Heap Compression for Memory-Constrained Java

CSE Department, PSU

G. Chen M. Kandemir
N. Vijaykrishnan M. J. Irwin

Sun Microsystems

B. Mathiske M. Wolczko

OOPSLA’03

October 26-30 2003

Overview

PROBLEM: Typical portable devices support less than 1MB of memory *(2003)*

SOLUTION: Reduce execution heap size of current embedded JVMs

- ✓ Extending existing garbage collection methods to allow for
  - ✓ **Heap Compression**: When current execution cannot continue, compress heap
  - ✓ **Lazy Allocation**: Exploit limited usage of large object components

Results Summary

Compared to Mark-Compact Garbage Collection

- Analysis of ten applications suitable for mobile devices
  - One method reduced heap demand by 35% on average (16-54%)
  - Introduced less than 2% performance degradation due to compression/decompression

Garbage Collection Overview

Existing Methods

- Mark-Sweep (MS)
  - Phase One: (Mark) Traverse reference tree, marks objects reachable from root.
  - Phase Two: (Sweep) Scans heap, put all unmarked objects into a free-table
  - Results in "Fragmentation Problem" -- live objects and free space interleaved.
  - Requires heap size larger than application footprint

Garbage Collection Overview

Existing Methods

- Mark-Compact (MC)
  - Addresses the "Fragmentation Problem”
  - Slides all marked objects to one end of the heap
  - Requires updates to all object references

Mark-Compact-Compress (MCC)

- Allows application to run with heap smaller than footprint
- Mark Phase: Same as existing, but counts size of live objects.
  - Allocate new object based on free space
  - Object smaller: Proceed without compression
  - Object larger: Compress all live objects on the heap

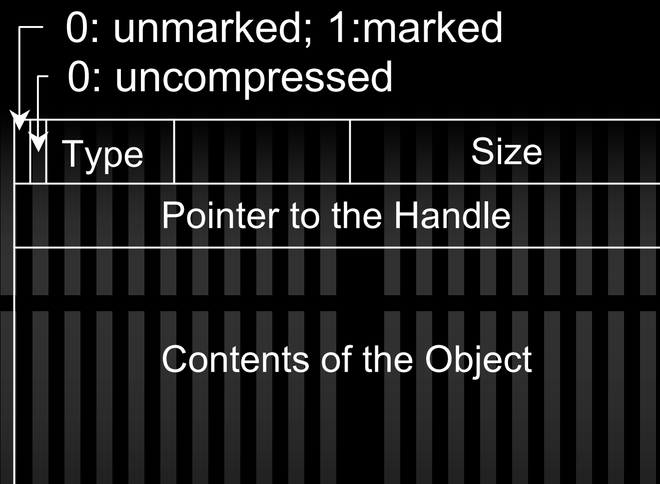
Mark-Compact-Compress (MCC)

- Zero Removal Compression
  - Based on observation; large portion of memory locations contain only zeros
- Compression / Decompression Overhead
  - Overhead not incurred frequently
    - Heap demand occurs during a very short period of execution
    - Decompression cost amortized over multiple accesses.
    - Many objects never accessed after compression

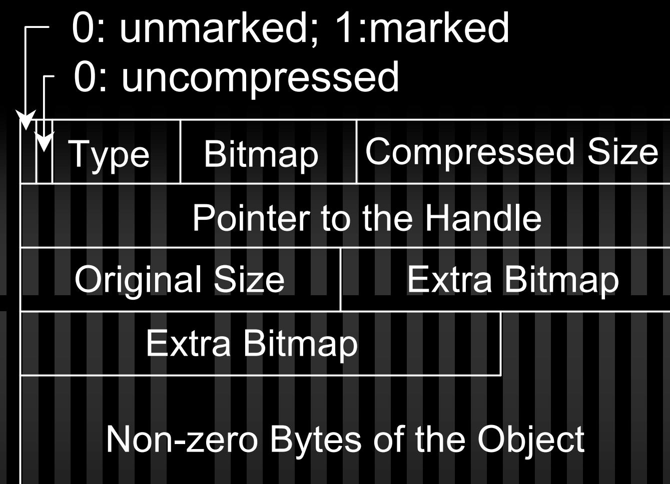
Compression Algorithm Selection

- Any compression/decompression algorithm will work
- For best results, algorithm should satisfy
  - Compressor should have good compression ratio
  - Compression / decompression should be fast
  - Neither compressor nor decompressor should require large working area

Compression
Heap Objects



Uncompressed object



Compressed object

- Each bit in the bitmap corresponds to a byte of the objects data in the uncompressed format.
- 0-bit indicates a zero byte; not stored in the compressed format
- 1-bit indicates non-zero byte; contents stored in non-zero bytes
- When required: Scan the entire heap, compress all uncompressed objects.

Decompression

Object Access

- ✓ Allocate free block size of uncompressed object. (invoke GC if required)
- ✓ Decompress the object
- ✓ Update the pointer in the handle
- ✓ The compressed object is marked for GC

Decompression

Mark Phase

- ✓ Collector traverses reference tree
- ✓ Visit to compressed object
  - ✓ Check object class for referenced fields
  - ✓ If required, decompress, retrieve contents
- ✓ Fields are decompressed one-by-one
- ✓ Decompressed data is discarded
- ✓ Never invokes allocation

Lazy Allocation

- ✓ Different portions of an object allocated on-demand
- ✓ Only applied to large arrays to minimize runtime overhead
- ✓ MCCL garbage collection

- Decompression of large objects
  - Inefficient for a few fields
- Break large objects into (1KB max) “subobjects”
- Currently only arrays are broken down

Minimum Heap Sizes

without out-of-memory exceptions

Benchmark	Minimum Heap Size (KB)						Normalized against MC (%)				
	MS	MC	MCL	MCC	MCCL	MCCL+	MS	MCL	MCC	MCCL	MCCL+
Auction	128	83	76	72	62	58	154.2	91.6	86.8	74.7	69.9
Calculator	55	40	40	34	34	32	138.5	100.0	85.0	85.0	80.0
JBrowser	260	226	196	195	164	157	115.0	86.7	86.2	72.6	69.5
JpegView	127	85	85	79	77	64	149.4	100.0	92.9	90.6	75.3
ManyBalls	57	35	35	31	31	29	162.9	100.0	88.6	88.6	82.9
MDoom	178	124	71	114	76	57	143.5	57.3	91.9	61.3	46.0
PhotoAlbum	96	55	55	50	50	46	174.5	100.0	90.9	90.9	83.6
Scheduler	56	37	36	32	32	31	151.4	97.3	86.5	86.5	83.8
Sfmap	292	162	118	175	91	78	118.5	72.8	108.0	56.2	48.1
Snake	72	42	42	35	35	33	171.4	100.0	83.3	83.3	78.6
Average:	132	90	75	81	65	59	147.9	90.5	89.2	79.0	65.6

Compressions / Decompressions

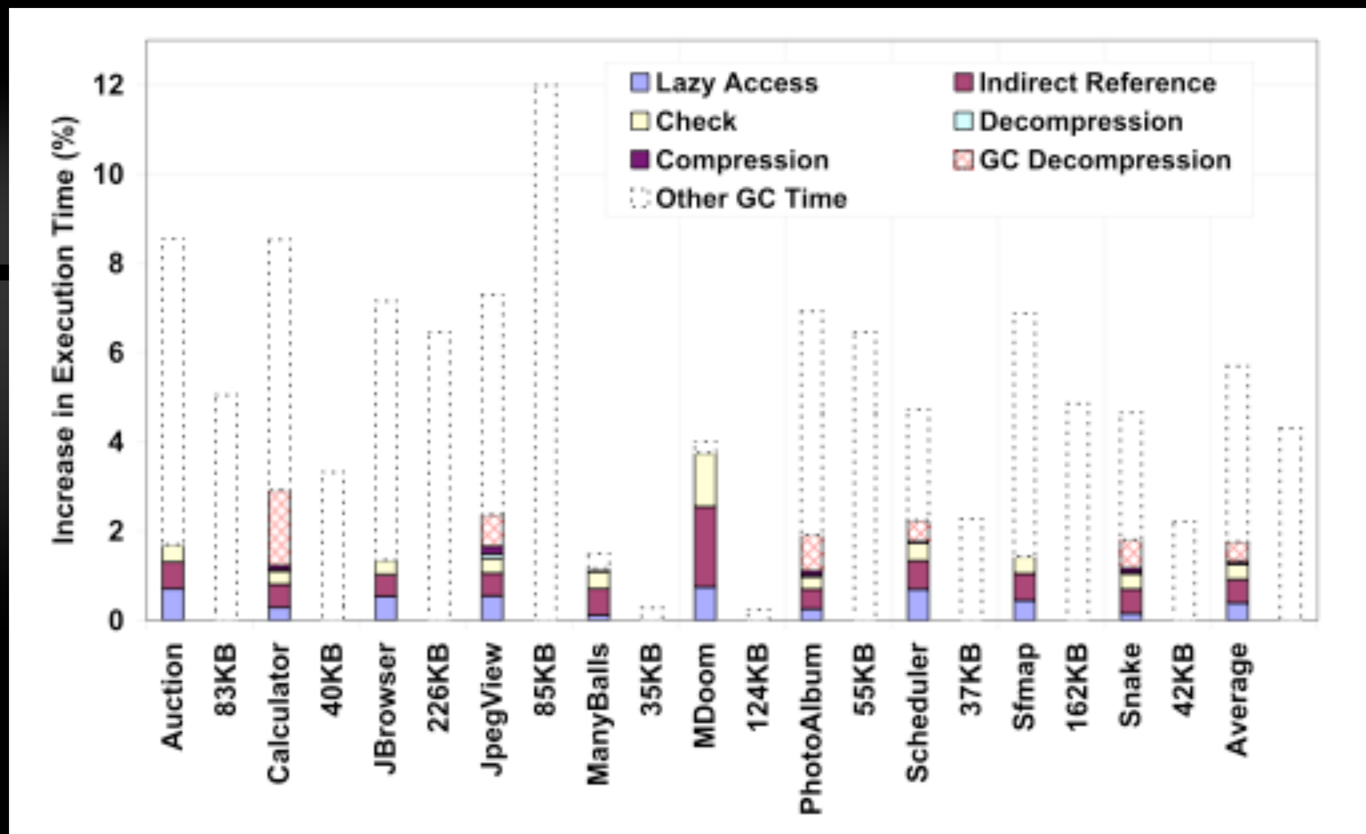
Benchmark	Total Number of Comp.	Decompressions Total Number	Decompressions % of Total Comp.	Number of Objects Decompressed N times $N=2$	Number of Objects Decompressed N times $N=1$
Auction	530	131	24.72%	0	131
Calculator	226	46	20.35%	5	36
JBrowser	939	261	27.80%	0	261
JpegView	1926	867	45.02%	228	411
ManyBalls	264	83	31.44%	18	47
MDoom	365	207	56.71%	14	179
PhotoAlbum	235	54	22.98%	0	54
Scheduler	172	36	20.93%	0	36
Sfmap	1429	90	6.30%	0	90
Snake	234	38	16.24%	0	38
Average:	632	181	28.69%	26	128

(a) MCC

Benchmark	Total Number of Comp.	Decompressions Total Number	Decompressions % of Total Comp.	Number of Objects Decompressed N times $N=2$	Number of Objects Decompressed N times $N=1$
Auction	530	131	24.72%	0	131
Calculator	226	46	20.35%	5	36
JBrowser	939	261	27.80%	0	261
JpegView	0	0	N/A	0	0
ManyBalls	264	83	31.44%	18	47
MDoom	0	0	N/A	0	0
PhotoAlbum	235	54	22.98%	0	54
Scheduler	172	36	20.93%	0	36
Sfmap	0	0	N/A	0	0
Snake	234	38	16.24%	0	38
Average:	260	65	24.96%	2	60

(b) MCCL

Runtime Overhead
with minimum heap for MC



* Bar on left is MCCL, bar on the right is MC