Garbology

A Study of How Objects Die

Raoul Veroy
Samuel Guyer
with ***Karl Cronburg****

Tufts University

Onward! 2017

Overview

- Empirical study of garbage collection
- Focus on how programs dispose of objects
- Interesting findings
- So what? Why are we doing this?

We envision a new approach to
GC algorithms

GC is easy

- Theory: find all reachable objects using a graph search
- Practice: good performance is hard
- Key insight: program behavior is not random
- There are patterns we can exploit
  e.g., generational, garbage first, age-based, type-directed, etc.

Finding Patterns

- Prior work: object demographics
  Example: classify lifetimes into short, medium, long

- Lots of work on allocation patterns, heap structure

- Conspicuously missing: how do objects die?

What program actions create garbage?

Hard Problem

- No “free” operation
  How do we pinpoint when objects die?

- Tool: GC tracing

- Record every heap event
  Allocations, pointer updates, stack references

- Breakthrough: Merlin algorithm
  Computes precise death times for all objects

How?

Merlin

- Key insight: time-stamping
  *Timestamp objects when they **might** become unreachable*

```
o.f = p;
o = p.next();
```

- At GC time, last time stamp is the time of death
  Death times discovered in retrospect

- Problem: stack updates are very frequent

- Compromise:
  Timestamp stack refs at each allocation site

GCTrace tool

- Built in JikesRVM
  - Timestamp heap modifications online
  - Timestamp stack at allocation sites

- Good: New insights — unprecedented precision

- Issue: time is measured in allocations
  Fine for a lot of memory management research

- But what does it mean?

Allocation Time

- Cannot be (easily) tied back to program structure
- Is actually fairly coarse
  A lot of code can go by between allocations
- Gives unnecessarily bad view of predictability

Allocation lifetimes

```
void foo() {
    Object o = new Thing();
    do_something(o);
    // o dies here
}
```

- Every o instance dies in the same place
- BUT possibly wildly different lifetimes
- Why?
- Any number of allocations in `do_something()`

Elephant Tracks

- Key idea: tick the clock at method entry/exit
  - 10-20X more frequent than allocations
  - Only need to timestamp local refs

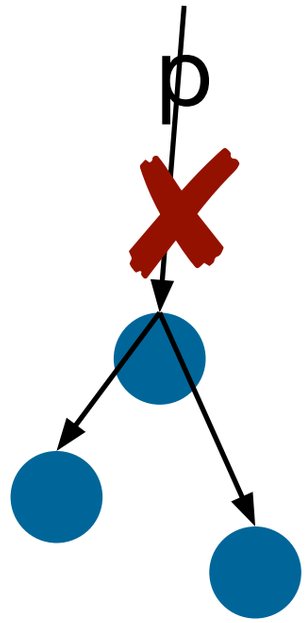
- Bonus: time means something!

- Method time: each tick is a calling context
  Can localize death events to a place in the program

Methodology

- Run ET on benchmarks
  Caveat: the usual suspects — are they representative?
- Collect program traces
  Including precise death times
- Analyze the trace: where do objects die and why?
- Classify information in various ways

Clusters

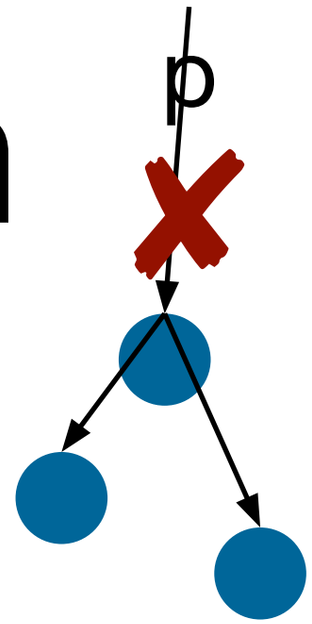


- One event often kills a cluster of objects
  e.g., discard a reference to a data structure

- Finding: clusters are usually small
  Programs dispose of data structures in a piecemeal way

benchmark	1	2	3	4	5	6	7	8	9	10	11	12	13+
201_compress	2983	1205	156	4	2	1							
202_jess	3733934	2091909	3420	99	18		2						
205_raytrace	5839977	235636	15797	3112	68	1	2						
209_db	2907822	112389	171	5	15546	1	2						
213_javac	2859266	898705	157208	66884	13922	6591	3101	1121	393	230	187	124	3276
227_mtrt	6000607	268224	25106	4270	96	2	2						
228_jack	3632399	432625	124296	2166	628	1374	35	16	672	16	16		
avrora	1397610	161820	105905	220	29	12	11	7		1			
batik	730811	74379	28456	7612	3550	134	38	165	1	6		1	4
fop	1433033	573648	87696	7944	3569	461	187	97	18	25		1	5
luindex	267693	58206	8939	85	33	20	9	1		1		1	
lusearch	6905703	1516410	677400	289	131156	53267	6	6	2	2		1	3
specjbb	3785434	568349	6951	16	1								1
tomcat	8660718	925172	483161	11280	2372	7778	154	83	235	22	1	21	4
xalan	5737589	399461	60187	16352	4220	63	9	3		4		1	

Cause of Death

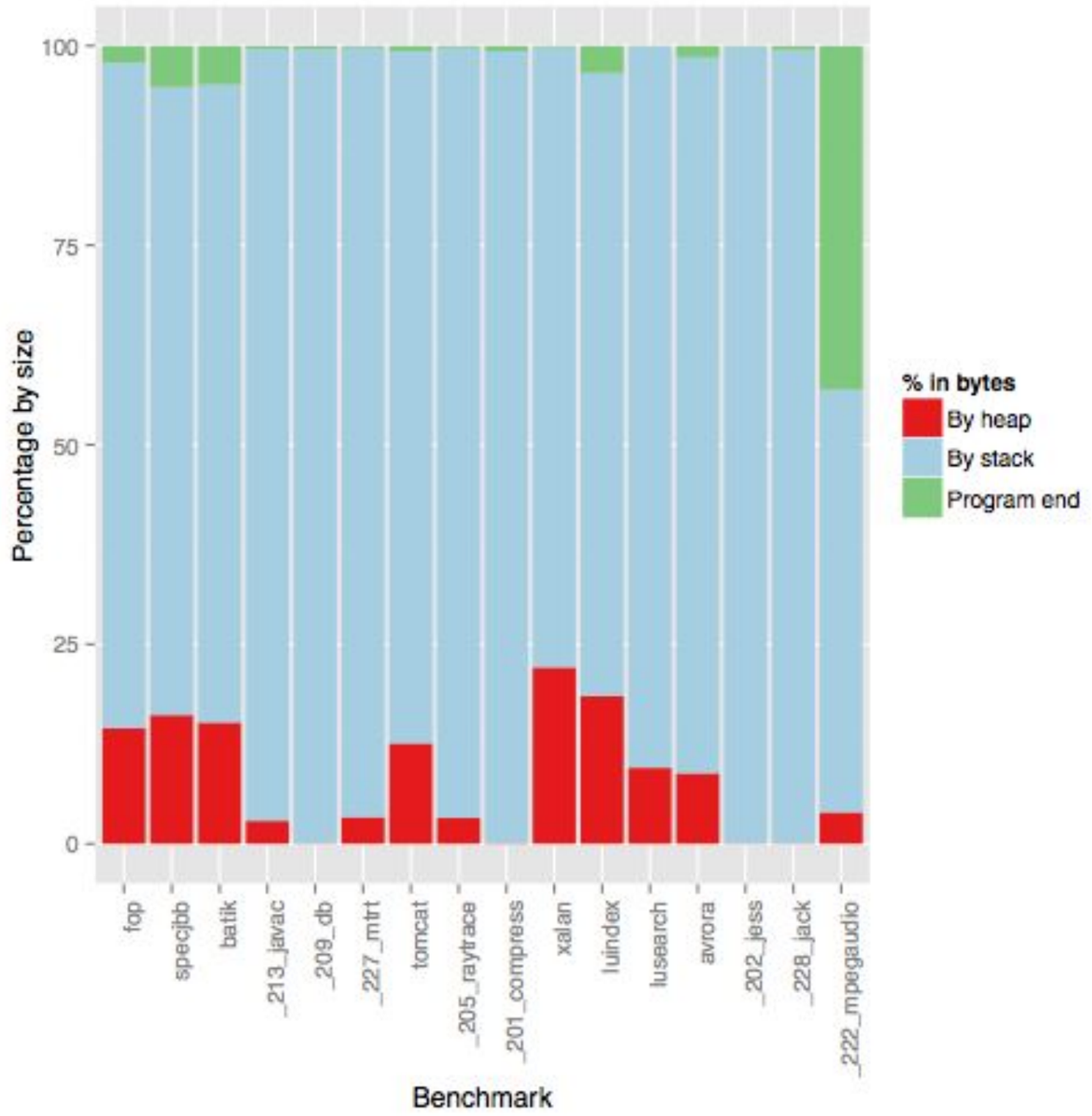


- Three broad categories

- Objects are made unreachable by…
    1. ***Die-by-heap***: a heap write
    2. ***Die-by-stack***: a stack reference going out of scope
    3. End of the program

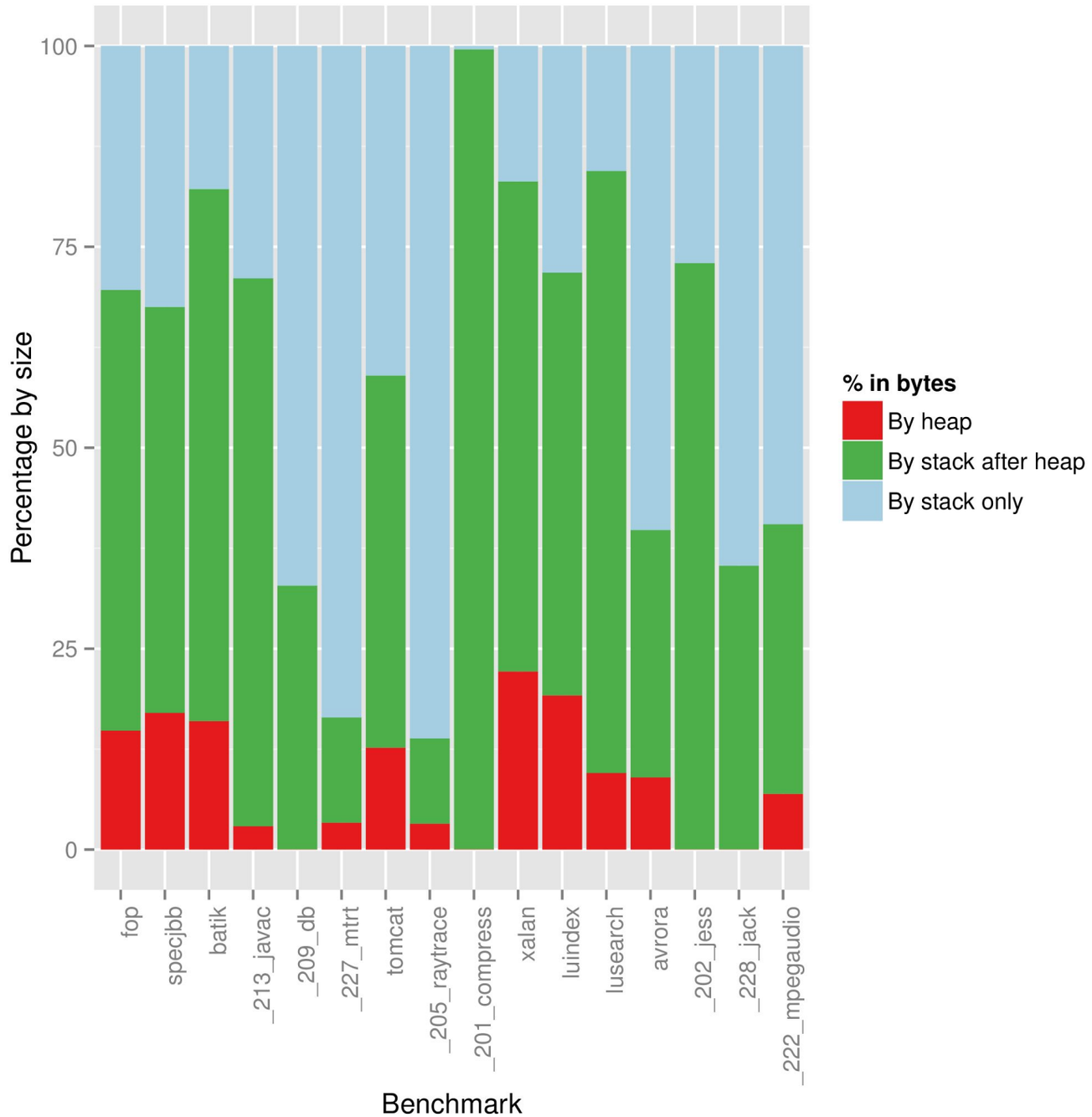
- All objects in a cluster have the same cause of death

- Finding: many objects die by stack
- What's going on?
- Common pattern:
  Remove an object from a container, process, drop
- *Program end* not interesting



- Split by-stack category: ***stack-after-heap***

- Finding: programs often disassemble data structures rather than disposing of them wholesale



Properties of clusters

- Size, types, lifetime (later)

- Interesting property: cyclic garbage
  A problem for reference counting collectors

- Compute SCC on objects in each cluster

- Finding: garbage cycles are very small

Cycles by type

Types in the cycle	Number of cycles	Number of objects	
		Minimum	Maximum
LinkedList$Link	106917	1	2
ElemContext	41898	1	1
ConcurrentLinkedQueue$Node	9230	1	1
HashMap, HashMap$1	5979	2	2
NativeMethodAccessorImpl, DelegatingMethodAccessorImpl	3907	2	2
LinkedBlockingQueue$Node	3003	1	1
HashSet, HashMap, HashMap$Entry	2173	4	4
PythonTree, ArrayList, Object	1988	4	4
DTMDefaultBaseTraversers$AncestorTraverser, SAX2DTM DTMAxisTraverser	1300	3	3
HashMap, HashMap$2	704	2	2
FileChannelImpl, AbstractInterruptibleChannel$1	132	2	2
NativeConstructorAccessorImpl, DelegatingConstructorAccessorImpl	131	2	2
SAX2DTM DTMDefaultBaseTraversers$ChildTraverser, DTMAxisTraverser	102	3	3
Cleaner	86	1	1
CSSLexicalUnit$SimpleLexicalUnit, CSSLexicalUnit$IntegerLexicalUnit	72	2	2
TypeVariableBinding, MethodBinding, TypeVariableBinding	71	3	3
ConcurrentHashMap, ConcurrentHashMap$EntrySet	70	2	2
SocketChannelImpl, AbstractInterruptibleChannel$1	65	2	2
XMLNSDocumentScannerImpl, XMLDocumentScannerImpl$TrailingMiscDispatcher	62	2	2
BufferUnderflowException	40	1	1
Label, Frame	24	2	2
ConcurrentHashMap, ConcurrentHashMap$KeySet	13	2	2

Lifetimes

- Traditional measures
  Allocation time, coarse categories (short, medium long)
- Hard to predict

- Finding: death sites are very stable
  We can predict where an object will die, even if we cannot predict the length of its life

Top Death Sites

Benchmark	Site	% of total allocation	# allocation sites
_201_compress	spec/benchmarks/_201_compress/Input_Buffer.readbytes	85.64	2
_209_db	spec/benchmarks/_209_db/Entry.equals	66.6	4
_205_raytrace	spec/benchmarks/_205_raytrace/Point.GetZ	58.05	18
_202_jess	spec/benchmarks/_202_jess/jess/Node2.runTests	56.6	9
_228_jack	spec/benchmarks/_228_jack/RunTimeNfaState.Move	46.5	20
lusearch	.../lucene/queryParser/QueryParserTokenManager.ReInit	40.29	3
_227_mtrt	spec/benchmarks/_205_raytrace/Point.GetZ	39.96	26
avrora	avrora/sim/radio/Medium$Receiver.earliestNewTransmission	29.8	3
fop	org/apache/xmlgraphics/ps/PSGenerator.formatDouble	24.49	118
luindex	.../lucene/analysis/standard/StandardTokenizerImpl.yyreset	23.78	1
tomcat	org/dacapo/tomcat/Page.stringDigest	23.69	29
specjbb	spec/jbb/StockLevelTransaction.process	22.56	2
xalan	org/apache/xalan/transformer/TransformerImpl.transform	20.4	49
_213_javac	spec/benchmarks/_213_javac/Type.tClass	17.4	79
batik	org/apache/batik/bridge/BridgeContext.finalize	13.77	222

Memory Flow

- Connect two ends:
  Allocation site and death site

- How many bytes flow between pairs?

- Finding: a small number account for a large fraction of memory use

Benchmark	Allocation site	Death site	Size (MB)	%	Min age(kB)	Max age(kB)
202_jess	Node2.appendToken	Node2.runTests	280.7	56%	0.05	50659
228_jack	RunTimeNfaState.Move	RunTimeNfaState.Move	122.18	46%	0	<1
201_compress	Decompressor.<init>	Decompressor.decompress	45.34	42%	71.91	83
avrora	MediumReceiver. earliestNewTransmission	MediumReceiver. earliestNewTransmission	31.91	28%	0	1
lusearch	QueryParser.parse	QueryParserTokenManager.ReInit	354.2	26%	0	4205
205_raytrace	PolyTypeObj.Intersect	Point.GetZ	60.59	23%	0	<1
specjbb	StockLevelTransaction.process	StockLevelTransaction.process	77.65	22%	0	46
luindex	FileDocument.Document	StandardTokenizerImpl.yyreset	10.14	21%	20.03	799
209_db	Database.set_index	Database.remove	33.93	20%	15.17	1061
tomcat	Page.stringDigest	Page.stringDigest	190.11	20%	0	3077
227_mtrt	PolyTypeObj.Intersect	Point.GetZ	38.84	14%	0	<1
fop	PSGenerator.formatDouble	PSGenerator.formatDouble	26.32	13%	0.03	17594
213_javac	Type.tClass	Type.tClass	44.2	13%	0	<1
batik	DacapoClassLoader.loadClass	DacapoClassLoader.loadClass	14.07	10%	0	49040
xalan	VariableStack.reset	VariableStack.reset	105.53	9%	319.37	27840

Conclusion

- Programs exhibit very strong patterns of memory use
- Program structure strongly predicts these patterns
- How can we exploit these patterns?
- New kinds of garbage collectors?

Thank
You!