Z-Rays: Divide Arrays and Conquer Speed and Flexibility

Jennifer B. Sartor et al.

Presented by Yuhao Zhu

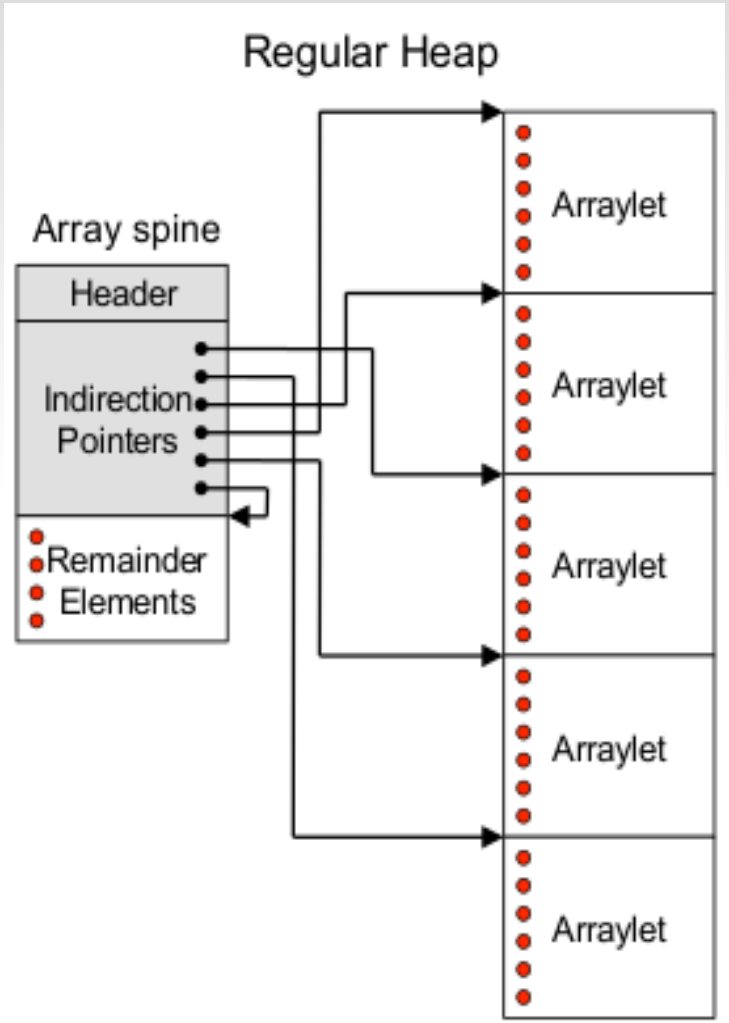
CS 395T

Motivations

- Contiguous implementation of arrays incurs fragmentation (wastes space), and is GC-unfriendly, especially latency
- Discontiguous implementation overcomes above problems, but brings about the throughput issue due to the high overhead of indirection

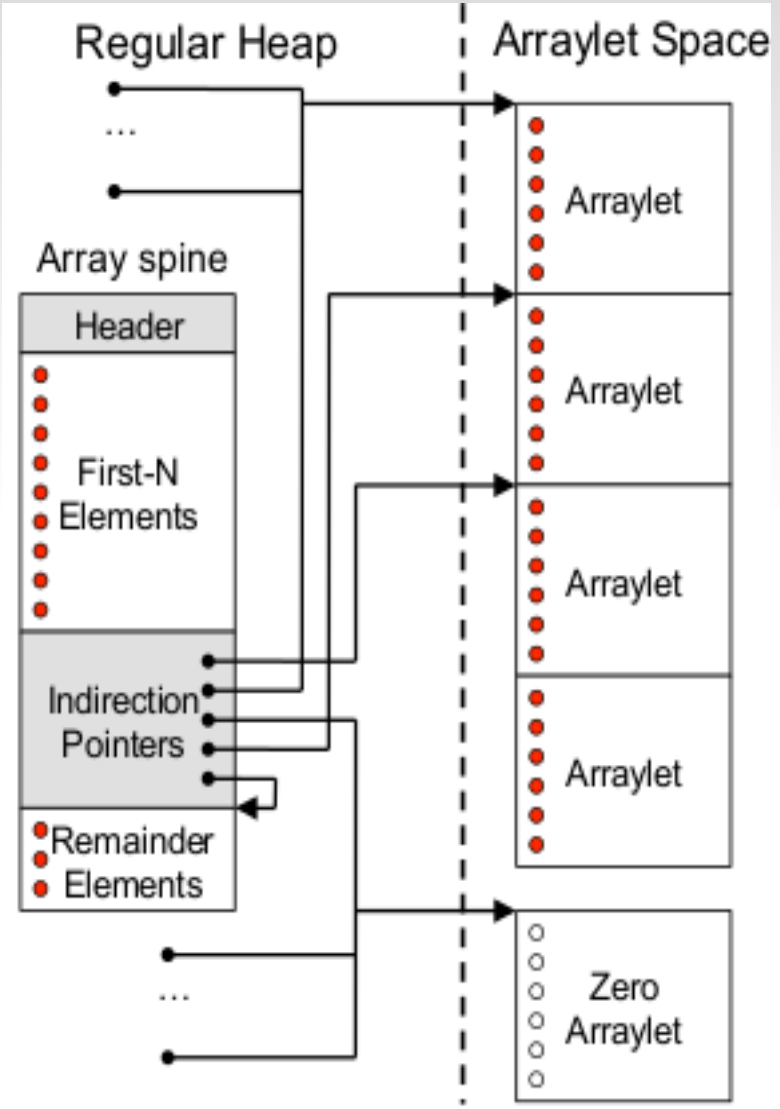
Discountiguous Arrays

- Organization
  - Header
  - Indirection pointers
  - Remainders
- Why does it work?
- Why does it not work?



Z-rays implementation

- 5 optimizations in a moment
- Separate arraylet space
  - sub-space of the heap
  - Collected under the control of its parent spine
  - Each arraylet has its own liveness bit
- Spines are in the nursury space and collected as normal



First-N

- Nearly 90% of all array accesses occur at access positions less than 4KB. So inline them in the spine and access without indirection
- Most effective optimization
- Addressed the performance issue of basic discontiguous design

Lazy Allocation

- Space optimization
- Employs an immutable zero arraylet, to which all indirection pointers are pointing upon creation
- Need to be performed atomically due to possible race condition of multiple threads

Zero Compression

- Space optimization, utilizes the zero arraylet
- Reinstall the indirection pointer to the zero arraylet when all elements in an arraylet are zeros
- Performed during GC time
- Incurs additional indirection and scanning operatins, but compensated by the reduction in the memory cost

Fast Array Copy

- Discontiguous arrays make array copy complicated
- One optimization is to hoist the indirection operation outside of the loop when performing sequential copy
- *One* indirection instead of *n*

Copy-on-Write

- Space optimization
- A generalization of lazy allocation
- Only create the private instance of an arraylet after first write
- Realized by tainting the least significant bit of the indirection pointers pointing to the shared arraylet

Implementation Notes

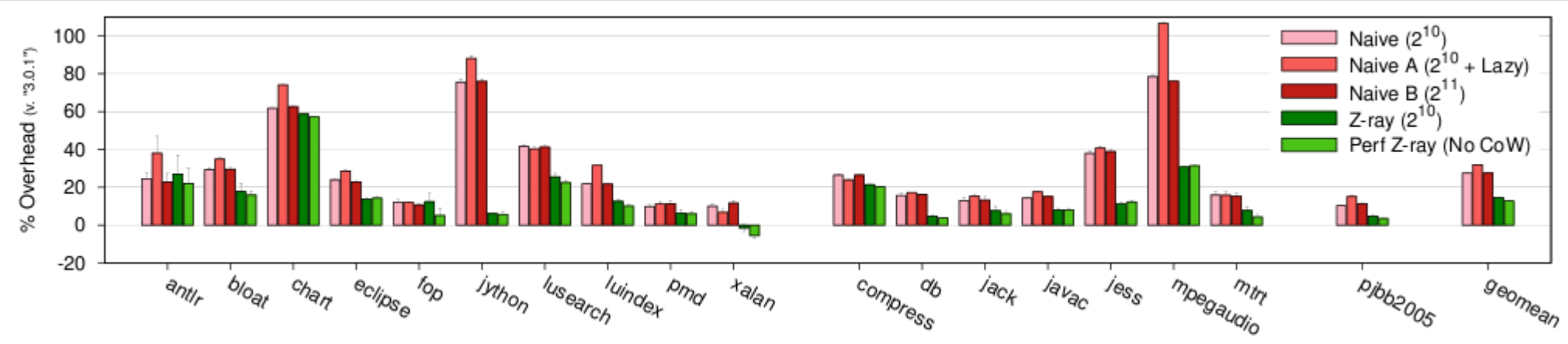
- Read/Write Barriers
- The arraylet space is non-moving, and the age of an object is indicated by its parent spine
- Promote survived spines into mature space, which effectively promotes corresponding arraylets
- What's the arraylet space allocator? Any comment?
  - Do we mark the liveness bits of arraylets whose source spine is in nursery space?
  - If no, what's the implication?

Evaluations

- Benchmark characteristics

	Allocation			Heap		Accesses					Array Copy	
	MB/	Array %		Composition		per	write %		read %		byte	%
Benchmark	μsec	all	prim.	MB	%	μsec	fast	slow	fast	slow	μsec	$>N$
antlr	72	83	80	12	52	157	9.3	7.6	73.5	9.6	52	23
bloat	77	65	60	18	51	264	1.0	0.4	97.8	0.8	52	0
chart	23	49	48	18	49	320	5.3	7.1	49.8	37.8	44	76
eclipse	57	75	55	38	57	373	4.6	1.4	89.4	4.7	30	25
fop	11	34	26	19	47	94	1.7	0.1	97.3	0.9	5	0
hsqldb	29	38	21	67	31	463	0.7	0.3	98.1	0.9	5	16
jython	125	77	66	24	51	584	1.2	0.3	98.0	0.6	132	3
luindex	32	40	36	12	52	186	28.6	0.2	70.7	0.5	21	0
lusearch	201	87	82	15	57	699	14.5	0.5	84.1	1.0	31	8
pmd	156	33	1	23	45	419	0.9	1.01	96.2	1.9	7	69
xalan	766	88	52	31	73	342	7.5	0.24	91.5	0.7	41	0
compress	24	100	100	4	57	191	12.9	22.5	25.3	39.3	0	0
db	4	64	9	11	56	48	0.8	8.9	65.8	24.4	15	99
jack	28	32	26	6	51	92	4.8	0.2	94.3	0.7	49	0
javac	22	49	42	12	41	106	7.3	0.4	90.9	1.4	6	4
jess	75	47	0	7	54	197	1.9	0.2	97.1	0.8	66	0
mpegaudio	0.2	15	6	3	52	669	14.3	0.1	85.5	0.1	35	0
mtrt	30	25	18	9	42	267	4.3	0.2	95.2	0.3	0	0
pjbb2005	70	63	42	193	64	1109	2.4	0.3	96.5	0.8	271	0
min	**0.2**	**15**	**0**	**3**	**31**	**48**	**0.7**	**0.1**	**25.3**	**0.1**	**0**	**0**
max	**766**	**100**	**100**	**193**	**73**	**1109**	**28.6**	**22.5**	**98.1**	**39.3**	**271**	**99**
mean	**47**	**56**	**40**	**-**	**52**	**338**	**6.4**	**2.6**	**84.6**	**6.4**	**45**	**17**

Evaluations (cont.)



- COW degrades the performance
  - Due to the maintenance of barriers
- Z-rays could even IMPROVE the performance
  - The indirection overhead is compensated by the reduction of collection time

Evaluations (cont.)

- How does Z-ray affect the performance?
  - Mutator: indirection beyond First N
  - Collector: varies significantly
    - - indirection
    - + improvements through space efficiency

Benchmark	Total Overhead (%) C2D	Atom	C2D Overhead Breakdown (%) Ref.	Prim.	Mutator	GC
antlr	22.0 ±8.2	37.7 ±12.3	-3.2	14.4	17.9	98.2
bloat	15.9 ±2.0	28.7 ±8.6	4.3	11.4	14.2	73.9
chart	57.2 ±0.4	54.9 ±0.3	0.2	57.0	61.4	-6.9
eclipse	14.2 ±1.2	24.9 ±7.3	1.9	10.3	15.7	-28.1
fop	5.1 ±3.7	19.0 ±9.0	8.9	14.2	4.4	33.6
hsqldb	23.8 ±24.5	7.5 ±1.8	2.2	33.9	26.9	12.9
jython	5.7 ±1.1	12.6 ±3.2	2.6	2.8	5.0	60.9
lusearch	22.4 ±1.3	24.0 ±0.9	4.2	23.9	22.6	18.3
luindex	10.1 ±0.9	14.9 ±1.0	1.3	10.4	9.6	26.8
pmd	6.0 ±1.3	7.2 ±1.2	5.5	0.8	7.9	-19.4
xalan	-5.5 ±1.3	11.1 ±2.7	-4.8	-0.7	2.0	-56.0
compress	20.2 ±0.3	51.2 ±0.4	0.4	20.3	21.9	-82.9
db	3.7 ±0.1	14.0 ±0.1	3.4	-0.4	3.8	-4.0
jack	5.9 ±1.6	7.6 ±1.1	0.3	4.7	6.6	-15.6
javac	8.0 ±0.6	11.5 ±1.2	2.2	5.9	8.3	4.2
jess	12.2 ±1.0	17.0 ±2.8	10.3	1.4	12.0	29.0
mpegaudio	31.4 ±0.4	44.1 ±0.6	2.3	14.4	31.2	358.0
mtrt	4.2 ±1.7	6.8 ±1.6	1.4	3.4	4.4	1.7
pjbb2005	3.4 ±0.5	5.1 ±2.5	-0.1	0.6	3.6	0.6
min	**-5.5**	**5.1**	**-4.8**	**-0.7**	**2.0**	**-56.0**
max	**57.2**	**54.9**	**10.3**	**57.0**	**61.4**	**4.2**
geomean	**12.7**	**20.2**	**2.2**	**10.1**	**13.3**	**-11.3**

Evaluations (cont.)

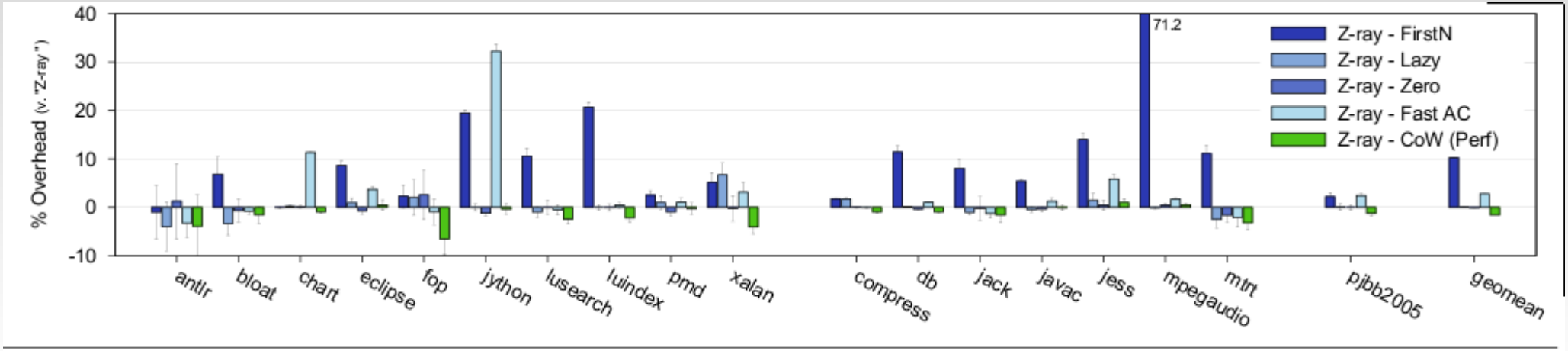


Figure 5. Overhead taking away each optimization from our Z-ray configuration.

- First-N is the most significant optimization
- Fast array copy benefits benchmarks with frequent array copying very much
- COW degrades performance

Discussion

- Why is it called Z-rays???
- Any concurrency to explore?
- How to configure Z-rays for different design goals?
- Any further optimization?