Syncopation:
Generational Real-time Garbage Collection in the Metronome

David F. Bacon    Perry Cheng    David Grove
IBM T.J. Watson Research Center
Martin T. Vechev
Cambridge University
The Metronome Project

- Make standard Java suitable for real-time programming

- Real-time Garbage Collection [POPL’03]
  - Solves biggest obstacle for real-time Java
  - Prototype in Jikes RVM
  - Uniprocessor algorithm
  - Worst-case pause 12ms; utilization at least 45%

- Metronome v2
  - Worst-case pause 1.5ms; utilization at least 70%
  - For periodic scheduled tasks up to 200 Hz, jitter is 4us
  - Multiprocessor algorithm
  - Implementation in IBM’s J9 virtual machine product
  - Alpha versions in use by customers and universities
IBM Real-Time Java (J9 Virtual Machine)

- **Metronome Real-time Garbage Collection**
  - Provides real-time without changing the programming model

- **Ahead-of-Time Compilation**
  - Ahead-of-time (AOT) compilation and JXE Linking
  - Removes JIT non-determinism, allows code ROMification

- **RTSJ (Real-Time Specification for Java) - standard**
  - Scheduling
  - Scopes (yuck)
Why is real-time collection different?

- Why is real-time collection different from other GCs?
  - Must break work into increments (while heap changes)
  - Must bound individual work increments
  - Must guarantee space bounds and timely termination

- Why is real-time collection different from other tasks?
  - Deadline for overall task is low-frequency
  - Allows re-distribution of work over individual increments
Redistributing Collector Work

Example Application

Allocates half as fast as the collector can collect

\[ c = -2a \]
Allocation Rate Determines Utilization

\[ a = \text{allocation rate} \]
\[ c = \text{collection rate} \]

Base Application Memory

Resulting Schedule: Utilization = 50%

Utilization = 66%
### View of Metronome Collector Schedule

<table>
<thead>
<tr>
<th>Application</th>
<th>Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocates $au \Delta t$</td>
<td>collects part of $m$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\Delta t &= u \Delta t + (1-u) \Delta t \\
&= \Delta t
\end{align*}
\]
Generational Metronome

Goal:

Increase utilization and reduce memory consumption
Nursery Reduces Allocation Rate

- High allocation rate

- Lower allocation rate - but slows main heap collection
Synchronous Nursery Collection

- Main heap collection is concurrent (in even increments)
  - Concurrent nursery collection would be extremely complex
- Small nursery can be collected in a very short time
  - Greatly simplifies design
- How much time?
  - Nursery size $N$, survival rate $\eta$, collection rate $R_N$
  - Nursery collection work is proportional to live data
  - Time to collect is $N\eta/R_N$
Quantifying the Scheduling Tradeoff

<table>
<thead>
<tr>
<th>Application</th>
<th>Collector</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocates $au\Delta t$</td>
<td>Nursery coll. $a\eta u\Delta t$</td>
</tr>
<tr>
<td></td>
<td>Heap coll. part of $m$</td>
</tr>
</tbody>
</table>

$$\frac{a\eta u\Delta t}{R_N}$$

$$u\Delta t$$

$$(1-u)\Delta t$$

$\Delta t$
When to use Generational Collection?

![Graph showing relative heap size vs utilization for different generational collection strategies.](image-url)
Handling Non-uniform Program Behavior
Can Application be Modeled So Simply?

\[ a = \text{allocation rate} \]

\[ c = \text{collection rate} \]
Maximum Mutator Allocation Rate (MMAR)
Complications due to non-uniformity

- Cost of collection is dependent on varying parameters
  - High allocation rate: frequent nursery collection
  - High survival rate: expensive individual collection
- Tension
  - Survival rate drops as nursery gets larger (more time to die)
  - But WCET of nursery collection increases
Syncopation: Move work to unstressed interval

- **Spike in allocation rate forces frequent nursery GC's**
  - Degenerates into work-based collection

- **Syncopation: pre-tenure all objects**
  - During allocation spikes, allocate directly into heap
  - No nursery collection needed

- **Problem: we don’t find out until it’s too late**
  - Once collector quantum is used, can’t GC for a mutator quantum
Scheduling: Multiple Beats per Measure

\[ \Delta t \]

\[ u = 2/3 \]

\[ u = 9/12 = 3/4 \]

12 beats/measure
3/12 time; \( u = 3/4 \)

\[ u = 8/12 = 2/3 \]

pre-tenuring

\[ u = 9/12; \text{ reserve restored} \]
Cost of Syncopation

- During syncopation, heap consumed at rate $a$, not $a\eta$
  - No benefit from generational collection during this time
  - Even if program obeys the generational hypothesis

- Space bound requires characterization of spikes
  - Which requires more detailed characterization of application
  - Tradeoff: more detail, tighter bound

- See Mann et al (next paper in session)
Increasing the Effective Nursery Size
Reducing the Survival Rate

- Multiple beats per measure means smaller nurseries
  - Which leads to higher survival rates
  - Which reduces effectiveness of generational collection
- Large objects contribute disproportionately
  - *Arraylet pre-tenuring* solves this problem
What Are Arraylets?

- Large arrays create problems for real-time collection
  - Fragment memory space
  - Can not be moved in a short, bounded time
- Solution: break large arrays into arrayoids & arraylets
  - Access via indirection; move one arraylet at a time
  - Optimize: opportunistic contiguity, strip-mining, CSE
Arraylet Pre-tenuring

- Exploit arraylet indirection
  - Arrayoid allocated in nursery
  - Arraylets allocated in heap
  - Nursery consumption:
    - Array Size / Page Size + Header Size
- On nursery collection
  - If array is live, copy
    - Arraylet pointers unchanged
  - If array is dead, reclaim arraylets
- Issues for pointer arrays
  - Arraylets could ref. nursery
  - Requires extra fixup work
  - Must factor into GC time
Reduction in Allocation Rate (32K nursery)
Conclusions

- **Syncopation allows RT scheduling of generational GC**
  - Modest over-provisioning required (extra beat)

- **Generational collection can improve performance**
  - Space and utilization

- **Arraylet pre-tenuring increases effectiveness**
  - Increases virtual nursery size; more time to die

- **Cost: more precise (complex) application characterization**
Ongoing Work

- Handlers [w/Dan Spoonhower, CMU]
  - Allow low-latency (microsecond) response for aperiodic tasks
  - Safe – checked at load time; eliminate the need for scopes
- E-code support: Logical Execution Time [w/Kirsch & Henzinger]
  - Provide deterministic timing across multiple platforms
  - Scheduling is a first-class virtual machine primitive
  - Supported in product code!
- Trace analysis and visualization [w/Matthias Hauswirth, Lugano]
- Probabilistic real-time methodology
- Verifiable concurrent collectors [w/Martin Vechev, Cambridge]
- Wait-free scalable SMP collector
- Porting to Stargate nodes (and other embedded ARM nodes)
Questions?

http://www.research.ibm.com/metronome
Syncopation: Generational Real-time Garbage Collection in the Metronome

PNG

100us
1000us
10000us

0 25 49 73 97 121 145 169 193 217 241 265 289 313 337 361 385 409 433 457 481 505 529 553 577
Allocation Stability vs. Time Scale

Peak Allocation Rate

- Time Window (microseconds)
- MB/s (peak)

<table>
<thead>
<tr>
<th>Time Window</th>
<th>MB/s (peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>250</td>
</tr>
<tr>
<td>500</td>
<td>150</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
</tr>
<tr>
<td>5000</td>
<td>50</td>
</tr>
<tr>
<td>10000</td>
<td>50</td>
</tr>
<tr>
<td>Average</td>
<td>50</td>
</tr>
</tbody>
</table>