire if the threads in a program are spread over all the CPUs:
 - Each CPU has a separate cache
 - Cache consistency protocol require cache flushes to RAM (cache consistency protocol is driven by calls to `lock()` and `unlock()`)

Cache consistency (cont')

- Overhead of cache consistency protocols worsens as:
 - Overhead of a cache synchronization increases (this increases as the number of CPUs increase)
 - Frequency of cache synchronization increases (this increases with the rate of mutex `lock()` and `unlock()` calls)
- Lessons:
 - Increasing number of CPUs can *decrease* performance of a server
 - Work around this by:
 - Having multiple server processes instead of just one
 - Binding each process to a CPU (avoids need for cache synchronization)
 - Try to minimize need for mutex `lock()` and `unlock()` in application
 - Note: `malloc()/free()`, and `new/delete` use a mutex

Unfair mutex wakeup semantics

- A mutex does *not* guarantee First In First Out (FIFO) wakeup semantics
 - To do so would prevent two important optimizations (discussed on the following slides)
- Instead, a mutex provides:
 - Unfair wakeup semantics
 - Can cause *temporary* starvation of a thread
 - But guarantees to avoid *infinite* starvation
 - High speed `lock()` and `unlock()`

Unfair mutex wakeup semantics (cont')

- Why does a mutex *not* provide fair wakeup semantics?
- Because most of the time, speed matter more than fairness
 - When FIFO wakeup semantics are required, developers can write a `FIFOMutex` class and take a performance hit

Mutex optimization 1

■ Pseudo-code:

```
void lock()
{
    if (rand() % 100) < 98) {
        add thread to head of list; // LIFO wakeup
    } else {
        add thread to tail of list; // FIFO wakeup
    }
}
```

■ Notes:

- Last In First Out (LIFO) wakeup increases likelihood of cache hits for the woken-up thread (avoids expense of cache misses)
- Occasionally putting a thread at the tail of the queue prevents *infinite* starvation

Mutex optimization 2

- Assume several threads concurrently execute the following code:

```
for (i = 0; i < 1000; i++) {  
    lock(a_mutex);  
    process(data[i]);  
    unlock(a_mutex);  
}
```

- A thread context switch is (relatively) expensive
 - Context switching on *every* `unlock()` would add a lot of overhead
- Solution (this is an unfair optimization):
 - Defer context switches until the end of the current thread's time slice
 - Current thread can *repeatedly* `lock()` and `unlock()` mutex in a single time slice

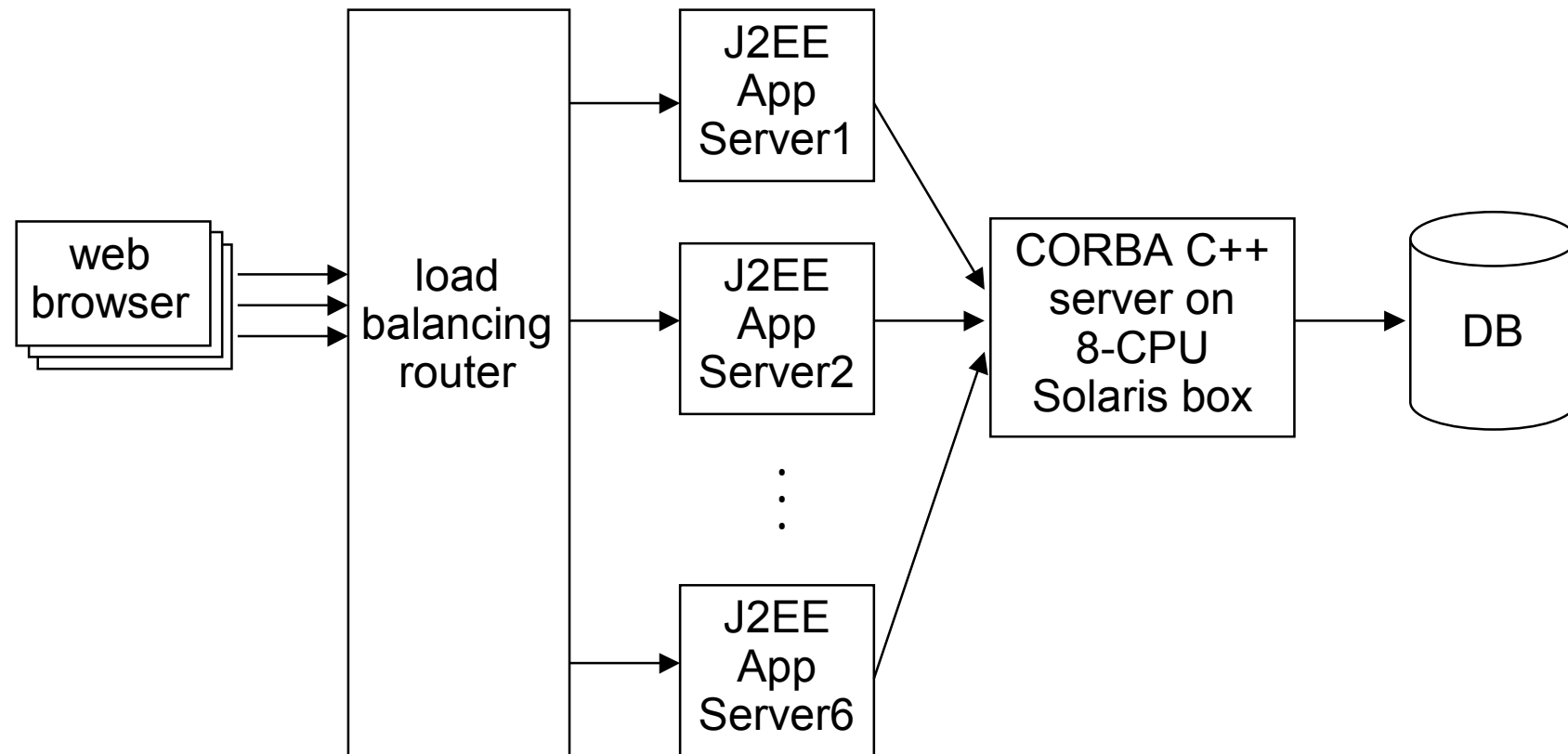
3. Improving Throughput

Improving throughput

- 20X increase in throughput was obtained by combination of:
 - Limiting size of the CORBA server's thread pool
 - This Decreased the maximum length of the mutex wakeup queue
 - Which decreased the maximum wakeup time
 - Using several server processes (each with a small thread pool) rather than one server process (with a very large thread pool)
 - Binding each server process to one CPU
 - This avoided the overhead of cache consistency
 - Binding was achieved with the `pbind` command on Solaris
 - Windows has an equivalent of process binding:
 - Use the `SetProcessAffinityMask()` system call
 - Or, in Task Manager, right click on a process and choose the menu option
(this menu option is visible only if you have a multi-CPU machine)

4. Finishing up

Recap: architectural diagram



The case study is not an isolated incident

- The project's high-level architecture is quite common:
 - Multi-threaded clients communicate with a multi-threaded server
 - Server process is not “bound” to a single CPU
 - Server's thread pool size is unlimited
(this is the default case in many middleware products)
- Likely that *many* projects have similar scalability problems:
 - But the system load is not high enough (yet) to trigger problems
- Problems are *not* specific to CORBA
 - They are independent of your choice of middleware technology
- Multi-core CPUs are becoming more common
 - So, expect to see these scalability issues occurring more frequently

Summary: important things to remember

- Recognize danger signs:
 - Performance drops as number of CPUs increases
 - Wide variation in response times with a high number of threads
- Good advice for multi-threaded servers:
 - Put a limit on the size of a server's thread pool
 - Have several server processes with a small number of threads instead of one process with many threads
 - Bind each a server process to a CPU
- Acknowledgements:
 - Ciaran McHale's employer, IONA Technologies (www.iona.com) generously gave permission for this presentation to be released under the stated open-source license.