Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems

Kun Suo^, Jia Rao^, Hong Jiang^ and Witawas Srisa-an*

^ The University of Texas at Arlington

* The University of Nebraska at Lincoln
Exploiting Parallelism

• The rise of multicore architectures and other forms of hardware parallelism
  o Multi-core processors, accelerators, multi-queue devices, co-processors, etc.

• Exploiting parallelism in multicore systems
  o Application runtimes → meet app-specific needs
  o Operating systems → balance load, preserve locality, save energy
  o Synchronization primitives → minimize overhead

• Interplays among runtime, OS, and synchronization not well understood
  o Runtime assumes guaranteed thread-level parallelism
  o OS schedules threads based on their CPU demands

Semantic gap
Intricate Program-OS Interaction

- **Hardware & OS**: 4-socket 12-core Intel Xeon E5-4640, 512GB memory, Linux 4.14
- **App**: Linux perf benchmark (configured with 48 threads)

**Futex benchmarks**

- **Ineffective Linux load balancing** incurs more than 80% degradation
- **Perfect Linux LB** can lead to better performance

**CPU pinning** overwrites Linux LB and guarantees 1-to-1 thread-to-core mapping

Higher is better
Reasons of Imperfect Balancing

• Only runnable/ready threads are eligible for load balancing
• OS may choose not to migrate threads

Harmful interactions between parallel programs and the OS scheduler
Parallel GC in HotSpot JVM

1. Init Phase
2. Parallel Phase
3. Final Phase

- Parallel GC (default in JDK 1.8)
- Stop-the-world (STW) pause

Application (Mutator) execution time
Assigning Tasks to GC Threads

GCTask

_GCTaskQueue

GCTaskManager

Monitor

GC task

_GC_task

_GCTask

_GCTaskQueue

GC thread

_GenericTaskQueue

GCTaskThread

Core
Native `mutex` Lock in HotSpot

1. Mutex is uncontended, fast path
2. Mutex is contended, slow path
3. Competitive handoff

- Mutex_lock
- GC thread
- Mutex_unlock: release lock byte, assign heir, promote acquirers, wake heir
- GCTaskQueue is empty
- GC starts
- Transfer waiters from WaitSet to cxq
- Mutex_unlock: release lock byte, assign heir, promote acquirers, wake heir
CPU Stacking and Unfair Locking

GC threads sleeping on the condition variable

If both OnDeck and previous owner reside on the same CPU, OnDeck (almost) never wins

Other waiters cannot proceed until OnDeck wins

Wakes up, competes for lock, sleeps if failed

Fast path

void GCTaskThread::run()
{
    for (;;) {
        GCTask* task = manager()->get_task();
        task->do_it();
    }
}
Loss of Concurrency

GC task distribution among GC threads

- OldToYoungRootsTask
- ScavengeRootsTask
- ThreadRootsTask
- StealTask
Loss of Concurrency

GC thread distribution

Core ID

GC thread ID

Number of get_task() called
Inefficient Work Stealing

The breakdown of GC time

- Final Synchronisation
- All Other Tasks
- Steal Task(termination)
- Steal Task(steal)
- Initialisation

Bar chart showing the percentage breakdown of GC time for various applications and tasks.
Why Work Stealing fails to Address the Imbalance?

• HotSpot work stealing
  o **Randomly** pick up two GC threads and steal from the one with a longer queue
  o A GC thread enters a distributed termination protocol after $2^N$ failed steal attempts

Two random choices stealing not effective if there is significant task imbalance among GC threads
Our Approaches

• GC thread affinity
  - Dynamically bind GC threads to separate cores, considering load

• Optimized work stealing
  - Semi-random stealing
  - Only steal from live threads, $2 \times N_{\text{live}}$ attempts

$2 \times N_{\text{attempts}}$
Mitigating the GC Imbalance
Improvement on Overall Performance

DaCapo execution time

<table>
<thead>
<tr>
<th></th>
<th>Vanilla-JVM</th>
<th>w/ GC-affinity</th>
<th>w/ steal</th>
<th>Together</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
</tr>
<tr>
<td>jython</td>
<td>1.1</td>
<td>0.9</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td>luserach</td>
<td>1.2</td>
<td>1.0</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>sunflow</td>
<td>1.3</td>
<td>1.1</td>
<td>1.0</td>
<td>0.8</td>
</tr>
<tr>
<td>xalan</td>
<td>1.4</td>
<td>1.2</td>
<td>1.1</td>
<td>1.0</td>
</tr>
</tbody>
</table>

SPECjvm2008 throughput

<table>
<thead>
<tr>
<th></th>
<th>Vanilla-JVM</th>
<th>w/ GC-affinity</th>
<th>w/ steal</th>
<th>Together</th>
</tr>
</thead>
<tbody>
<tr>
<td>compiler.compiler</td>
<td>1.0</td>
<td>1.2</td>
<td>1.0</td>
<td>1.4</td>
</tr>
<tr>
<td>compress</td>
<td>1.1</td>
<td>1.3</td>
<td>1.2</td>
<td>1.4</td>
</tr>
<tr>
<td>crypto.signverify</td>
<td>1.2</td>
<td>1.4</td>
<td>1.3</td>
<td>1.5</td>
</tr>
<tr>
<td>xml.transform</td>
<td>1.3</td>
<td>1.5</td>
<td>1.4</td>
<td>1.6</td>
</tr>
<tr>
<td>xml.validation</td>
<td>1.4</td>
<td>1.6</td>
<td>1.5</td>
<td>1.7</td>
</tr>
</tbody>
</table>

thread affinity: 30.4%
optimized stealing: 17.5%
combined: 42.5%

Lower is better
Higher is better
Improvement on GC time

Lower is better

More than 80%

Normalized GC time

Vanilla-JVM  Optimized-JVM

h2  jython  luserach  sunflow  xalan  compiler.compiler  compress  crypto.signverify  xml.transform  xml.validation
Application Results

Spark execution time

Lower is better

Up to 13%

Vanilla-JVM  Optimized-JVM

wordcount(S)  wordcount(L)  wordcount(H)  kmeans(S)  kmeans(L)  kmeans(H)  pagerank(S)  pagerank(L)
Application Results

Cassandra read latency

Lower is better

Latency (ms)

Median
Mean
95%
99%

Vanilla-JVM
Optimized-JVM

Improved tail latency
More Results in the Paper

• Scalability

• Different heap sizes

• Multiple Java programs

• Comparison with NUMA-aware GC thread placement and work stealing [Gidra-ASPLOS’13]
Insights & Takeaways

• Thread stacking can be mitigated through more frequent OS load balancing, but not eliminated
  o Enable SMT, disable power saving, ignore NUMA

• Possibly a bigger problem than inefficient GC
  o Inherent tradeoff between sync and OS scheduling
    • Sync -- limit concurrent lock contenders
    • OS -- most effective if all threads are active
  o Up to 68% perf. difference in PARSEC benchmarks

• More general solution in OS scheduling
• Rethinking sync optimization: OS friendly vs. unfriendly
Thank you!

Questions?