SYSTOR 2010

# Scalability Limitations when Running a Java Web Server on a Chip Multiprocessor

Takeshi Ogasawara

IBM Research – Tokyo

# Summary of Talk

- We identified a performance scalability problem for a Java-based Web server in a real chip multiprocessor (CMP) machine.
  - Long-lived objects triggered long garbage collections (GCs)
  - Long-lived objects is tightly linked with Web client connections
  - Pause time of frequent long GC degrades the qualify of service (QoS) and performance scalability on many threads.

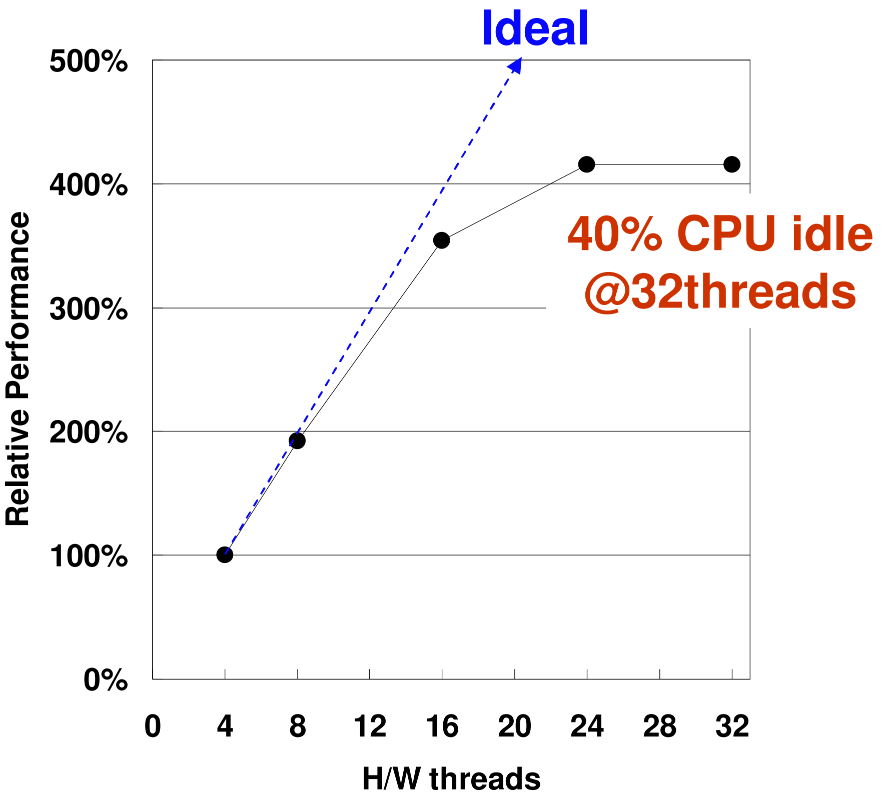
- We evaluated object pooling to address this problem.
  - Implemented object pools with thread-affinity-based selection
    - Thread local or global
  - Recycling these long-lived objects improved performance scalability by 48% at 32 hardware threads

# Background

- The number of hardware threads on a chip multiprocessor (CMP) is increasing in modern processors.
- It is critical for a Web server to take advantage of the numerous hardware threads to handle the increasing demands for Web services from large numbers of simultaneous clients.
- The performance of a Web server can scale well as the number of hardware threads increases.

# Performance Scalability Problem of a Web Server in a CMP

- The throughput scaled poorly as the number of hardware threads was increased in a CMP.
  - Threads are not blocked by resource contention.
- We believe that the increased number of hardware threads caused a change in the behavior of the Web server software.



# QoS Failure Limited the Performance Scalability

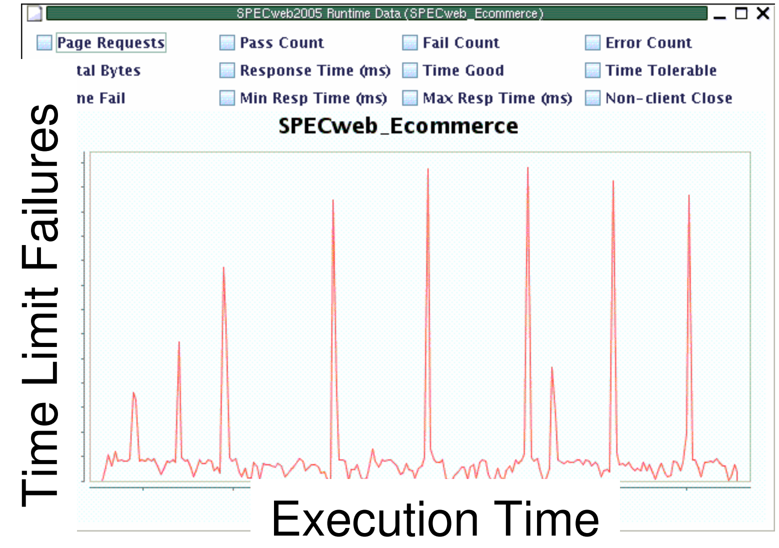
- What is happening when additional threads do not improve the throughput?

  → *QoS failure* – The frequencies of the responses that could not meet the time criteria exceeded the limits.

  - For a good server, most of the requests from Web clients should be responded within a given time limit.
  - For example, the QoS criteria specify that 95% of the total requests must be responded within 3 seconds.

# Source of QoS Failures

- The number of responses that did not meet the time limit periodically increased.
- These spikes can be associated with the spikes in the GC pause time.
  - Long GC pauses are a source of QoS failures.
- Next question – Why did such long GCs happen more often with additional hardware threads?



# Source of Long GCs

- Long GC pauses were caused by *Full GCs*.
  - Full GC is one of two GC types (minor and full) in generational GC.
  - Full GC happens when there is no free space for *long-lived objects*.

- To identify what objects are long-lived, we profiled the lifetimes and classes of objects.

→ Objects linked to the connections from clients were long-lived.

- To achieve better scalability, we should reduce the frequency of Full GCs by reducing the number of allocations of long-lived objects.

# Object Pooling

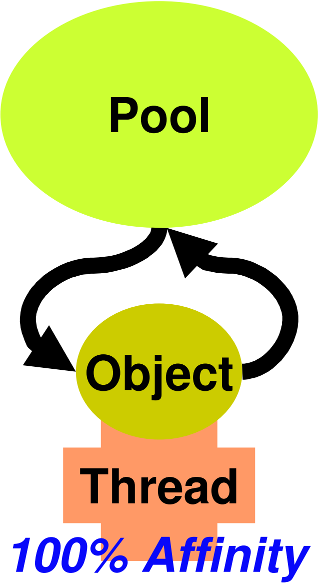
- Conventional technology
- Not used for usual objects in modern JVMs
  - Used in older JVMs to avoid slow allocations
  - Can be used for recycling the OS resources (e.g., threads, DB connections, etc.)

- We used object pooling to reduce the number of long-lived objects.

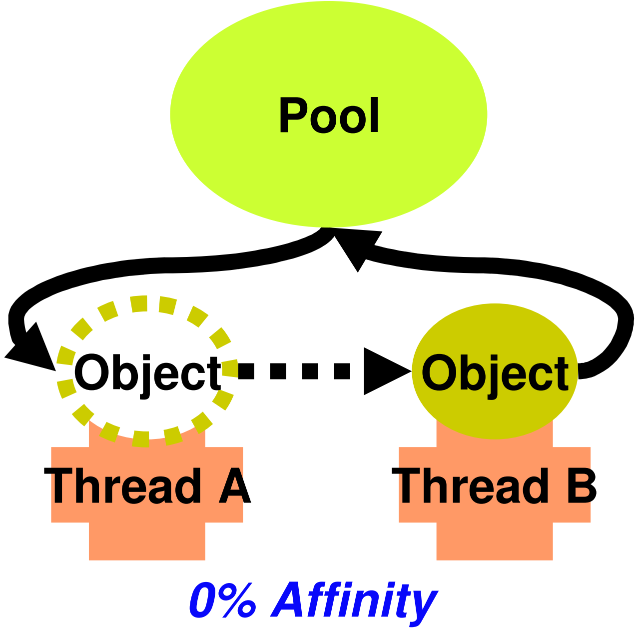
# Steps of Object Pools

1. Profile the lifetimes of objects
   - Collect object allocations with their call stacks and their garbage-collection
2. Find the objects that live long enough to be moved to the old space
   - We assume that objects surviving many minor GCs are long-lived.
3. Create a object pool for each class of the objects
   - Thread-local pool or global pool
4. Replace the code of ‘`new`’ with ‘`getFromPool()`’
5. Insert ‘`returnToPool()`’ when the objects are no longer used
   - Done by hand

# Thread Affinity of Pool Objects

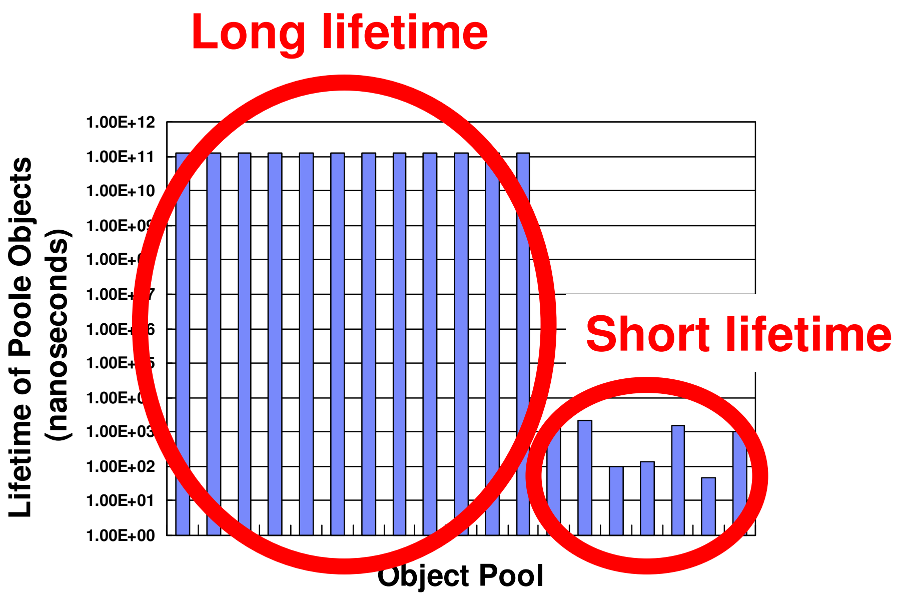


- *Thread affinity of a pool object* – how often the same thread obtains and returns the pool objects
- Thread affinity is important for good performance and low memory footprint.
  - For objects with high thread affinity, *thread-local pools* can avoid the cost of thread synchronization.
  - For objects with low thread affinity, *global pools* can avoid imbalance in resource allocation among pools.



# Association between Object Lifetime and Thread Affinity

- Lifetime groups
  - Long group
    - Avg. – 126 seconds
    - Linked to the connection times of the Web users
  - Short group
    - Avg. – < 1 usec
- Association between the lifetime groups and the thread affinity
  - Long lifetime → <2% affinity
  - Short lifetime → 100% affinity

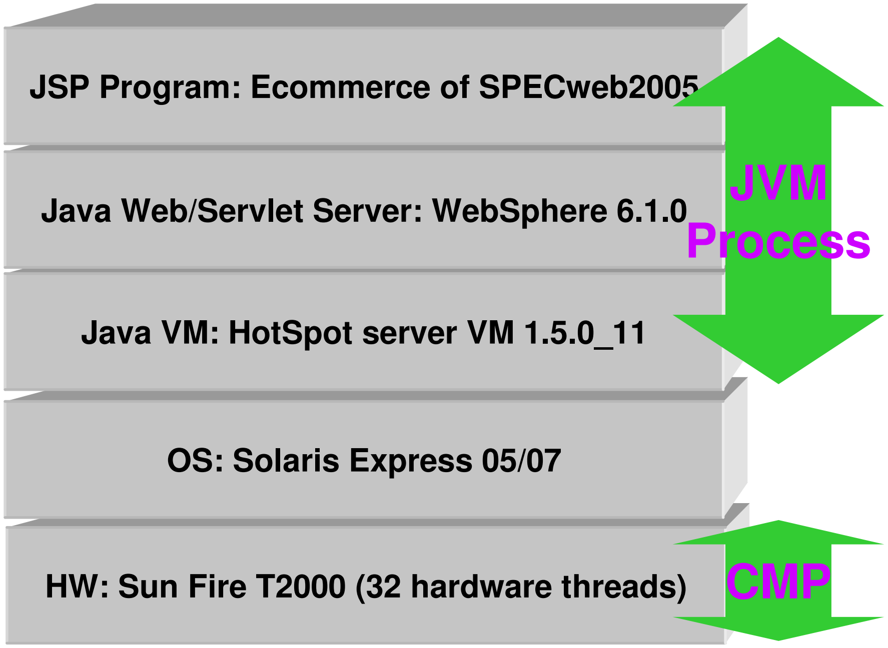


Long lifetime

Short lifetime

# Experimental Environment

- A Java-based Web server running on a CMP machine
  - A single JVM process executes most of the S/W stack.
  - A CMP machine provides 32 hardware threads.


JSP Program: Ecommerce of SPECweb2005

Java Web/Servlet Server: WebSphere 6.1.0

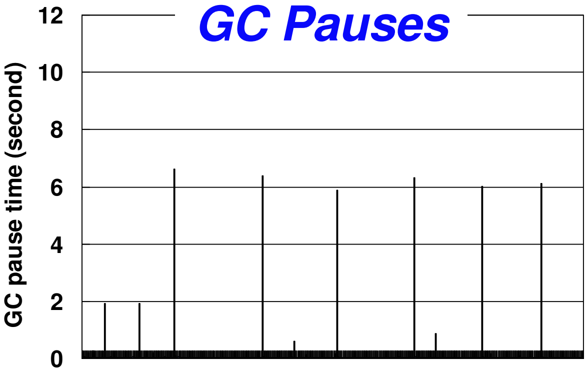
Java VM: HotSpot server VM 1.5.0_11

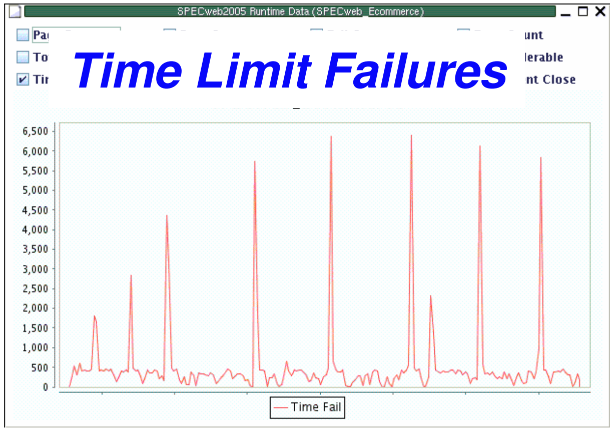
OS: Solaris Express 05/07

HW: Sun Fire T2000 (32 hardware threads)

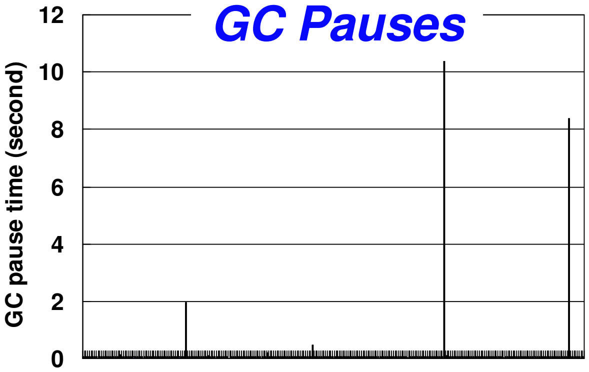
*Stack of H/W and S/W*

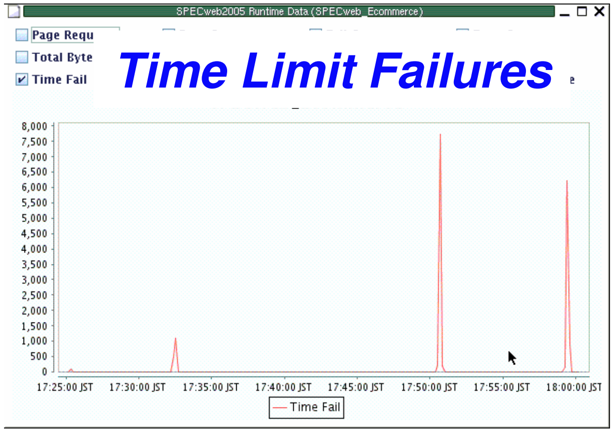
# Reduced GC Pauses and QoS Failures


GC Pauses

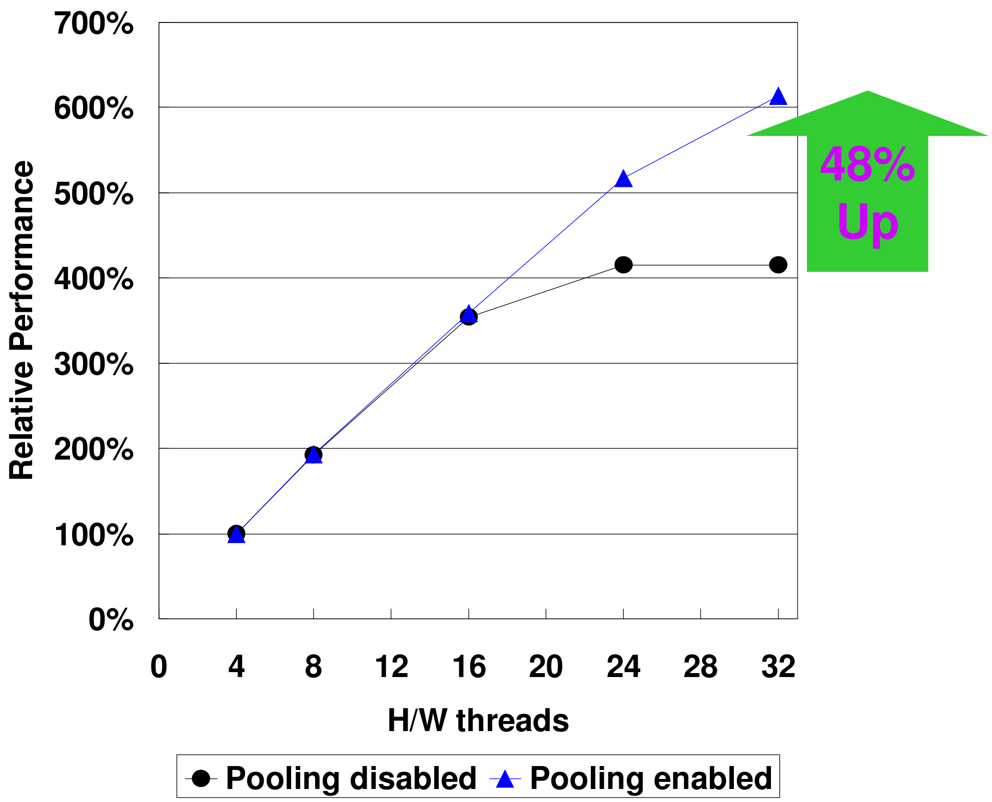

Time Limit Failures

**Object Pooling**


GC Pauses


Time Limit Failures

# Improved Performance Scalability



# Conclusions

- We analyzed a scalability problem for a Java-based Web server in a real CMP machine.
  - Long-lived objects triggered long GCs that degrade the QoS.
  - The clients' activities are tightly linked with the lifetimes of such objects.
- We evaluated object pooling to address this problem.
  - Object pools with thread-affinity-based selection
  - Recycling these long-lived objects improved the scalability by 48%

# Backup

# Source of Long GCs

- Long GC pauses were caused by *Full GCs*.
  - Full GC is one of two GC types (minor and full) in generational GC.
  - Full GC happens when there is no free space for *long-lived objects*.

- To identify what objects are long-lived, we profiled the lifetimes and classes of objects.

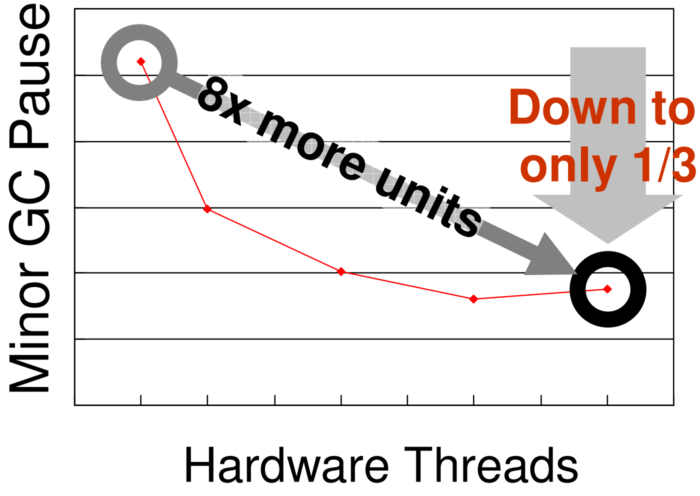
→ Objects that are linked to the connections from clients were long-lived.

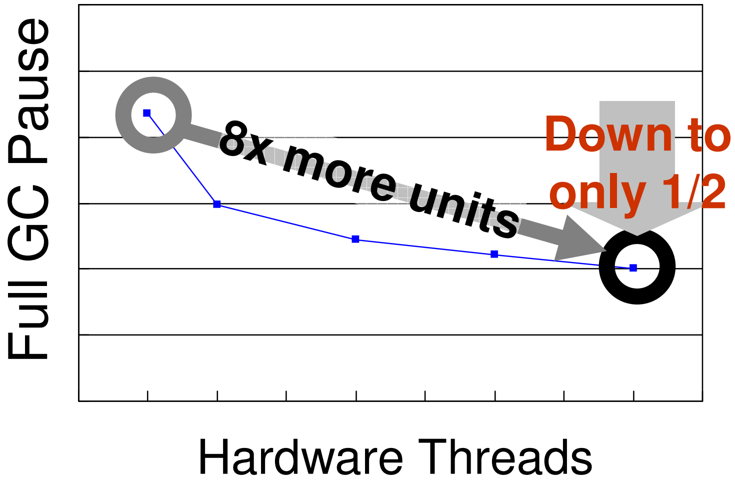
- These objects will be observed in any server because they are independent of the internal design of a server.

- To achieve better scalability, we should reduce the frequency of Full GCs by reducing the number of allocations of long-lived objects.

# Another Reason of Reducing the GC Count – GC Scales Poorly in a CMP

- We have more live objects that GC scans & copies with more exec units in a CMP.
- However, the scalability is limited because GC is memory-bound work.



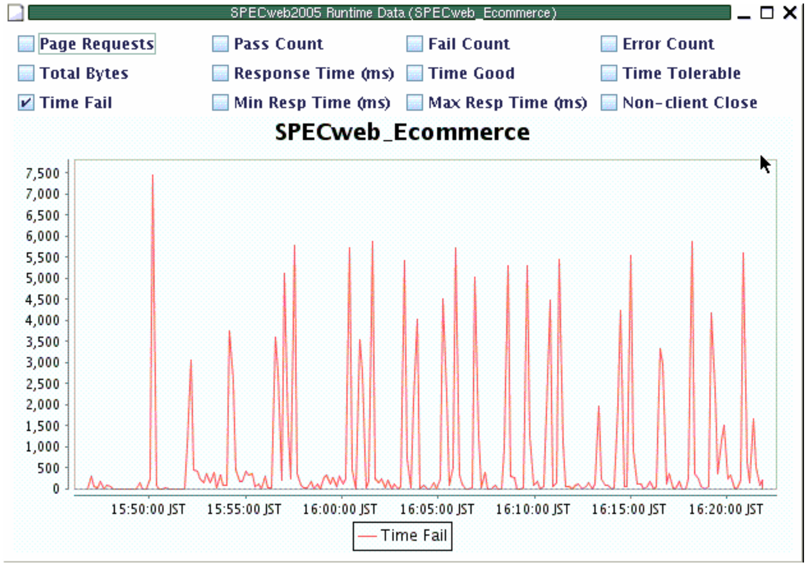


# Reuse Ratio of Pool Objects

- Very high
    - >94% on average

# Other Approach – Mostly-Concurrent Mark-and-Sweep (CMS) Collector

- The CMS collector intends to reduce the GC pause time for Full GCs by running a collector thread concurrently.
- The QoS and the throughput were degraded.
  - The pause time for Full GCs were reduced.
  - But another pause (initial mark pause) was added.



SPECweb2005 Runtime Data (SPECweb_Ecommerce)

☐ Page Requests
☐ Total Bytes
☑ Time Fail
☐ Pass Count
☐ Response Time (ms)
☐ Min Resp Time (ms)
☐ Fail Count
☐ Time Good
☐ Max Resp Time (ms)
☐ Error Count
☐ Time Tolerable
☐ Non-client Close

*More QoS failures ☹*