Dynamic Prediction of Collection Yield for Managed Runtimes

Michal Wegiel
mwegiel@cs.ucsb.edu
Chandra Krintz
ckrintz@cs.ucsb.edu

Computer Science Department
University of California, Santa Barbara

May 13, 2009

Presentation by Andreas Rottmann
Department of Computer Sciences
University of Salzburg
a.rottmann@gmx.at

Outline

Outline

Outline

The Goal: Minimizing GC cost

Approaches

- Generational GC
- Employ parallelization and concurrency
- Compacting GC
- Coordinate GC & OS virtual memory managment
    - Enhance locality (minimize TLB and cache misses)
    - GC avoidance using yield prediction

Detecting ineffective collections

Does it make sense?

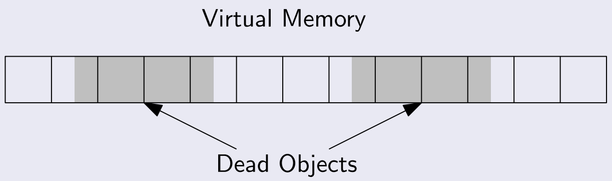
- Existing systems don't consider GC productivity
- Many GC runs are *not productive*
- Average GC yield of $< 5\%$ of heap size common in Java applications
- Avoid ineffective collections

Yield Prediction

- Need a fast and accurate estimator of GC yield
- Marking phase gives 100% accuracy, but...
- ...amounts to 50% – 90% of total GC time

The Yield Predictor

Observation: Dead objects form clusters



- The hardware already tracks usage of virtual memory pages
- Use hardware page reference counts to predict collection yield

Exploiting Virtual Memory

Background

Each virtual memory page has:

- Dirty Bit
- Recently-Referenced Bit (RR)

Managed by OS/hardware to implement swapping

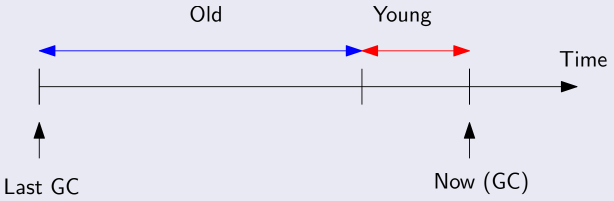
- Correlation: not-RR $\leftrightarrow$ objects on that page dead

YP Parameters

Skip Threshold

Skip collections predicted to yield $< x\%$ of heap size

Old-Young Ratio



Data Structures

Timestamp Array

- Carries (approximate) time of last access for each page
- Populated by *polling thread*, regularly querying the RR bits from OS

MRE-cleared, *OS-cleared* page bits

- Just in software
- Used to multiplex the RR hardware bit between MRE and OS

Mispredicted-Dead Array

- To account for seldom-used, permanent data structures

RR multiplexing

C_{OS}, C_{YP}: "cleared" flags

	Read	Clear
YP	$RR_{YP} = RR_{HW} \oplus C_{OS}$	$RR_{HW} \leftarrow 0, C_{YP} \leftarrow 1, C_{OS} \leftarrow 0$
OS	$RR_{OS} = RR_{HW} \oplus C_{YP}$	$RR_{HW} \leftarrow 0, C_{OS} \leftarrow 1, C_{YP} \leftarrow 0$

The Algorithm – Outline

Main Points

- Distinguish between mutator and GC access to a page
  - Before GC: Snapshot RR page bits
  - After GC: Clear RR bits set by GC activity
- Adjust timestamps by GC pause length (after GC)
- Keep track of mispredicted-as-dead when doing GC
- RR polling thread disabled during GC

The Algorithm – Pseudocode (1)

Estimate Yield

```
rr_list = get_rr_pages(heap_start, heap_end)
for page in rr_list:
    timestamp[page] = current_time
dead_count = 0
limit = OLD_YOUNG_RATIO * (current_time - last_full_gc)
predicted_dead = {}
for page in range(heap_start, heap_end):
    age = current_time - timestamp[page]
    if age >= limit and not mispredicted_dead[page]:
        predicted_dead[page] = True
        dead_count += page_size
```

The Algorithm – Pseudocode (2)

Skip or Collect

```
if dead_count < SKIP_THRESHOLD * heap_size:
    dead_count = max(dead_count, min_expansion)
    expand_heap(dead_count)
    total_expansion += dead_count
else:
    do_regular_gc()
    total_expansion = try_to_shrink_heap(total_expansion)
    update_mispredicted_dead()
    if heap_relocated:
        heap_start, heap_end = get_heap_boundaries()
clear_rr_pages(heap_start, heap_end)
for page in range(heap_start, heap_end):
    timestamp[page] += gc_time
last_full_gc = current_time
```

Implementation – MRE part

Sun's HotSpot JVM

- Uses a generational GC Scheme (young, old, permanent)
- Young generation managed by copying GC
- Major GC upon space exhaustion of old space
- Three different GC implementations
    - Compressor
    - HotSpot Compactor
    - Mapping Collector
- Algorithm works with all three GC implentations.

Implementation – OS part

Linux 2.6 kernel module

Exposes `/proc/ref/bits` write-only file:

- Written to by the polling thread in the MRE (10ms interval)
- Data written: address range, result pointer
- The kernel writes RR pages in requested range into result area

Kernel modifications

- Minor modification required for handling of the per-page *os-cleared* and *mre-cleared* bits.

Performance – Conditions

Platform

- Intel Core 2 Duo (dual-core), 2GB RAM, Linux 2.6.17 kernel
- 16 Java programs
- Mix of standard benchmarks & open-source Apps

Clustering

- Most clusters < 4KiB, *but average size* $> 200 KiB$
- $\geq 50\%$ of dead space fully covered by pages

GC yield

- 9 of 16: $< 5\%$ – *low yield group*
- 7 others: $> 23\%$

Performance – Impact

Prediction Accuracy

For 5% skip threshold: 4% of heap size

Prediction Cost

With skip threshold $=$ 0% (no skips):

- $<$ 4% on average
- $<$ 10% max

Impact on Applications

For the low-yield group:

- Speedup: $\geq$ 44% on average, depending on GC
- Skip rate: 75% on average

For others:

- Maximum overhead: 3%

End of Presentation

Thanks for your Attention!