Improving the Design and the Performance of Managed Runtime Environments

Gaël Thomas

INRIA/LIP6
Exploiting Multicores in Managed Runtime Environments

Lokesh Gidra, Gaël Thomas, Julien Sopena Marc Shapiro

INRIA/LIP6
Managed Runtime Environments (MREs)

MRE: simulates an abstract hardware/OS

- **Safety**: isolate code from the rest of the system
- **Portability**: write once, run anywhere
MREs are efficient

Efficient Garbage Collectors
(generational, 1984)

Efficient JIT Compilers
(Self, 1987-1989)

Powerful processors
(more than 100MHz, 1992)

Safety/portability requirements
(HTML, 1993)

Inefficient ~ 1990 Efficient
MREs are everywhere

- Smartphones
- Web browsers
- Desktop
- Web servers
But they were not prepared to multicore

Most MREs were designed for a monocore architecture

⇒ Necessary to study their bottlenecks on a multicore architecture
Pause time of the GC increases with GC threads

⇒ Negative scalability!

HotSpot JVM’s Garbage Collectors

Pause Time
Application Time

Time in Milliseconds (Lower is better)

Number of GC Threads

1 6 12 24 36 48
ParScavenge
1 6 12 24 36 48
ConMS
1 6 12 24 36 48
G1

GC Scalability (Lusearch) [PLOS’11]

Gaël Thomas
A Study of the Scalability of Garbage Collectors on Multicores
Why suspending the application

The concurrency issue

Pending queue
Why suspending the application

The concurrency issue

Pending queue
Why suspending the application

The concurrency issue

Pending queue
Why suspending the application

The concurrency issue

Pending queue
Why suspending the application

The concurrency issue

Application executes $B \rightarrow f = G$

Pending queue
Why suspending the application

The concurrency issue $\Rightarrow$ G freed while still used
Common believe

1. Stop-the-world:
   - Suspend the application
   + Simple to implement

2. Concurrent:
   fine-grain locking, code instrumentation
   - Hard to implement
   - Degrades application performance
   + Do not pause the application

Current believe:

STW are unacceptable for server apps [Iyengar, ISMM 2012]
Long pauses due to larger heaps
Our hypothesis

Increase in transistor count is for both memory and CPU
✓ Large heaps come with large core count
✓ STW GC should be still useful, provided they scale

Can we make a GC scales with the number of cores to avoid the price of concurrent collectors?
Contribution

Identify the bottlenecks of Parallel Scavenge
  (the most scalable GC of OpenJDK – used by default)
  ✓ Heavy contended locks
  ✓ Lack of NUMA-awareness

Solve the bottlenecks
  ✓ Remove all the locks during the collection
  ✓ Propose 3 NUMA-aware heap layouts
    ✷ Interleaved: balance memory accesses across the nodes
    ✷ Fragmented: balance + increase memory locality
    ✷ Segregated: balance + perfect memory locality
1. Background

2. The lock bottleneck

3. The NUMA bottleneck

4. Evaluation

5. Conclusion
1. Background

2. The lock bottleneck

3. The NUMA bottleneck

4. Evaluation

5. Conclusion
Background: the copying collector

From Space

C → D → F
B → E → G

To Space

Pending queue
Background: the copying collector

Step 1: identify the root objects (globals, stack)
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue

From Space

To Space

Pending queue
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue

From Space

To Space

Pending queue
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue

Gaël Thomas
A Study of the Scalability of Garbage Collectors on Multicores
Background: the copying collector

Step 2: copy an object from the pending queue + update pending queue
Background: the copying collector

Step 3: invert the spaces + consider to space empty

Advantage: spaces are never fragmented
1. Background

2. The lock bottleneck

3. The NUMA bottleneck

4. Evaluation

5. Conclusion
Poor Synchronization in Parallel Scavenge

![Diagram showing coarsely grained synchronization and use of monitors]

Coarse grained synchronization + use of monitors
Simplify synchronizations

Stop-the-World pause

Termination protocol

Mutators

VM Thread

GC Threads
1. Background

2. The lock bottleneck

3. The NUMA bottleneck

4. Evaluation

5. Conclusion
Impact of a NUMA architecture

Problem 1: unbalanced memory accesses

Interconnect or memory controllers saturate
Impact of a NUMA architecture

Problem 2: remote memory accesses

Interconnect saturates
Impact of a NUMA architecture

![Graph showing the impact of NUMA architecture on speedup with varying number of threads. The graph compares Unbalanced-Remote, Balanced-Remote, Balanced-Local, Unbalanced-Local, and Linear Speedup. The x-axis represents the number of threads, and the y-axis represents speedup.]
Inefficient Memory Layout in ParallelScavenge (PS)

The initial thread fixes the mapping of physical pages

SPECjbb2005 allocates ~95% of memory from a single node
Solution 1: Interleaved Space

Map the pages on the node in round-robin

⇒ perfect memory balance
Solution 1: Interleaved Space

Idea: map the pages on the node in round-robin

⇒ perfect memory balance but bad memory locality
Solution 2: Fragmented Space

Associate fragments to memory nodes
⇒ node-local allocation

![Diagram showing physical memory, heap, and node allocation]
**Solution 2: Fragmented Space**

**Naturally balance the load**
- ✓ GC threads uniformly distributed on the nodes
- ✓ Efficient work stealing between GC threads
  ⇒ balance the copies on all the nodes

![Diagram showing balance in space distribution between nodes](image)
Solution 2: Fragmented Space

Increase locality for the application

- Mutator mostly accesses recently allocated objects
- Recently allocated object is on the mutator’s node
Solution 2: Fragmented Space

Increase memory locality during copy

From Space

Node 0

Node 1

To Space

Thread on node 1

Node 0

Node 1

Pending queue
Solution 2: Fragmented Space

Increase memory locality during copy

From Space

Node 0

C
D
B

To Space

Node 0

A

Node 1

F
G
E

Pending queue

E
B

Thread on node 1

Gaël Thomas
A Study of the Scalability of Garbage Collectors on Multicores
Solution 3: Segregated Space

Fragmented space + node-local scanning
(Send remote references to the owner of the object)

Perfect locality for the GC
But have to pay the price of inter-node message exchanges

Good locality for the mutators
Mutator mostly accesses recently allocated objects

Natural balance of the load if allocation rates of the mutators are similar
# Summary of the spaces

<table>
<thead>
<tr>
<th></th>
<th>Interleaved</th>
<th>Fragmented</th>
<th>Segregated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Balance</td>
<td>Perfect</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Mutator Locality</td>
<td>Bad</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>GC Locality</td>
<td>Bad</td>
<td>Good</td>
<td>Perfect</td>
</tr>
<tr>
<td>Comment</td>
<td></td>
<td></td>
<td>Inter-node messages</td>
</tr>
</tbody>
</table>
1. Background

2. The lock bottleneck

3. The NUMA bottleneck

4. Evaluation

5. Conclusion
Evaluation

Hardware:

- AMD Opteron 6172 sockets
- 8 nodes and 48 cores

25 evaluated applications:

- SPECjbb2005: ~ 3.5 GB
- SPECjvm2008: ~ 1 to 2 GB
- DaCapo 9.12: ~ 500

Focus on SPECjbb2005 in the presentation
Effect of optimizations on the GC

(a) SPECjbb

Number of GC Threads
Effect of optimizations on the GC

- ParallelScavenge
- Interleaved Space
- Fragmented Space
- Segregated Space
- Fragmented Space + Lock-free

(a) SPECjbb

GC Throughput (GB/Sec)

Number of GC Threads

High memory imbalance (95% on node 0)
Effect of optimizations on the GC

- ParallelScavenge
- Interleaved Space
- Segregated Space
- Fragmented Space
- Fragmented Space + Lock-free

(a) SPECjbb

GC Throughput (GB/Sec)

Number of GC Threads

Latency effect due to locality marginal

Bad locality Hampers scalability
Effect of optimizations on the GC

(a) SPECjbb

<table>
<thead>
<tr>
<th>Segregated Space</th>
<th>Parallel Scavenge</th>
<th>Interleaved Space</th>
<th>Fragmented Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Locality required for scalability
Effect of optimizations on the GC

- ParallelScavenge
- Interleaved Space
- Fragmented Space
- Segregated Space
- Fragmented Space + Lock-free

(a) SPECjbb

Effect of lock
Negligible with few cores

Effect of lock becomes important with many cores

Gaël Thomas
A Study of the Scalability of Garbage Collectors on Multicores
Effect of optimizations on the GC

(a) SPECjbb

Too many messages between the nodes (12% of objects are still remote)
Effect of optimizations on the Application

![Graph showing performance improvements with different optimizations](image)

- **Parallel Scavenge**
- **Interleaved Space**
- **Fragmented Space**
- **Fragmented Space + Lock-free**

**XML Transform**

- **Balance**: Increase performance of app.
- **Locality**: Marginal effect

**GC time excluded**

Gaël Thomas
*A Study of the Scalability of Garbage Collectors on Multicores*
Overall effect

---

**ParallelScavenge**

**Interleaved Space**

**Fragmented Space**

**Segregated Space**

**Fragmented Space + Lock-free**

---

**Optimizations translates into a x2 on throughput of SPECjbb**

**Pause time of 1 collection:**

from 105ms to 49ms

---

(a) SPECjbb

**Appication Throughput in Kops/sec**

(Higher is Better)

---

**Number of GC Threads**

---

Gaël Thomas

A Study of the Scalability of Garbage Collectors on Multicores
Scalability with the cores

(a) SPECjbb

(b) XML Transform

(c) Compiler.Sunflow

(d) XML Validation

(e) Crypto AES

(f) Eclipse

ParallelScavenger  
Fragmented Space + Lock-free

Gaël Thomas  
A Study of the Scalability of Garbage Collectors on Multicores
Scalability with the cores

ParallelScavenge

Fragmented Space + Lock-free

Stop-the-world scales with memory intensive applications

Gaël Thomas
A Study of the Scalability of Garbage Collectors on Multicores
Scalability with the cores

Adding GC threads is useless for non memory-intensive applications
To take away

STW GCs assumed to be inherently non-scalable is probably a mistake
⇒ Stop-the-world GC still well suited for contemporary H/W

Most important NUMA effects

✓ Balancing memory accesses has the most important impact
✓ Increasing memory locality is required to scale
✓ Latency improvement due to locality negligible

Next step

✓ Avoiding most of the messages between the nodes

[A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores, ASPLOS 2013]