Dynamic Selection of Application-Specific Garbage Collectors

Sunil V. Soman
Chandra Krintz
University of California, Santa Barbara
David F. Bacon
IBM T.J. Watson Research Center
Background

- VMs/managed runtimes
  - Next generation internet computing
  - Automatic memory management for memory safety
  - High performance, multi-application servers
- GC performance impacts overall performance
  - So GC must perform well
GC Research

- Focus on general-purpose GC
  - One GC for all apps
- No **single** GC provides best performance in all cases
  - **Across** applications
  - Even for the **same** application given different memory constraints
Motivation

![Graphs showing the relationship between heap size relative to minimum and throughput or execution time for SPECjbb2000 and _209_db benchmarks.](image)

- **Throughput** measured in $10^6$/Throughput.
- **Execution Time** measured in seconds.
- **Switch point** highlighted on the graph.

**Legend:**
- SS
- MS
- GMS
- GSS
- CMS
Motivation

- Other researchers have reported similar results [Attanasio ’01, Fitzgerald ’00]
  - Spread between best & worst at least 15%
  - Generational not always better
- Employ a VM instance built with the optimal GC
- Goal of our work: **employ multiple GCs in the same system and switch between them**
Framework

- Implemented in the JikesRVM
  - Uses the Java Memory management Toolkit (JMTk)
    - Extended to enable multiple GC support
- Can switch between diverse collectors
  - Easily extensible

- GC System = Allocator + Collector
Framework Implementation

- Most code is shared across GCs
  - 44MB vs 42MB (average across GCs)
Framework Implementation

- Example
  - MS -> GSS
  - MS -> GMS

- Immortal
- GC Data Structures
- Large Object Space
- Low Semispace
- High Semispace
- Mark-Sweep
- Nursery
- Alloc
- Unmap
Framework Implementation

- Example
  - MS -> GSS
  - MS -> GMS

- Alloc
  - No GC
  - Do not unmap

- Nursery
  - Mark-Sweep
    - High Semispace
    - Low Semispace
    - Large Object Space
    - GC Data Structures
      - Immortal
      - No GC
      - Do not unmap

- Higher
  - Nursery
  - Mark-Sweep
    - High Semispace
    - Low Semispace
    - Large Object Space
    - GC Data Structures
      - Immortal
      - No GC
      - Do not unmap

- Lower

ISMM '04
Framework Implementation

- Nursery
  - Mem mapped on demand
  - Unused memory unmapped
- Mark-Sweep
  - Need MS & copying states
  - Use object header
  - Bit stealing
- High Semispace
- Low Semispace
- Large Object Space
- GC Data Structures
  - Worst-case cost = 1 GC
  - No need to perform GC in many cases

- Immortal
Making Use of GC Switching

- Dynamic switching guided by
  - Cross-input offline behavior
  - Annotation

- Methodology
  - 2.4 GHz/1G single proc x86 (hyperthreading)
  - JikesRVM v2.2.0, linux kernel 2.4.18
  - SpecJVM98, SpecJBB, Jolden, Javagrande

- Pseudo-adaptive system
Annotation-guided GC (AnnotGC)

- Annotation
  - Hints about optimization opportunities
  - Widely used to guide compiler optimization
- Different benchmarks & inputs examined offline
  - Cross-input results combined
  - Select “optimal” GC and annotate program with it
  - Best GC for a range of heap sizes
- JVM at program load time
  - Switch to annotated GC
  - Ignore annotation if GC-Switching not avail.
AnnotGC Implementation

Observations
- Only 0 or 1 switch points per benchmark
- Switch point relative to min heap does not vary much across inputs

Annotate min heap size & switch point
- 4 byte annotation in Java class file
- Encoded in a compact form
- 40 MB min assumed if not specified
AnnotGC Results

![Graphs showing performance metrics for SPECjbb2000 and _209_db with different heap sizes.](image-url)
AnnotGC Results

- Degradation over best: 4%
- Improvement over worst: 26%
- Improvement over GenMS: 7%
## AnnotGC Results

### Average Difference Between Best & Worst GC Systems

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Degradation over Best</th>
<th>Improvement over Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>6.28% (443 ms)</td>
<td>3.53% (279 ms)</td>
</tr>
<tr>
<td>jess</td>
<td>2.82% (85 ms)</td>
<td>56.17% (5767 ms)</td>
</tr>
<tr>
<td>db</td>
<td>2.88% (532 ms)</td>
<td>12.47% (3028 ms)</td>
</tr>
<tr>
<td>javac</td>
<td>5.64% (392 ms)</td>
<td>24.12% (2944 ms)</td>
</tr>
<tr>
<td>mpegaudio</td>
<td>3.54% (214 ms)</td>
<td>3.21% (209 ms)</td>
</tr>
<tr>
<td>mtrt</td>
<td>4.51% (270 ms)</td>
<td>42.29% (5170 ms)</td>
</tr>
<tr>
<td>jack</td>
<td>3.22% (147 ms)</td>
<td>32.70% (2787 ms)</td>
</tr>
<tr>
<td>JavaGrande</td>
<td>3.97% (2511 ms)</td>
<td>17.71% (15500 ms)</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>2.22% (3.17*10^6/tput)</td>
<td>27.95% (82.6*10^6/tput)</td>
</tr>
<tr>
<td>MST</td>
<td>4.07% (30 ms)</td>
<td>48.42% (1001 ms)</td>
</tr>
<tr>
<td>Voronoi</td>
<td>9.20% (144 ms)</td>
<td>31.78% (1063 ms)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>4.38%</strong></td>
<td><strong>26.22%</strong></td>
</tr>
<tr>
<td><strong>Average (without MS)</strong></td>
<td><strong>3.36%</strong></td>
<td><strong>24.13%</strong></td>
</tr>
</tbody>
</table>
Extending GC Switching

- Switch automatically
  - No offline profiling
  - Employ execution characteristics
- Performance hit: lost optimization opportunity
  - Inlining of allocation sites
  - Write barriers
- Solution: Aggressive specialization guarded by
  - Method invalidation
  - On-stack replacement
Investigating Auto Switching

- Exploit general performance characteristics
  - Best performing GCs across heap sizes & programs

- Deciding when to switch
  - Default GC: MS
  - Heap size & heap residency threshold
  - If residency > 60%,
    - switch to GSS if heap size > 90MB
    - else use GMS

- Different collectors for startup & steady-state
Preliminary Implementation

- Method invalidation - for future invocations
- On-stack replacement - for currently executing methods
  - Deferred compilation & guarded inlining [Fink’04]
  - Unconditional OsrPoints
- Our extension
  - OsrInfoPoints for state info
    - Without inserting checks in application code
AutoSwitch Results

- Improvement over worst: 17%  (Annot: 26%)
- Degradation over best: 15%   (Annot: 4%)
- Overhead
  - OSR - negligible
  - Recompilation - depends on where switch occurs

- Lost optimization opportunities
  - all variables live at every GC point
  - OsrInfoPoints are “pinned” down
  - DCE, load/store elim, code motion, reg allocation
Related Work

- Application-specific GC studies
  - Profile-direction GC selection [Fitzgerald’00]
  - Comparison of GCs [Attanasio’01]
  - Comparing gen mark-sweep & copying [Zorn’90]

- Switching/swapping
  - Coupling compaction with copying [Sansom’92]
  - Hot-swapping mark-sweep/mark-compact [Printezis’01]
  - BEA Weblogic JRockit [BEA Workshop’03]
Conclusion

- Choice of best GC is app-dependent
- Novel framework
  - Switch between diverse collectors
  - Enables annotation driven & automatic switching
- Significantly reduce impact of choosing the "wrong" collector
  - AnnotGC: 26%, degradation: 4%, GMS: 7%
  - Enabled by aggressive specialization
Future Work

- Improving AutoSwitch
  - Improved OSR
  - Cuts AutoSwitch overhead by half
    - Paper available upon request

- Better heuristics for automatic switching
  - Decide *when* to switch & *which* GC to switch to

- High-performance application-specific VMs
  - Guided by available resources
  - Application characteristics
  - Self-modifying
App-specific Garbage Collectors

Front Loaders

Rear Loaders

Recyclers

Side Loaders

All products are trademarks of Heil Environmental Industries, Ltd. 2002-2003
Extra slides
Preliminary Implementation

- Aggressive specialization
  - Inline allocation sites
  - Insert write barriers only when necessary

- Method Invalidation
  - Invalidated method replaced by stub

- On-stack Replacement (OSR)
  - Potentially at every GC point
On-stack replacement

- Insert **OsrInfoPoints** at each GC point
  - Keep track of state
  - Call sites, allocation sites, prologs, backedges, explicit yieldpoints, exceptions
  - All local/stack variables kept alive

- Lazy OSR

```java
osr_helper {
  ...
}
```

b. OSR triggered lazily by external event
GC Performance Evaluation (2)

- Comparison of parallel garbage collectors [5]
  - Mark-sweep, copying, Gen MS, Gen copy, hybrid (copying + MS)

- Results
  - Hybrid has lowest heap residency
  - Gen copy handles fragmentation better
  - Minor collections for Gen copy faster
  - Mark-sweep has fewer GCs

- No one collector best across applications
AnnotGC Results

- Degradation over best: 4%
- Improvement over worst: 26%
- Improvement over GenMS: 7%
GC Performance Evaluation

- Comparing generational mark-sweep & copying [66]
- Mark-sweep
  - Slower free-list allocator
  - Requires less heap space: 20% less on average
- Copying collection
  - Fastest allocator
  - Copying overhead
  - Copy reserve space required
- Generational copying not clearly superior
Comparison of parallel garbage collectors [5]
- Mark-sweep, copying, Gen MS, Gen copy, hybrid (copying + MS)

Results
- Hybrid has lowest heap residency
- Gen copy handles fragmentation better
- Minor collections for Gen copy faster
- Mark-sweep has fewer GCs

No one collector best across applications
Fitzgerald et al [32] compared GCs across 20 benchmarks
- Null collector, non-gen copying, Gen copy with various WB implementations
- Every collector best at least once
- Spread between best & worst at least 15%
- Generational not always better
  - Non-generational may outperform by 15-20%
- Recommend “profile-directed” selection
  - The “best” GC chosen from different pre-compiled binaries based on profiles
- It is difficult or impossible to replace GCs at runtime