Evaluation of thread-local garbage collection

A. Yu. Filatov[1] V. V. Mikheev[2]

[1]Novosibirsk State University
a.filatov@g.nsu.ru

[2]Compilers and Programming Languages Lab
Huawei Novosibirsk Research Center
mikheev.vitaly@huawei.com

Background

Experimental JVM based on Huawei JDK

Background

Experimental JVM based on Huawei JDK

- Ahead-of-time (AOT) compilation: Java bytecode → machine code
  - No Closed World assumption (supports dynamic classloading)
  - Whole Program analysis
  - State of the art optimizations:
    - Conditional devirtualization
    - Partial escape analysis
    - Profile-guided optimizations
    - Graph-coloring register allocation

Background

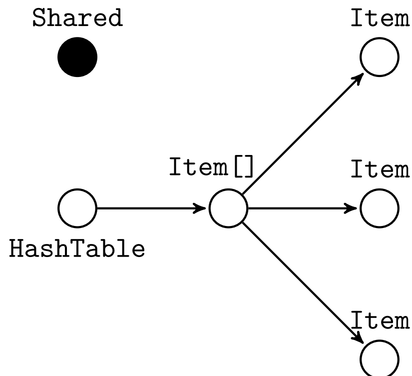
Experimental JVM based on Huawei JDK

- Ahead-of-time (AOT) compilation: Java bytecode → machine code
  - No Closed World assumption (supports dynamic classloading)
  - Whole Program analysis
  - State of the art optimizations:
    - Conditional devirtualization
    - Partial escape analysis
    - Profile-guided optimizations
    - Graph-coloring register allocation
- Elaborate runtime:
  - Thread-specific *allocation* from heaplets
  - Parallel concurrent mark-sweep generational garbage collection
  - Tiered execution of dynamic code: interpreter + non-optimizing JIT

Motivation

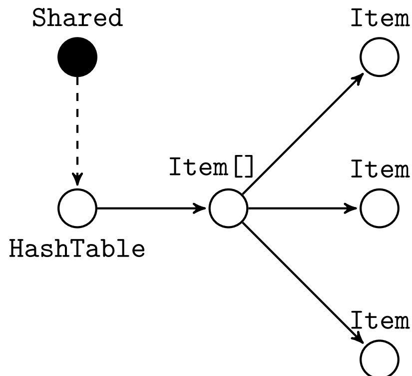
Write barriers

Run-time partitioning: *local* and *global* objects



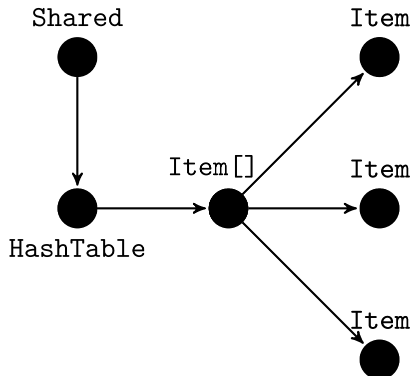
Write barriers

Run-time partitioning: *local* and *global* objects



Write barriers

Run-time partitioning: *local* and *global* objects



Overview

Main points:

- Local heaps: synchronization-free object allocation and *reclamation*
- Local GC as a co-routine – safe point free context
- Stop-the-world overheads are reduced

Overview

Main points:

- Local heaps: synchronization-free object allocation and *reclamation*
- Local GC as a co-routine – safe point free context
- Stop-the-world overheads are reduced

Expected benefits:

- Scalability
- Data locality and NUMA-awareness
- Better throughput

Heap layout

Blocks:

- Small (16 Kb): objects of the same size. First-fit.
- Medium (16 Kb): 104 bytes - 8Kb objects. Best-fit.
- Large: pagewise allocation.

Heap layout

Blocks:

- Small (16 Kb): objects of the same size. First-fit.
- Medium (16 Kb): 104 bytes - 8Kb objects. Best-fit.
- Large: pagewise allocation.

$$block.owner = \begin{cases} \perp & \text{if block is shared} \\ ThreadID & \text{if block belongs to local heap} \end{cases}$$

Heap layout

Blocks:

- Small (16 Kb): objects of the same size. First-fit.
- Medium (16 Kb): 104 bytes - 8Kb objects. Best-fit.
- Large: pagewise allocation.

$$block.owner = \begin{cases} \perp & \text{if block is shared} \\ ThreadID & \text{if block belongs to local heap} \end{cases}$$

Source of confusion: local/global **objects** vs. local/shared **blocks**

coloring — allocation/reclamation

Thread-local garbage detection

Tracing thread-local GC:

- **Mark** reachable objects – thread's stack and live variables
- **Reclaim** unreachable objects
- **Sweep** mark bit

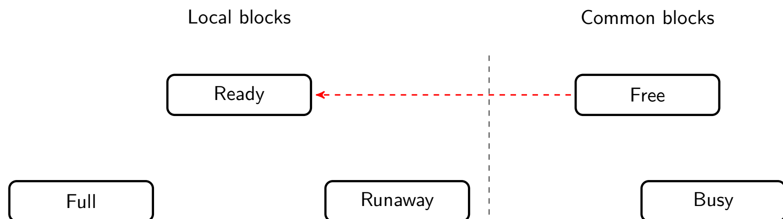
Incremental:

- Marking stops at global objects
- Reclaim only selected subset of blocks

$(\frac{dead\ objects}{block\ size} < 25\%) \rightarrow$ **Full** block

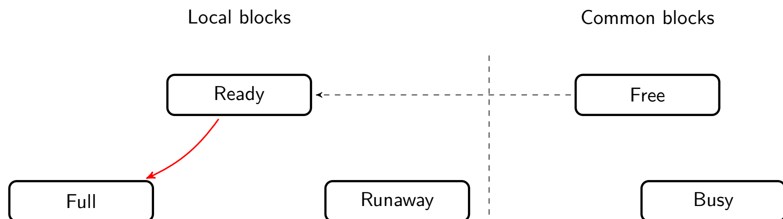
$(\frac{global\ objects}{block\ size} > 50\%) \rightarrow$ **Runaway** block

Block state transitions



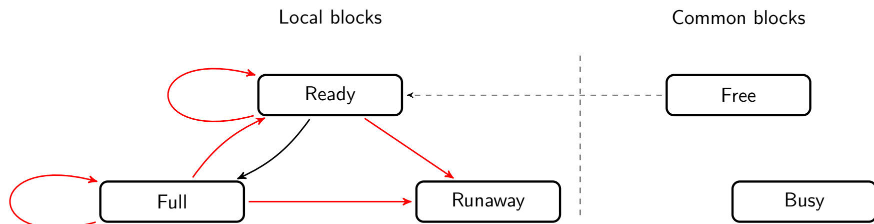
Block allocated

Block state transitions



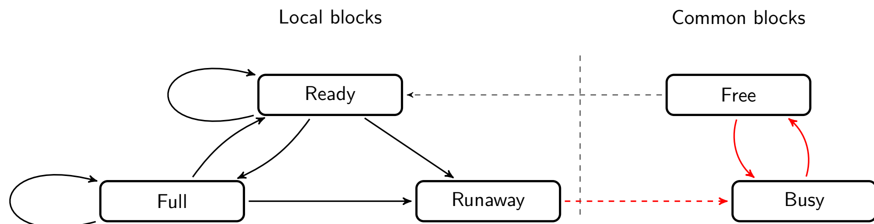
Block exhausted

Block state transitions



Thread local GC

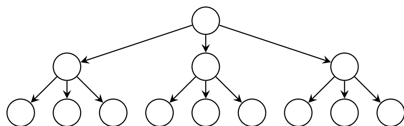
Block state transitions



Global GC

Block promotion

Observation: **Marking** traverses large thread-local alive structures.



Idea: "Promote" objects to speed-up marking and introduce conservatism

Performance measurements

System:

- `CentOS Linux 7, 3.10.0-957.e17.x86_64`
- `Intel Xeon Gold 6130 @ 2.10Ghz x 128` (4s, 16c/s, 2t/c)
- 500 Gb RAM

Performance measurements

System:

- `CentOS Linux 7, 3.10.0-957.e17.x86_64`
- `Intel Xeon Gold 6130 @ 2.10Ghz x 128` (4s, 16c/s, 2t/c)
- 500 Gb RAM

Experimental Huawei JDK, Java 8 u242:

- Base memory manager
- TLGC
- TLGC +promote

Object Avalanche

```
static Double avalanche(Double bound) {
  Double result = 1d;
  for (Double f = 0d; f < bound; f++)
    result += f / (result + 1);
  return result;
}
```

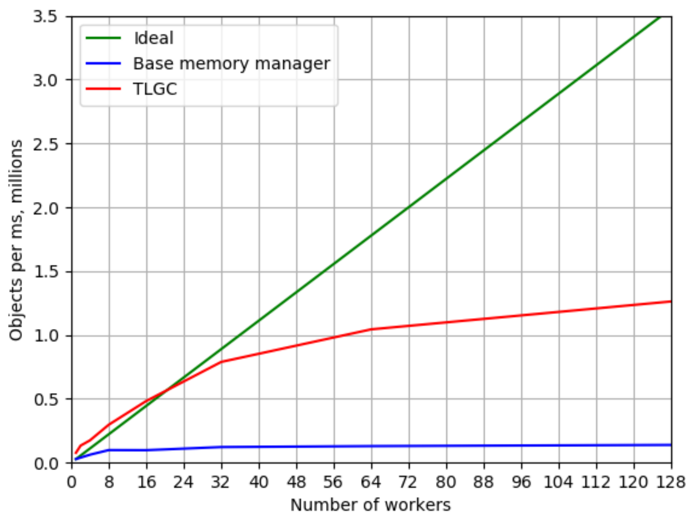
-Xmx10g, $bound = 5 \cdot 10^7$

Object Avalanche

Lower – better, milliseconds

GC / Workers	Base memory manager		TLGC	
	Execution	STW	Execution	STW
1	1800 (1.00x)	153	649 (1.00x)	4
2	2531 (1.40x)	244	750 (1.15x)	2
4	3217 (1.78x)	241	1146 (1.76x)	49
8	4069 (2.25x)	385	1348 (2.07x)	42
16	8197 (4.55x)	865	1658 (2.55x)	32
32	13116 (7.28x)	856	2029 (3.12x)	224
64	24730 (13.73x)	1781	3066 (4.72x)	127
128	46179 (25.64x)	1964	5071 (7.80x)	407

Object Avalanche



Factorie

http://www.benchmarks.scalabench.org

Benchmark uses toolkit for deployable probabilistic modeling to extract topics using Latent Dirichlet Allocation.

- Full heap limit – 1000 Mb. Close to application working set size.
- Local heap limit – 46 Mb.
- "Gargantuan" running mode.

Lower – better, seconds

Base memory manager		TLGC		TLGC +promote	
Execution	STW	Execution	STW	Execution	STW
670 (1.00)	383	2797 (4.17)	10	297 (0.44)	11

Apache Spark: examples.mllib.DenseKMeans

main/scala/org/apache/spark/examples/mllib/DenseKMeans.scala

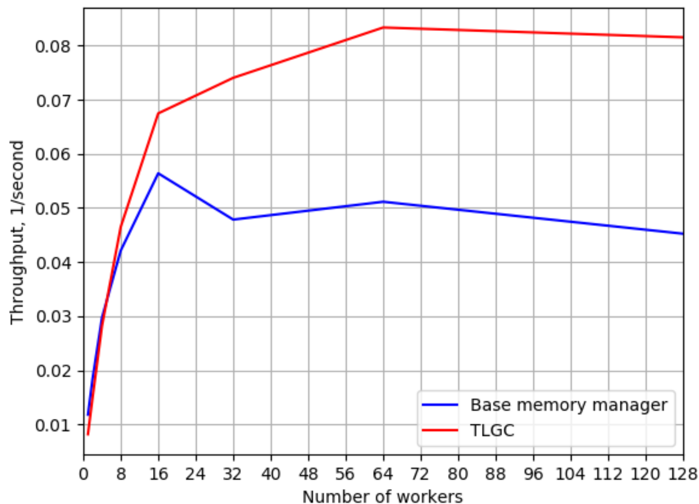
```
val examples = sc.textFile(params.input).map {
 line => Vectors.dense(line.split(' ').map(_.toDouble))
}.cache()

val numExamples = examples.count()
...
val cost = model.computeCost(examples)
```

Apache Spark: examples.mllib.DenseKMeans

Lower – better, milliseconds

Workers \ GC	Base memory manager		TLGC	
	Execution	STW	Execution	STW
1	84690	1618	122042	2926
2	54109	1420	66696	1714
4	33651	775	35454	1312
8	23740	734	21513	626
16	17731	980	14822	550
32	20899	1250	13507	1071
64	19553	1274	12000	484
128	22103	1220	12267	380

Apache Spark: examples.mllib.DenseKMeans



Conclusions

- Memory manager key parts:
  - Write barriers
  - Thread-local heaps
  - Mark-sweep incremental local GC

Conclusions

- Memory manager key parts:
  - Write barriers
  - Thread-local heaps
  - Mark-sweep incremental local GC
- Observed benefits:
  - Scalability
  - Lower STW pause
  - Better throughput

Conclusions

- Memory manager key parts:
  - Write barriers
  - Thread-local heaps
  - Mark-sweep incremental local GC
- Observed benefits:
  - Scalability
  - Lower STW pause
  - Better throughput
- Possible improvements:
  - Incremental marking
  - Owner-driven compaction
  - NUMA-awareness

Q & A