Understanding the Connectivity of Heap Objects Technical Report: CU-CS-923-01 *

Martin Hirzel, Johannes Henkel, Amer Diwan
University of Colorado, Boulder

Michael Hind
IBM Research

Abstract

Modern garbage collectors partition the set of heap objects to achieve the best performance. For example, generational garbage collectors partition objects by age and focus their efforts on the youngest objects. Partitioning by age works well for many programs because younger objects usually have short lifetimes and thus garbage collection of young objects is often able to free up many objects. However, generational garbage collectors are typically much less efficient for longer-lived objects, and thus prior work has proposed many enhancements to generational collection.

Our work explores whether the connectivity of objects can yield useful partitions or improve existing partitioning schemes. We look at both direct (e.g., object A points to object B) and transitive (e.g., object A is reachable from object B) connectivity. Our results indicate that connectivity correlates strongly with object lifetimes and deathtimes and is therefore likely to be useful for partitioning objects.

*This work is supported by NSF ITR grant CCR-0085792, an IBM Faculty Partnership Award, and an equipment grant from Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

1 Introduction

Modern garbage collectors partition the set of heap objects to achieve the best performance. Ideally, a partition has three properties: (i) objects that are often accessed at the same time are grouped together in the same partition (spatial locality) improving memory system performance; (ii) objects in a partition can be garbage collected independently of other partitions, which helps to avoid the overhead and long pause times of full collections; and (iii) objects in a partition die at the same time, which enables a garbage collector to collect many objects at the same time. Although age-based partitioning [39, 34], used in generational garbage collectors, provides the above properties for the youngest objects, these properties do not necessarily hold for the older objects [21]. To alleviate some of the shortcomings of age-based partitioning, researchers have proposed many enhancements [4, 10, 11, 24, 43].

Our work explores whether the *connectivity* of objects can yield new partitioning schemes or improve existing schemes. We investigate both *direct* connectivity (e.g., object O_1 points to object O_2) and *transitive* connectivity (e.g., object O_1 is reachable from object O_2). Using empirical evidence, this paper studies the potential usefulness of partitioning by connectivity; a collector that exploits partitioning for connectivity is a subject for future work.

We conducted our research in the Jikes Research Virtual Machine (RVM)[1] [1] (compiler [6] and runtime system) to collect lifetime and connectivity data for Java programs. Our benchmarks include the SPECjvm98 suite, the Java-Olden suite, and a number of other applications including a web server. Broadly speaking, we investigated three kinds of connectivity: (i) connectivity from the stack (ii) connec-

[1]The Jikes RVM is an open source research virtual machine for Java that was formerly called Jalapeño. It is available at `www.ibm.com/developerworks/oss/jikesrvm`.

tivity from globals[2], and (iii) connectivity in the heap.

Our results indicate that all three kinds of connectivity correlate strongly with object lifetimes and deathtimes: (i) objects that are reachable only from the stack are usually shortlived [4]; (ii) objects that are reachable from globals usually live for most of the program execution; (iii) heap objects with a pointer (or path of pointers) between them usually have the same deathtime. We also discuss an algorithm for performing partial garbage collections based on connectivity information. We defer implementation and evaluation of the collector to future work.

The remainder of the paper is organized as follows: Section 2 introduces terms that we use throughout the paper. Section 3 explains how we conduct our measurements, and Section 4 describes our benchmark programs. Section 5 presents our results. Section 6 distinguishes our contributions from prior work and Section 7 concludes.

2 Background

From the point of view of a memory manager, a running program creates new objects and modifies pointers in existing objects. A garbage collector traverses the heap starting from the *roots* (stack and global pointers) to determine which objects are unreachable and collects them. The heap can be viewed as a directed graph, where each object is a node and each pointer is a directed edge. In these terms, the running program can be viewed as a *mutator* that adds and removes edges and creates new nodes. The garbage collector deletes nodes unreachable from the roots along with their outgoing edges.

A *global object graph* (GOG) is a graph that has a node for each object created in a program execution and a directed edge for each association between two objects via a pointer in a program execution. The GOG is the union of the object graphs at all points in a program execution. Thus, if at one time an object O_1 points to object O_2, and at another time O_1 points to object O_3, then the GOG has edges from O_1 to both O_2 and O_3. Associated with each node of the GOG is a *birthtime*, the time at which the object was created, and a *deathtime*, the time at which the object became unreachable. The *lifetime* of an object is the difference of its deathtime and birthtime. We follow other garbage collection researchers (e.g., [4]) in measuring time in bytes allocated since the start of the program.

A strongly-connected component (SCC) in the GOG is a maximal set of objects such that each object in the SCC is reachable from all other objects in the SCC [13]. A weakly-connected component (WCC) in the GOG is a maximal set of objects such that if one ignores the directions of edges, each object in the WCC is reachable from all other objects in the WCC. For example, a doubly-linked list is an SCC or part of a larger SCC, and a tree is a WCC or part of a larger WCC or SCC.

Blackburn *et al.* [4] classify objects by their lifetime and deathtime into shortlived, longlived, and immortal objects. We extend this definition by further classifying immortal objects into quasi and truly immortal as follows:

- An object that dies at the termination of the program is *truly-immortal*.
- Else, if the time from the object's deathtime to the termination of the program is shorter than the object's lifetime, then the object is *quasi-immortal*.
- Else, if the object's lifetime is shorter than the threshold $T_a \times$ *high_watermark*, the object is *short-lived* (we use $T_a = 0.2$). The *high_watermark* is the maximum number of bytes in reachable heap objects during the program execution.
- Else, the object is *long-lived*.

The motivation for the definition of shortlived as a fraction of *high_watermark* is that generational garbage collectors often reserve a fixed fraction of the heap for the nursery. In our measurements, we found the shortlived/longlived distinction to be largely independent from the precise definition of shortlived. Ideally, a tracing garbage collector should not expend any effort on quasi immortal or truly immortal objects, but focus mostly on shortlived and occasionally on longlived objects.

Some of our benchmarks, such as SPECjvm98, are invoked by harness code that is also written in Java and executed by the VM. We consider objects that survive until the termination of the benchmark proper as truly immortal even if they do not survive until the termination of the harness.

3 Methodology

Unless stated otherwise, we conducted our experiments using a "BaseBasenoncopyingGC" image (called "BaseBaseMarkSweep" in version 2.0.0) of version 1.1 of the Jikes RVM on Linux/PowerPC.

[2] We consider static fields of Java classes as global variables.

This image uses the baseline (non-optimizing) compiler to compile both VM and application methods and uses a mark-and-sweep garbage collector. Using a mark-and-sweep collector ensures that objects do not move, and thus the memory address of an object is a reliable way of identifying it during its lifetime.

To gather the runtime program characteristics required to build the GOG, we modified the Jikes RVM [1] to trace the following events:

- The *object allocation event* identifies the newly allocated object and its allocation context (thread and activation record).
- The *pointer assignment event* identifies the object pointed to and the location where the pointer is stored.
- The *deallocation event* identifies a dead object. The garbage collector generates these events.

An object allocation event creates a node in the GOG and annotates it with birthtime, type, and allocation context. An object deallocation event updates the deathtime and lifetime of the corresponding object. Since the garbage collector generates the deallocation events, the timings for these events are not precise: an object reported as dead at garbage collection n may have become unreachable any time after collection $n-1$. To reduce this imprecision, we perform relatively frequent garbage collections, and we use precise deathtime traces [22] for the set of numbers most sensitive to this issue (Section 5.3.4).

There are two kinds of pointer assignment events. A pointer assignment event where the pointer is stored into a field of a heap object creates an edge between two objects if the edge does not already exist. A pointer assignment event where the pointer is stored into a global or a stack variable updates information in the pointed-to object that we use to determine whether it escapes.

4 Benchmark Programs

Table 1 describes our benchmark suite. Our benchmark suite includes the SPECjvm98 [32] and JavaOlden [7] suites. Prior work on garbage collection has also used these programs. In addition, we used four real-world applications as benchmarks: an XML database, a web-server, a chat-server, and an XSLT processor.[3] The micro-benchmark *null* consists of an empty main method. The ***average*** row in each table gives the arithmetic mean of the results for all benchmarks, except *null*.

The second column of Table 1 shows the total size of all class files for each benchmark in KB. The JavaOlden benchmarks are small, they are kernels illustrating a single algorithmic task. The third and fourth column of Table 1 give descriptions of the benchmarks and their inputs. The Column "GC interval" shows the number of bytes allocated between forced garbage collections. We chose GC intervals of approximately 1/25 of the high watermark in Bytes.

The Jikes RVM is written in the Java programming language and is self-hosted; it provides its own runtime services, allocating objects in the same heap as user data. Thus, the effectiveness of the garbage collector impacts not only the application, but also the virtual machine. For most of our measurements, we present two sets of numbers: one for all allocated objects (including VM and library objects), and one for just the objects allocated on behalf of the application. The numbers for all allocated objects are relevant because VM boot-up and compilation are part of the program's execution. The application-only numbers illustrate the differences between benchmarks.

Table 2 shows some statistics for our benchmarks. Of the SPECjvm98 benchmarks, *compress* and *mpegaudio* do not exercise the garbage collector much.

The "Owner" columns in Table 2 categorize heap objects into the percentage of total objects that are allocated (i) in the Jikes RVM boot image, (ii) by the running VM, and (iii) by the application. The Jikes RVM boot image contains a memory snapshot of compiled core VM classes and their associated objects. When the VM boots, this file is read and the corresponding objects are recreated [2]. Once the Jikes RVM has booted, we use the allocation site to classify objects into RVM or application objects. If the allocation site is within the standard Java library, we traverse the dynamic chain until we encounter a caller in the RVM runtime system or the application. For the larger benchmarks, most of the objects are allocated by the application itself; on the other end of the spectrum, the synthetic benchmark *null* intentionally does not allocate any application objects.

The "Lifetime" columns in Table 2 categorize heap objects by their lifetime following the definitions in Section 2. The numbers in parentheses count only objects where the owner is the application (we use this convention throughout this paper). The "n/a" for the *null* benchmark indicates that the application does no allocation.

We see that most benchmarks have a high percentage of shortlived objects confirming the weak gener-

[3] The benchmarks are available at `http://systems.cs.colorado.edu/colorado_bench/`.

Table 1: Benchmark programs.

Benchmark	Size/KB	Description	Input	GC interval
null	0.2	Empty main method, does nothing.	–	553,905
SPECjvm98				
compress	17.4	Modified Lempel-Ziv method (LZW).	-s 100 -m1 -M1	1,193,412
db	9.9	Performs database functions on memory resident database.	-s 100 -m1 -M1	951,881
jack	127.8	Parser generator, earlier version of JavaCC.	-s 100 -m1 -M1	678,956
javac	1909.2	The Java compiler from the JDK 1.0.2.	-s 100 -m1 -M1	1,080,090
jess	387.2	Java Expert Shell System.	-s 100 -m1 -M1	684,702
mpegaudio	117.3	Decompresses audio files.	-s 100 -m1 -M1	789,493
mtrt	56.5	Multi-threaded raytracer.	-s 100 -m1 -M1	895,044
Java-Olden				
bh	17.3	Solves the N-body problem using hierarchical methods.	-b 500 -s 10	591,819
bisort	4.6	Sorts by creating and merging bitonic sequences.	-s 100000	588,607
em3d	7.1	Simulates electromagnetic waves propagation in 3D object.	-n 2000 -d 100	805,814
health	9.8	Simulates Columbian health care system.	-l 5 -t 500 -s 0	559,042
mst	5.8	Computes minimum spanning tree of a graph.	-v 50	558,853
perimeter	9.8	Computes perimeter of quad-tree encoded raster images.	-l 16	1,118,846
power	11.2	Solves the power system optimization problem.	–	608,275
treeadd	3.1	Adds the values in a tree.	-l 20	1,375,696
tsp	5.9	Computes estimate for traveling salesman problem.	-c 60000	669,420
voronoi	13.9	Computes Voronoi diagram of a set of points.	-n 2048	590,537
Colorado Benchmarks				
ipsixql	1986.2	Performs queries against persistent XML document.	XQL queries against Shakespeare	689,387
jigsaw	4312.9	W3C's web-server, reference implementation of HTML 1.1.	download of complete contents	1,000,000
nfc	556.0	Chat-server.	10 rooms, 100 users, 100,000 messages	1,000,000
xalan	4200.0	XSLT tree transformation language processor.	summarize a Shakespeare play	905,853
Average				
average	655.6	No benchmark, arithmetic mean of all benchmarks but *null*.	–	825,510

Table 2: Benchmark statistics. The owner and lifetime numbers are in percent of allocated objects. The numbers in parentheses count only objects where the owner is the application.

Benchmark	High watermark	Total allocation		Owner (%)			Lifetime (%)			
	(bytes)	(bytes)	(objects)	Boot	RVM	App.	Shortlived	Longlived	Quasi imm.	Truly imm.
null	14,109,770	14,396,503	106,009	76.4	23.6	0.0	2.4 (n/a)	0.0 (n/a)	9.2 (n/a)	88.4 (n/a)
compress	27,688,408	132,931,724	226,002	35.8	62.8	1.3	42.9 (37.2)	9.6 (17.1)	0.0 (2.3)	47.4 (43.4)
db	23,503,105	97,899,266	3,401,539	2.4	3.2	94.4	87.6 (90.1)	0.1 (0.1)	0.1 (0.1)	12.2 (9.8)
jack	16,907,838	331,031,287	8,194,044	1.0	12.7	86.3	96.6 (97.6)	1.8 (2.1)	0.1 (0.1)	1.6 (0.2)
javac	25,296,557	285,631,761	8,228,933	1.0	23.3	75.7	81.0 (77.0)	13.5 (17.8)	1.1 (1.5)	4.4 (3.6)
jess	17,056,718	334,187,450	8,662,674	0.9	7.6	91.4	98.0 (99.3)	0.3 (0.3)	0.0 (0.0)	1.7 (0.4)
mpegaudio	16,578,151	35,850,575	380,054	21.3	77.7	1.0	68.6 (7.6)	0.0 (0.0)	0.0 (0.0)	31.3 (92.4)
mtrt	22,414,466	173,683,581	6,889,168	1.2	2.9	95.9	93.7 (95.1)	0.0 (0.0)	2.2 (2.3)	4.1 (2.6)
bh	14,580,046	42,900,870	1,212,329	6.7	5.1	88.2	88.4 (95.5)	1.2 (1.3)	0.4 (0.4)	10.1 (2.7)
bisort	14,628,360	16,085,265	176,878	45.8	17.2	37.1	11.5 (0.0)	0.0 (0.0)	0.0 (0.0)	88.4 (100.0)
em3d	19,534,225	22,101,972	135,894	59.6	28.6	11.8	21.1 (0.0)	0.0 (0.0)	0.0 (0.0)	78.9 (100.0)
health	16,563,273	38,618,097	1,332,116	6.1	4.0	89.9	82.2 (88.1)	0.5 (0.5)	0.5 (0.5)	16.9 (10.8)
mst	14,254,193	15,446,269	124,317	65.2	30.7	4.1	13.5 (0.0)	0.0 (0.0)	6.0 (0.0)	80.4 (100.0)
perimeter	27,458,366	31,528,263	595,507	13.6	10.3	76.1	8.5 (0.0)	0.0 (0.0)	0.0 (0.0)	91.5 (100.0)
power	14,914,494	38,101,825	912,770	8.9	5.3	85.8	85.1 (94.4)	0.0 (0.0)	0.0 (0.0)	14.9 (5.6)
treeadd	33,644,773	35,751,748	1,159,451	7.0	2.6	90.4	1.7 (0.0)	0.0 (0.0)	0.0 (0.0)	98.3 (100.0)
tsp	16,836,300	21,583,991	310,956	26.0	10.7	63.2	45.5 (60.7)	0.0 (0.0)	0.0 (0.0)	54.5 (39.3)
voronoi	14,832,375	17,712,879	191,434	42.3	26.1	31.6	27.0 (22.2)	0.0 (0.0)	0.1 (0.2)	72.9 (77.7)
ipsixql	17,141,410	99,908,400	2,357,562	3.4	16.6	80.0	84.0 (85.7)	6.7 (8.4)	1.9 (2.4)	7.3 (3.4)
jigsaw	26,487,443	257,452,354	4,289,782	1.9	68.0	30.2	93.4 (92.2)	0.0 (0.0)	0.0 (0.0)	6.5 (7.8)
nfc	25,643,076	173,637,549	2,154,719	3.8	21.7	74.6	93.3 (99.4)	0.0 (0.1)	0.0 (0.0)	6.7 (0.5)
xalan	22,784,083	123,412,189	1,637,966	4.9	92.5	2.5	87.5 (87.2)	0.1 (0.9)	0.1 (1.4)	12.2 (10.5)
average	20,416,555	110,736,063	2,503,528	17.1	25.2	57.7	62.4 (58.5)	1.6 (2.3)	0.6 (0.5)	35.3 (38.6)

ational hypothesis [21]. Few objects are longlived or quasi immortal but many benchmarks have a significant fraction of truly immortal objects. The number of truly immortal objects is particularly high for the micro-benchmark *null*. This is because *null* allocates only system objects, and many of these survive until the VM terminates. It may therefore be worthwhile to treat system objects specially in a memory manager for a VM implemented in Java. A significant percentage of application-only objects are also truly immortal. This is contrary to the strong generational hypothesis and motivates techniques like pretenuring [20, 4]. For those benchmarks where almost all objects are truly immortal, never attempting to collect garbage may be the best approach to memory management [16].

5 Results

This section presents our results. Section 5.1 investigates connectivity from the stack, Section 5.2 investigates connectivity from globals, and Section 5.3 investigates connectivity within the heap.

Modern collectors achieve short pause times by performing partial collections. However, these partial collections usually require write barriers, which are expensive. In Section 5.4 we investigate an idea for how connectivity information can enable safe partial garbage collections without write barriers.

5.1 Correlation between lifetime and connectivity from stack

This sections considers two kinds of connectivity from the stack: objects that are reachable only from the stack and objects that escape their allocating activation records or threads.

5.1.1 Objects reachable only from the stack

Given sufficient compiler support, objects pointed to *only* by the stack should be relatively cheap to garbage collect because they are not pointed to by the heap or global variables. Figures 1(a) and (b) present data for all objects and only for objects allocated on behalf of the application, respectively. In both figures the length of a bar gives the percentage of objects that are reachable only from the stack. Each bar has four segments, for shortlived, longlived, quasi immortal, and truly immortal objects.

Our results indicate that the benchmarks have a significant percentage of objects that are pointed to *only* from the stack: in 11 of the 22 benchmarks it is higher than 30%. For most benchmark programs, the majority of these objects are shortlived. If a compiler can identify allocation sites whose objects do not escape into the heap or globals, these objects can be allocated in a special area where they can be garbage collected cheaply.

Table 3: Escape rates (in percent of allocated objects). The numbers in parentheses count only objects where the owner is the application.

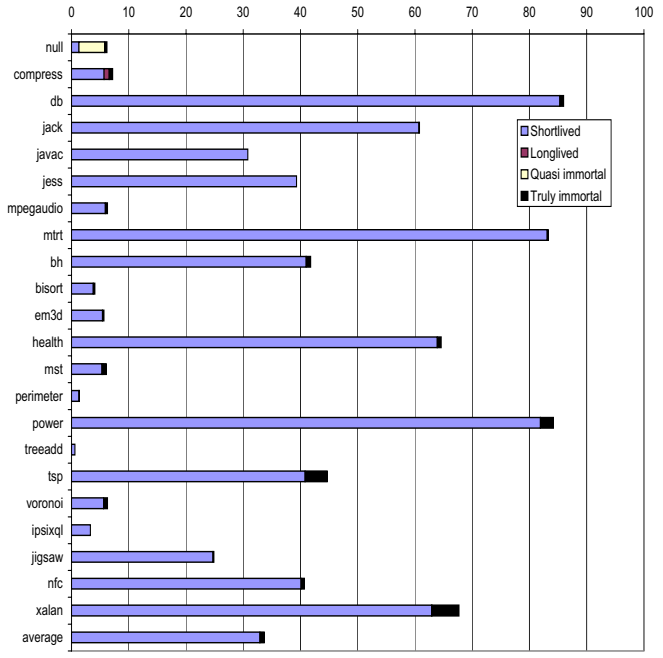
Benchmark	No escape	Stack frame	Thread
null	10.0 (n/a)	90.0 (n/a)	74.0 (n/a)
compress	10.5 (21.5)	89.5 (78.5)	39.4 (4.5)
db	0.8 (0.1)	99.2 (99.9)	2.6 (0.0)
jack	37.2 (39.2)	62.8 (60.8)	1.2 (0.0)
javac	29.7 (37.3)	70.3 (62.7)	1.4 (0.0)
jess	40.4 (43.3)	59.6 (56.7)	1.2 (0.0)
mpegaudio	9.4 (8.3)	90.6 (91.7)	24.8 (0.3)
mtrt	82.5 (85.6)	17.5 (14.4)	1.4 (0.0)
bh	41.1 (45.2)	58.9 (54.8)	7.0 (0.0)
bisort	6.4 (0.0)	93.6 (100.0)	46.3 (0.0)
em3d	14.8 (50.0)	85.2 (50.0)	60.6 (0.0)
health	13.8 (14.3)	86.2 (85.7)	6.2 (0.0)
mst	9.4 (0.0)	90.6 (100.0)	66.1 (0.0)
perimeter	2.2 (0.0)	97.8 (100.0)	13.9 (0.0)
power	84.7 (97.0)	15.3 (3.0)	9.1 (0.0)
treeadd	1.0 (0.0)	99.0 (100.0)	7.1 (0.0)
tsp	46.2 (66.7)	53.8 (33.3)	26.8 (0.0)
voronoi	7.4 (0.1)	92.6 (99.9)	44.0 (0.0)
ipsixql	4.0 (2.7)	96.0 (97.3)	4.2 (0.0)
jigsaw	26.5 (36.1)	73.5 (63.9)	6.2 (8.9)
nfc	28.2 (26.4)	71.8 (73.6)	21.3 (21.7)
xalan	66.6 (18.8)	33.4 (81.2)	7.4 (7.5)
average	26.8 (28.2)	73.2 (71.8)	19.0 (2.0)

5.1.2 Lifetime of escaping objects

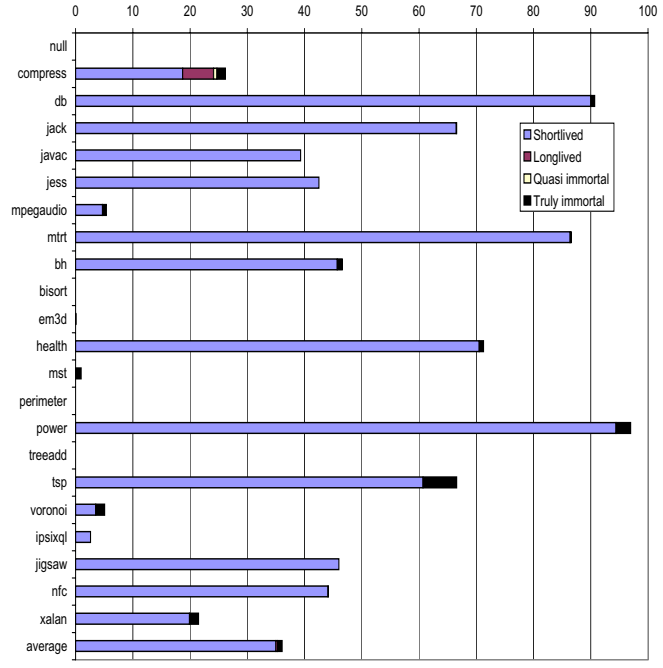
Recently, there has been much work on escape analysis [12, 18, 36]. Prior work has used escape analysis to eliminate synchronization or to allocate objects on the stack [17, 38]. We investigate whether object escapement has any correlation to object lifetime.

Table 3 gives the percentage of objects that escape the stack frame or thread that created them (the numbers in parentheses consider only application objects). An object escapes its stack frame if it (i) is returned from its allocating method, (ii) is assigned to a global, or (iii) is assigned to a field of some other object that is reachable from a caller activation record. An object escapes from a thread if it becomes reachable from a thread other than the one that created it. The Jikes RVM runtime system creates threads for garbage collection and finalization and thus even single-threaded benchmarks may have thread-escaping application objects.

We see that on average, only 26.8% of all objects are non-escaping, the rest escape at least their stack frame, and 19% even escape their thread.



(a) In percent of all allocated objects.



(b) In percent of objects allocated by application.

Figure 1: Lifetime of objects pointed to only by the stack. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.

Figure 2 shows the lifetime of objects that escape their thread. In Figure 2, the length of the bars shows the percentage of objects that escape their thread. Each bar is subdivided into four segments, one for each lifetime bin. Figure 2(a) shows data for all objects, whereas Figure 2(b) only shows data for the application objects.

From Figure 2 we see that while escaping objects are often truly immortal, it is not always true. In particular, *nfc* has many objects that escape a thread, but are shortlived. Since *nfc* is one of our most realistic benchmarks, we conclude that a garbage collector cannot ignore thread-escaping objects; indeed many of them may be shortlived. The intuition for this is that server applications often use shortlived thread-escaping objects to communicate between threads. We found the correlation between lifetime and escaping from the stack to be weaker than the correlation between lifetime and escaping from the thread.

We repeated the above experiments for our three multi-threaded benchmarks, *mtrt*, *jigsaw*, and *nfc*, and this time ignored objects that escaped to the garbage collector thread. While the data changed slightly, the main conclusions remained the same. For example, most of the objects that escaped a thread in *nfc* remained shortlived.

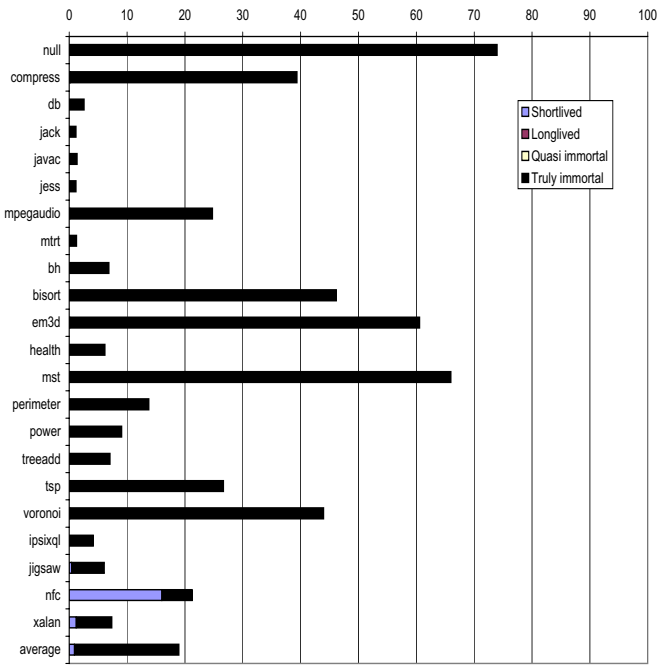
5.2 Correlation between lifetime and connectivity from globals

Figure 3 shows the lifetime of objects reachable from globals. It includes objects that may also be reachable from the stack or heap. In Figure 3, the length of the bars shows the percentage of objects reachable from globals. Each bar is subdivided into four segments, one for each lifetime bin. Figure 3(a) shows data for all objects, whereas Figure 3(b) only shows data for the application objects.

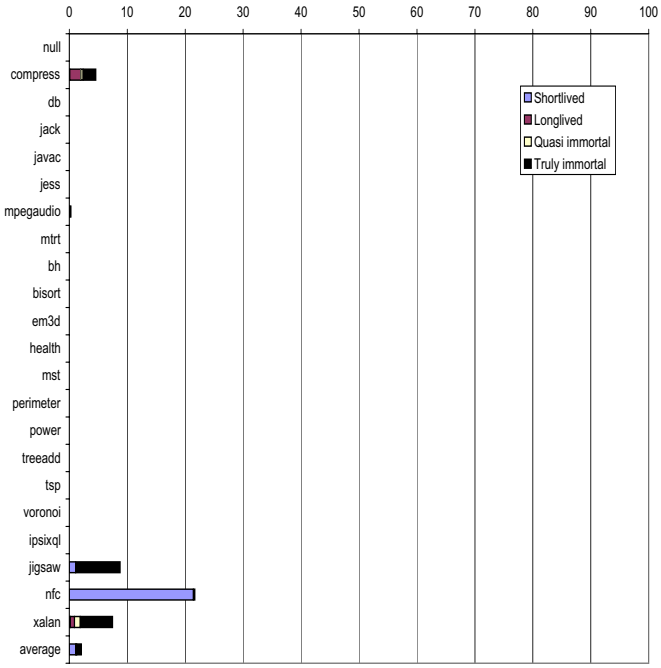
Because global variables exist as long as their classes exist, we expect objects reachable from globals to be immortal.[4] Figure 3 confirms our expectations. From Figure 3(a) we see that with the exception of *jack*, *nfc*, and *ipsixql*, most objects reachable from globals are truly immortal. Of these benchmarks, *jack* has a relatively small percentage of objects reachable from globals.

The benchmark *ipsixql* has a large Scc that is reachable from globals and is heavily mutated. Figure 4 demonstrates this pictorially. The horizontal

[4]With an interprocedural liveness analysis for global variables, a garbage collector may be able to collect objects even if they are still reachable from globals [23]. We disregard this possibility, because we believe interprocedural liveness analysis for globals is unrealistic for Java programs.

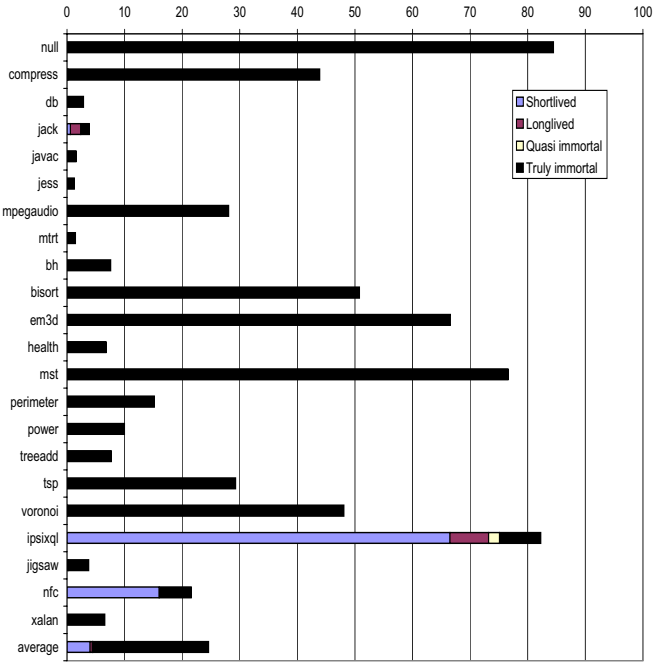


(a) In percent of all allocated objects.

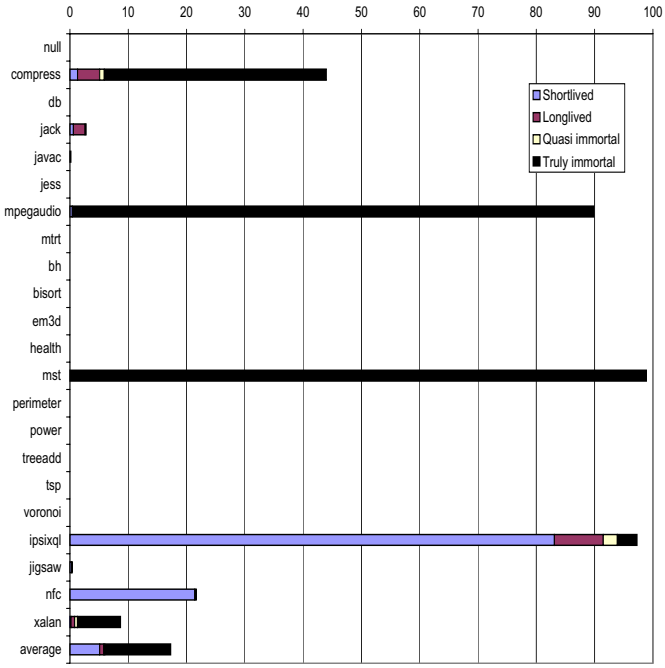


(b) In percent of objects allocated by application.

Figure 2: Lifetime of objects escaping their thread. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.



(a) In percent of all allocated objects.



(b) In percent of objects allocated by application.

Figure 3: Lifetime of objects reachable from globals. For most benchmarks the longlived and quasi immortal segments of the bars are nearly empty.

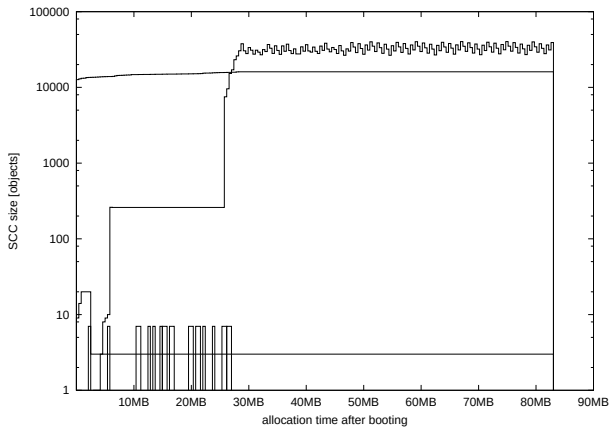


Figure 4: SCCs in *ipsixql*.

axis gives time in bytes allocated. The vertical axis (in log scale) gives the size of an SCC in number of objects. There is one curve for each SCC with at least two objects. A point (x, y) on a curve, C, means that at time x, SCC C has size y objects. Figure 4 shows that *ipsixql* has one very large SCC and several smaller ones. Although many of the objects in the largest SCC die together at the end of the program, significant parts of the SCC are continuously replaced with newly allocated objects. This large SCC is part of a cache that stores faulted objects.[5] The atypical behavior of this large SCC dominates the behavior of *ipsixql*, and among other things destroys the correlation between lifetime and reachability from globals. In addition, we will see in Section 5.4 that *ipsixql* incurs a significant write barrier overhead.

In summary, we see that for all benchmarks (except for *jack*, *nfc*, and *ipsixql*) there is a strong correlation between reachability from globals and lifetime. A generational garbage collector could exploit these observations by eagerly promoting objects reachable from globals to old generations.

5.3 Correlation between lifetime and connectivity from heap

In this section we consider several kinds of heap connectivity. We investigate how likely it is for objects that are connected by a pointer to have the same deathtime (Section 5.3.1) and whether the popularity of objects is related to their lifetimes (Section 5.3.2). Next, we investigate how likely it is for transitively connected objects to have the same deathtime (Section 5.3.3), and we conclude by evaluating how sensitive these same-deathtime results are to our methodology of tracing with frequent garbage collections (Section 5.3.4).

5.3.1 Linked objects

This section explores the deathtime of directly-linked objects. First, we look at how often objects are modified. Consider a program that repeatedly modifies an object O such that a field in O points to one of many different objects at different times. In this case, we can expect O's deathtime to be largely unrelated to the deathtime of objects that O points to. If, on the other hand, the program modifies few objects after initialization, then we can expect a significant correlation between the deathtime of connected objects.

We view the first non-null assignment to an object field as "initialization", and we view subsequent non-null assignments to the same field as "mutations". Column "Mutated" of Table 4 gives the percentage of all allocated heap objects that are mutated during program execution. The numbers in parentheses are the percentage of objects allocated by the application that are mutated during program execution. Table 4 shows that programs do not mutate the majority of objects, and thus, the lifetimes of linked objects are likely to be related.

Column "Write barrier" of Table 4 gives the overhead of the write barrier for our benchmark program runs. We omit timing data for *jigsaw* and *nfc* because they are interactive. Write barrier overheads are measured using a Jikes RVM (v2.0.2) FastSemispace image on a 1 processor PPC/AIX machine. The write barrier is implemented as a sequential store buffer. Because the Jikes RVM is written in the Java programming language, the overheads include the execution of the application, VM, and optimizing compiler at its default optimization level (1).

Table 5 investigates the correlation between direct object connectivity and object deathtimes. Column "$O_1 \rightarrow O_2$" of Table 5 gives the probability that two adjacent objects in the GOG have the same deathtime. We see that for many programs the probability is nearly 100%. In contrast, column "Any pair" gives the probability that any two possibly unlinked objects in the program die at the same time. We compute this value by considering all pairs, both linked and unlinked. We see that in most cases, the probability that linked objects die at the same time is much higher than the probability of any two objects dying at the same time.

Column "$O_1 \rightarrow O_2$, O_1 mutated" in Table 5 gives the probability that two objects, O_1 and O_2, have

[5] Because sources for *ipsixql* are not available our observations are based on the output of a decompiler, and thus are somewhat speculative.

Table 4: Mutation rates in % of allocated objects and write barrier overheads in % of total execution time. In the mutation rates, the numbers in parentheses consider only objects where the owner is the application.

Benchmark	Mutated	Write barrier
null	18.6 (n/a)	15.8
compress	10.5 (8.0)	3.9
db	0.7 (0.0)	1.5
jack	4.0 (4.2)	5.7
javac	18.2 (23.4)	19.4
jess	3.5 (3.3)	8.1
mpegaudio	7.5 (3.8)	3.4
mtrt	1.2 (0.9)	4.3
bh	5.7 (4.5)	1.2
bisort	29.8 (50.0)	6.6
em3d	14.8 (0.0)	2.0
health	16.4 (16.6)	1.0
mst	16.2 (1.3)	2.6
perimeter	3.5 (0.0)	1.2
power	2.2 (0.0)	0.1
treeadd	1.7 (0.0)	6.0
tsp	27.5 (33.3)	10.9
voronoi	33.8 (73.2)	5.2
ipsixql	1.7 (0.4)	19.8
jigsaw	4.1 (6.4)	–
nfc	14.6 (17.6)	–
xalan	2.0 (2.5)	32.4
average	10.5 (11.9)	7.6

Table 5: Pairs of objects with same deathtime (in percent of pairs of objects with given connectivity). The numbers in parentheses count only objects where the owner is the application.

Benchmark	Any pair	$O_1 \rightarrow O_2$	$O_1 \rightarrow O_2$, O_1 mutated
null	79.5 (n/a)	96.5 (n/a)	97.7 (n/a)
compress	22.8 (19.5)	95.5 (63.9)	96.8 (44.2)
db	2.0 (2.1)	22.7 (19.7)	12.5 (0.6)
jack	0.3 (0.3)	54.1 (45.4)	19.1 (5.2)
javac	0.7 (0.9)	66.1 (65.8)	70.4 (68.7)
jess	0.2 (0.3)	63.6 (58.6)	90.1 (89.0)
mpegaudio	10.0 (83.6)	94.8 (74.8)	94.6 (43.8)
mtrt	0.7 (0.7)	77.3 (71.2)	75.9 (57.6)
bh	2.6 (2.3)	89.3 (87.0)	71.0 (51.1)
bisort	78.9 (100.0)	98.9 (100.0)	99.6 (100.0)
em3d	63.7 (100.0)	99.1 (100.0)	97.7 (100.0)
health	4.4 (3.1)	11.4 (9.5)	4.7 (3.6)
mst	66.2 (100.0)	96.1 (100.0)	96.6 (100.0)
perimeter	84.0 (100.0)	99.3 (100.0)	96.0 (n/a)
power	3.8 (2.8)	96.3 (100.0)	97.8 (n/a)
treeadd	96.6 (100.0)	99.4 (100.0)	97.7 (n/a)
tsp	27.9 (15.7)	98.2 (100.0)	99.4 (100.0)
voronoi	44.7 (38.0)	89.1 (82.7)	85.3 (78.9)
ipsixql	1.3 (1.3)	79.1 (79.3)	78.8 (25.2)
jigsaw	0.8 (1.6)	88.9 (83.6)	92.1 (85.9)
nfc	1.0 (0.8)	75.7 (68.9)	69.5 (64.0)
xalan	2.2 (21.9)	94.3 (93.2)	94.2 (82.5)
average	24.5 (33.1)	80.4 (76.4)	78.1 (61.1)

the same deathtime given that O_1 points to O_2 and O_1 is mutated. For 14 of the 22 benchmarks (19 of 22 benchmarks when looking at the application-only numbers), these probabilities are lower than the ones in column "$O_1 \rightarrow O_2$".

Tables 4 and 5 show that for many benchmarks there is both a high probability that objects are not mutated and that objects linked by a pointer have the same deathtime. However, for some programs, such as *db*, we see that even though it has a low mutation rate (0.7%), it also has a relatively low probability of linked objects dying at the same time (22.7%). In other words, a low percentage of modified objects is no guarantee for a high correlation of deathtimes of connected objects. Apparently, even though *db* modifies only few objects, the modifications happen in key places and thus have a big impact on the deathtimes of linked objects.

A garbage collector can exploit these results by clustering linked objects together. Since on average linked objects have a 80.4% probability of dying at the same time, the garbage collector will be able to free up many objects at once.

5.3.2 Incoming pointers

This section investigates whether there is a correlation between the popularity of an object and its life-

time. A *popular object* is one that is pointed to by many other objects [24].

We counted the number of objects pointed to by at least two other heap objects. For most benchmarks, fewer than 40% of the objects had at least two predecessors. We also looked at the lifetime distribution of objects pointed to by at least two other heap objects. The distribution varied widely: in some cases, most of these objects were shortlived, while in other cases most of these objects were truly immortal. Because of space constraints we do not present the results in detail.

To conclude, we saw little correlation between the popularity of an object and its lifetime.

5.3.3 Sccs and lifetime

Table 5 suggests that direct connectivity is usually, but not always, a good indicator of deathtime. In this section we consider more global notions of connectivity: strongly and weakly connected components (SCCs and WCCs, Section 2).

Column "In nontriv. SCC" in Table 6 shows the percentage of objects that belong to SCCs with at least two objects in the global object graph. On average, only a minority of objects are members of nontrivial SCCs.

Column "Same SCC" in Table 6 gives the probability that two objects in the same SCC have the same deathtime. Column "Same WCC" in Table 6 gives the probability that two objects in the same WCC have the same deathtime. The numbers in parentheses consider only objects allocated by the application. Since an SCC implies stronger connectivity, we expected that the probability would be higher for an SCC than for a WCC.

Table 6 shows that for many programs there is a high probability that objects in the same SCC die together. For many benchmarks the probability for two objects in an SCC having the same deathtime is greater than the probability of two linked objects (Table 5) having the same deathtime. A garbage collector could exploit these observations by designating any object in an SCC as the *key object* [21], the object whose death likely coincides with the death of other objects connected to it. Thus, when that object dies, there is a good chance that the rest of the SCC is also garbage.

5.3.4 Trace granularity

For most of the numbers in this paper we analyzed traces with three kinds of events: object allocation events, pointer assignment events, and deallocation events (see Section 3). To obtain the deallocation events, we performed frequent garbage collections. In our traces, all objects that become unreachable between collection n and collection $n+1$ die at the time when collection $n+1$ happens. Thus, our traces are *granulated*: deathtimes are not precise, but rounded up to a multiple of the GC interval (rightmost column in Table 1).

Table 6: Column "In nontriv. SCC" shows objects in non-trivial SCCs (in percent of allocated objects). Columns "Same SCC" and "Same WCC" show pairs of objects with the same deathtime (in percent of pairs of objects in the same SCC or WCC, respectively). The numbers in parentheses count only objects where the owner is the application.

Benchmark	In nontriv. SCC		Same SCC		Same WCC	
null	13.0	(n/a)	99.8	(n/a)	95.4	(n/a)
compress	9.8	(13.5)	99.4	(100.0)	25.7	(64.6)
db	0.6	(0.0)	99.5	(100.0)	2.2	(2.1)
jack	0.4	(0.0)	99.0	(100.0)	0.5	(0.3)
javac	15.1	(19.0)	34.1	(34.0)	1.3	(2.6)
jess	0.7	(0.0)	94.9	(19.3)	0.2	(0.3)
mpegaudio	8.7	(8.2)	99.1	(100.0)	10.4	(99.1)
mtrt	0.6	(0.2)	99.5	(100.0)	21.8	(23.1)
bh	1.4	(0.0)	99.6	(100.0)	28.5	(5.4)
bisort	8.0	(0.0)	99.8	(n/a)	87.7	(100.0)
em3d	19.7	(74.9)	99.8	(100.0)	71.5	(100.0)
health	14.4	(14.6)	46.7	(46.3)	6.2	(4.7)
mst	13.9	(50.9)	99.8	(100.0)	76.8	(100.0)
perimeter	78.8	(100.0)	100.0	(100.0)	96.6	(100.0)
power	1.7	(0.0)	99.7	(n/a)	64.1	(100.0)
treeadd	1.2	(0.0)	99.8	(n/a)	99.9	(100.0)
tsp	25.7	(33.3)	100.0	(100.0)	87.3	(100.0)
voronoi	37.0	(91.5)	42.6	(39.2)	56.6	(38.0)
ipsixql	46.9	(56.5)	1.3	(1.3)	1.3	(1.3)
jigsaw	5.0	(3.1)	81.6	(63.8)	1.2	(3.2)
nfc	9.6	(10.8)	1.9	(0.8)	1.5	(0.8)
xalan	2.6	(2.0)	99.0	(98.6)	9.5	(26.0)
average	14.4	(22.8)	80.8	(72.4)	35.8	(46.3)

Until recently, the only known way to get *precise deathtime traces* (not granulated traces) was to perform a garbage collection at every allocation (e.g. [34]), which is prohibitively expensive. Recently, Hertz *et al.* proposed the Merlin algorithm [22] that generates precise deathtime traces much faster than the brute force method. When we used Hertz's precise deathtime traces to regenerate our results we found that it made a significant difference in the same deathtime numbers (Tables 5 and 6) but not in the classification of objects by lifetime into shortlived, longlived, quasi immortal, and truly immortal.

Table 7 shows how using granulated traces inflated the numbers in Tables 5 and 6. To obtain the numbers in Table 7, we recomputed the numbers in Tables 5 and 6 using precise deathtime traces. Then, we subtracted the numbers based on precise deathtime traces from the numbers based on granulated traces.

Table 7: Over-estimation of numbers in Tables 5 and 6 due to granulated traces.

Benchmark	Any pair	$O_1 \rightarrow O_2$	$O_1 \rightarrow O_2$, O_1 mutated	Same Scc	Same Wcc
compress	7.7	7.2	3.6	0.0	1.9
db	1.3	16.0	−1.0	0.0	1.3
jack	0.3	7.9	3.6	0.0	−0.3
javac	0.4	23.2	27.7	0.7	0.5
jess	−0.3	43.3	14.0	18.3	−0.3
mpegaudio	6.0	4.5	1.8	0.0	0.6
bh	2.3	1.4	6.5	0.0	5.1
bisort	0.0	0.0	0.0	n/a	0.0
em3d	0.2	2.4	100.0	0.0	−0.1
health	2.2	5.4	3.0	0.5	1.9
mst	2.1	2.2	5.8	0.0	2.0
perimeter	0.0	33.3	n/a	0.0	0.0
power	2.7	0.0	n/a	n/a	0.0
treeadd	0.0	0.0	n/a	n/a	0.0
tsp	4.6	0.0	0.0	0.0	0.0
voronoi	8.8	12.3	13.4	8.5	8.8
average	2.3	9.9	13.7	2.1	1.3

We report these differences for application objects only. Since Merlin cannot yet trace multithreaded programs, Table 7 does not contain the results for all the benchmarks. Merlin's inability to handle multithreaded programs is also the reason why we do not use precise deathtime traces throughout the paper.

As expected Table 7 shows that granulated traces inflate the same deathtime numbers, i.e. most entries are greater than zero. (Since our precise and granulated traces use different runs and different versions of the Jikes RVM, there is some noise in our data leading to a few negative numbers.) The following table juxtaposes (a) the average number of pairs of application objects with the same deathtime based on granulated traces (last rows in Tables 5 and 6) and (b) the average over-estimation in these numbers (last row in Table 7).

	Any pair	$O_1 \rightarrow O_2$	$O_1 \rightarrow O_2$, O_1 mutated	Same Scc	Same Wcc
(a)	33.1	76.4	61.1	72.4	46.3
(b)	2.3	9.9	13.7	2.1	1.3

From this table we see that even though the likelihood of two linked objects having the same deathtime is lower by 9.9% (on average) with precise traces than with granulated traces, our basic results still hold. In other words, the likelihood of linked objects or objects in the same Scc having the same deathtime is much higher than the likelihood of two random objects having the same deathtime.

5.4 Partial collections with clustering by connectivity

A *partial garbage collection* processes only a part of the heap, as opposed to a *full garbage collection* that processes the entire heap. For example, generational garbage collectors frequently collect the younger objects (where most objects are likely to be dead) without collecting the older objects. Partial collections in a generational collector, however, use potentially expensive write barriers. Table 4 gives the overhead of write barriers in percent of total execution time. We see that write barriers are often expensive, accounting for 7.6% of program execution time on average. Prior work confirms these findings [37]. Fitzgerald and Tarditi [16] did experiments where generational collectors "... did poorly on benchmarks that had low collection costs and high write barrier costs. For those benchmarks, the cost of the write barrier was higher than the reduction in collection cost".

We hope to use connectivity information to avoid write barrier overhead even for partial collections. Our approach is based on Harris's algorithm in [19]. Harris's algorithm starts by incrementally building a type graph at class loading time in a Java system. The type graph has an edge from type T_1 to T_2 if T_1 has a field that can point to an object of type T_2. He then collapses all strongly connected components. The collapsed graph is a directed acyclic graph, and he calls each of the collapsed nodes a *partition*. For a partition P_1, we define $\mathit{ancestors}(P_1) = \{P_2 \mid P_2 \stackrel{*}{\rightarrow} P_1\}$ as the set of partitions from which P_1 is reachable in the partition DAG.

Harris uses these partitions for incremental collection. However, we observe that they can also be used for performing partial garbage collection without write barriers. For example, consider partitions without any incoming edges. Objects in these partitions (i.e. instances of classes in the partition) can be garbage collected without scanning any other partitions. To collect a partition *with* incoming edges, say P_1, the garbage collector needs to look only at the objects in $\mathit{ancestors}(P_1)$. This is similar to generational collection in that a partition with no incoming edges is analogous to the youngest generation and a partition at depth d in the DAG is analogous to generation d.

To use the above scheme for efficient partial collections, two properties must hold. First, the number of objects in the ancestors must be small for most partitions, otherwise we will end up having to collect a good part of the heap at every partial collection. Second, the objects that are close to the roots should be the most profitable to collect since they are the easiest to collect. We now present data for each of the requirements. Figure 5(a) gives the number of objects in the ancestors of the partition of each object in a benchmark program. To generate this graph we weighed the partition DAG with the live objects

in each partition at a particular snapshot in program execution. A point (x, y) in Figure 5(a) means that y objects are in partitions whose ancestor sets have sizes of at most x% of all live objects. We see that most objects require the garbage collector to look at about 59% of the total objects at that point. As Harris observes, these numbers are high because Java does not yet support generic types and thus container data structures have fields of type `Object` (and can therefore point to all objects). Stronger static analysis (or a language with generic types) may yield better results.

Figure 5(b) presents the same graph as Figure 5(a) except that it uses optimal partitioning: for each object, it reports how many other objects reach it in the snapshot used for Figure 5(a). Figure 5(b) thus gives an upper bound on the quality of partitioning with a stronger analysis than type-based analysis. Figure 5(b) shows that at least in the optimal scenario, all objects can be garbage collected by examining only about 18% of the objects.

Figures 5(a) and (b) present data measured on snapshot object graphs. This is equivalent to sampling the objects that happen to be alive at one particular point in time; such a sample will overemphasize longlived, quasi immortal, and truly immortal objects. Thus, it may be the case that for shortlived objects a garbage collector may need to look at many fewer objects than 59%.

Figure 6 shows the average number of objects from which each object can be reached in the *global* object graph. Figure 6(a) presents data for all objects and Figure 6(b) presents data for application objects only. The length of the bars is the average number of objects with a path to an object on a logarithmic scale. There are four bars for each benchmark, one per lifetime bin. We see that the bars for shortlived objects are usually the shortest (we have explained the exceptional behavior of *ipsixql* in Section 5.2). That is encouraging because it means that to garbage-collect shortlived objects, we do not have to look at too many other objects. This data also suggests that Figures 5(a) and (b) are overly pessimistic since they are based on snapshots which will be biased towards longer-lived objects.

6 Related Work

We now summarize relevant work on understanding object behavior, generational garbage collection, other relevant memory management schemes, and escape analysis.

6.1 Understanding object behavior

Barry Hayes described and tested the weak and strong generational hypotheses [21]. The weak generational hypothesis states that "newly-created objects have a much lower survival rate than older objects" [21]. The strong generational hypothesis states that "even if the objects in question are not newly created, the relatively younger objects have a lower survival rate than the relatively older objects" [21]. He found that even though the weak generational hypothesis is often true, the strong generational hypothesis is usually false. He goes on to describe key object opportunism, where the assumption is that connected objects die together and this can be exploited by collecting a data structure when its root dies. We provide supporting evidence for this claim and explore the correlation of different kinds of connectivity with lifetime.

Stefanović and Moss [35] explore the age distribution of objects. They collect their data by garbage collecting frequently. Unlike our work, Stefanović and Moss do not empirically relate age behavior to connectivity.

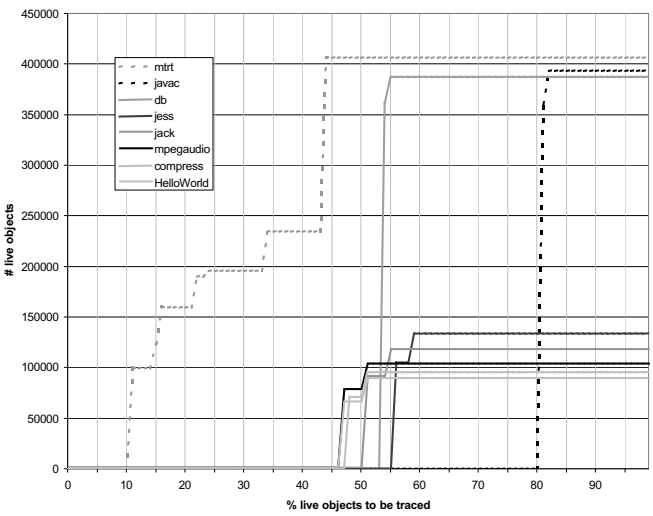
Dieckmann and Hölzle [14] measure the distribution of object lifetimes, sizes, and types and the reference density (fraction of fields that contain pointers) for the SPECjvm98 benchmarks. They focus on traits inherent in individual objects, whereas we study connectivity between various objects and how it correlates with lifetime.

Shuf *et al.* [31] study the cache and TLB behavior of the SPECjvm98 benchmarks and pBOB. They use the Jikes RVM to trace high-level heap accesses and then use a simulator to correlate cache and TLB misses with object sizes and layouts.

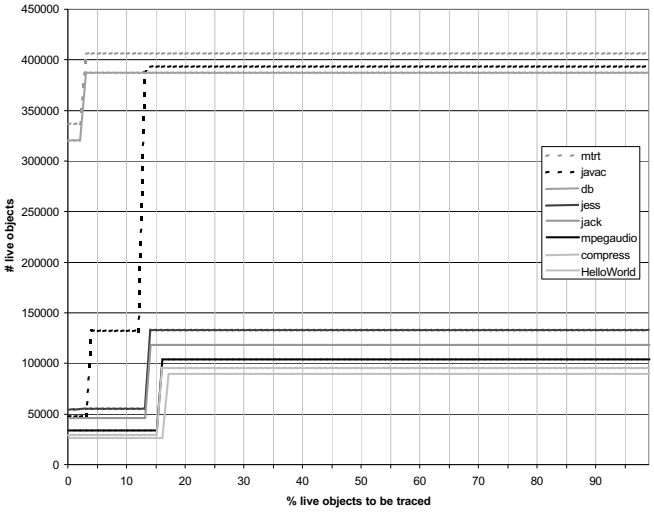
6.2 Generational garbage collection

There has been significant prior work on collectors that partition objects by age [26, 27, 39, 25, 42]. The most common of these collectors are generational collectors. Generational collectors generally have poor performance if a significant number of objects do not obey the generational hypotheses.

Prior work has proposed many variations and enhancements to generational collectors. Objects that are expected to live long can be pretenured [40, 4]. Pretenuring avoids having to repeatedly copy objects, but it typically requires profile information. Wilson *et al.* [43] describe an alternative to the breadth-first Cheney copying used in most garbage collectors [9]. Wilson's scheme groups an object with its immediate children and thus hopefully improves the spatial



(a) Number y of objects that are reachable from at most x% of the live objects, where reachability is based on static type information.



(b) Number y of objects that are reachable from at most x% of the live objects, where reachability is based on the actual pointers in the heap.

Figure 5: Reachability in snapshot of heap.

locality of data accesses. Chilimbi and Larus [11] describe an enhancement to generational garbage collection for improving data cache behavior.

Our results point at several possibilities for improving generational garbage collection. For example, our results suggest that moving an object near its connected objects is often a good idea since connected objects have a similar deathtime. Thus, connectivity information may give us the benefits of pretenuring and locality optimizations without requiring profile information. We will explore improvements to generational collection based on connectivity information in future work.

6.3 Segregating objects by criteria other than age

Region-based memory management can be viewed as an alternative to both explicit memory management and garbage collection. Allocation sites are annotated such that they allocate objects into separate regions. Deallocation points are determined statically, but the granularity of deallocation is an entire region, not an individual object. The annotations can either be performed automatically (for functional languages) based on a program analysis [38], or manually by the programmer [17].

Contaminated garbage collection does a runtime analysis to track the lowest activation record (the one closest to the bottom of the stack) from which an object is reachable [8]. Objects are only collected when the activation record associated with them is popped. In some ways, this technique can be thought of as a runtime region analysis.

If there is an ownership relation between two objects such that the owned object dies before the owner, and if the owned object has a fixed size, it may be inlined into the owner [15].

In contrast to generational collection, the above techniques segregate objects by deathtime (at some granularity) rather than by age. Using our connectivity results we hope to bring some of the benefits of segregation by deathtime to generational collection.

Harris [20] describes a variation of Baker's Treadmill collector [3] that segregates objects by connectivity and types. Section 5.4 discusses his paper in more detail.

Memory managers often segregate objects by size [5] or even by type [30], which enables some implicit bookkeeping of an object's location, instead of explicitly storing additional information.

Seidl and Zorn [29] segregate objects based on their memory access behavior in the context of an explicit deallocation system. The segregation improves locality and reduces the active working set. We expect that organizing objects based on their connectivity will have similar benefits with respect to memory system performance.

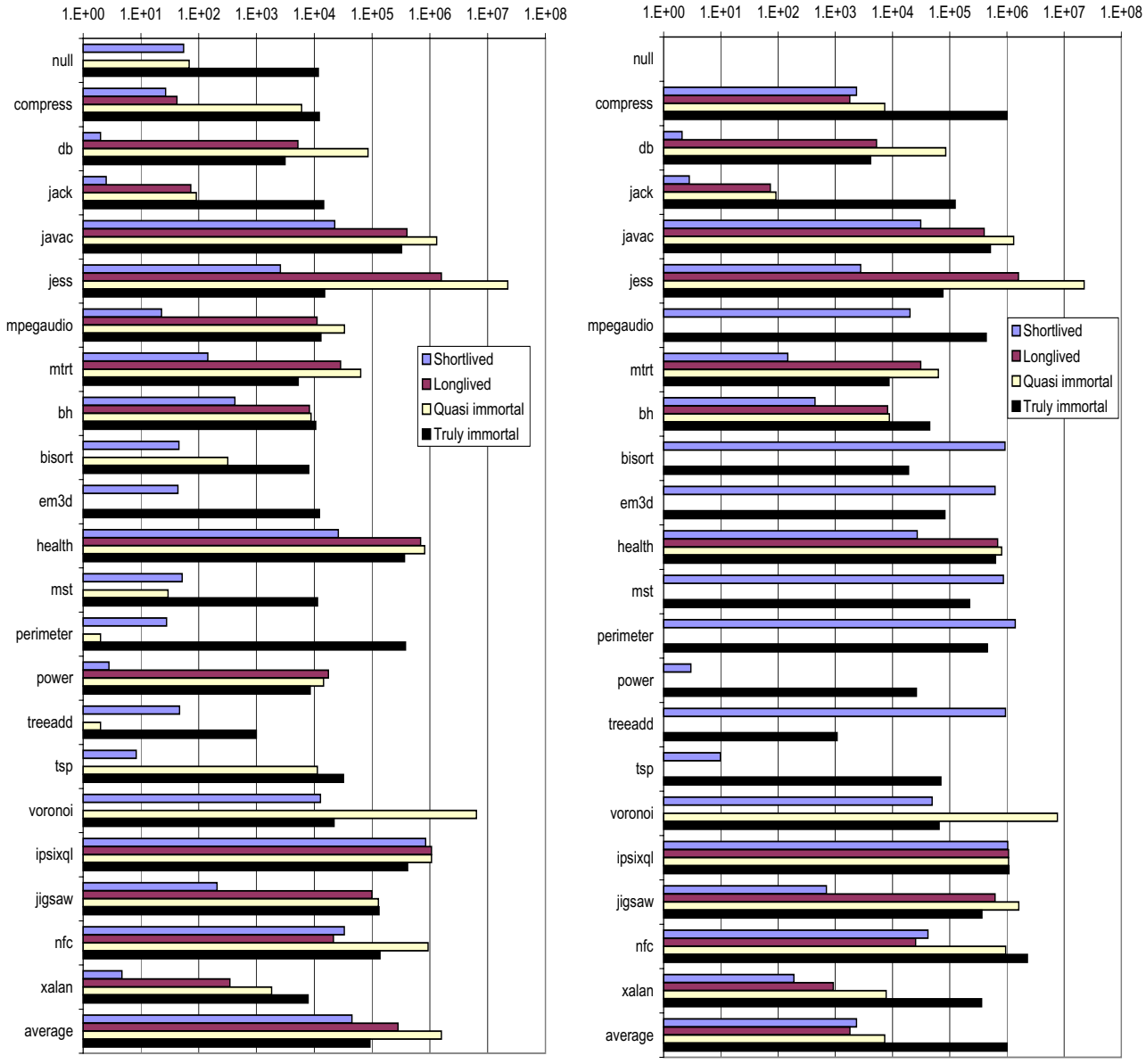


Figure 6: Average number of objects from which each object can be reached in the *global* object graph.

6.4 Escape analysis

If the lifetime of a data structure ends before the routine that allocated it returns and the size of the data structure is bounded, it can be allocated on the stack instead of the heap. Analyses that try to determine these properties of objects are called escape analyses [12, 28, 18, 41, 36]. Some escape analyses focus on objects that escape a thread (e.g., [33]). Our escape behavior numbers help judge the potential benefit of escape analyses for garbage collection.

7 Conclusions

This paper explores object connectivity and its relationship with object deathtime and lifetime. We classify connectivity into three categories: (i) connectivity from stack variables; (ii) connectivity from global variables; and (iii) connectivity from heap objects. We consider both direct connectivity (e.g., object O_1 points to object O_2) and transitive connectivity (e.g., object O_1 is reachable from object O_2).

Our results demonstrate that many kinds of connectivity correlate strongly with object deathtime or lifetime. More specifically, we find that (i) objects that are reachable only from the stack are usually shortlived; (ii) objects that are reachable from globals are usually quasi immortal or truly immortal; and (iii) objects that are connected via pointers (directly or transitively) usually die at the same time. Since our infrastructure (Jikes RVM) uses the same heap as the application, we present results for both all objects (including objects created on behalf of the Jikes RVM) and application objects (objects created on behalf of the application only).

In summary, our results provide valuable information on object behavior, which should be useful in both improving existing collection algorithms and designing new collection algorithms.

Acknowledgements

We thank Matthew Hertz for his help with tracing methodology. We thank David Bacon, Perry Cheng, Kathryn McKinley and the anonymous reviewers for their insightful comments.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[3] H. G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, 1991. Also appeared in *SIGPLAN Notices*, March 1992.

[4] S. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[5] H. Boehm, A. Demers, and M. Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[6] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, and H. Srinivasan. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, San Francisco, CA, June 1999.

[7] B. Cahoon. Java-Olden benchmarks. http://www-ali.cs.umass.edu/~cahoon/olden.

[8] D. Cannarozzi, M. Plezbert, and R. Cytron. Contaminated garbage collection. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[9] C. J. Cheney. A non-recursive list compaction algorithm. *Communications of the ACM (CACM)*, November 1970.

[10] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[11] T. Chilimbi and J. Larus. Using generational garbage collection to implement cache-conscious data placement. In *International Symposium on Memory Management (ISMM)*, 1998.

[12] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT press, 1990.

[14] S. Dieckmann and U. Hölzle. A study of allocation behavior of the SPECjvm98 Java benchmarks. In *European Conference for Object-Oriented Programming (ECOOP)*, 1999.

[15] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[16] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *International Symposium on Memory Management (ISMM)*, 2000.

[17] D. Gay and A. Aiken. Memory management with explicit regions. In *Programming Languages Design and Implementation (PLDI)*, 1998.

[18] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction (CC)*, 2000.

[19] T. Harris. Early storage reclamation in a tracing garbage collector. *ACM SIGPLAN Notices*, April 1999.

[20] T. Harris. Dynamic adaptive pre-tenuring. In *International Symposium on Memory Management (ISMM)*, 2000.

[21] B. Hayes. Using key object opportunism to collect old objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1991.

[22] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *ACM SIGMETRICS*, 2002.

[23] M. Hirzel, A. Diwan, and A. Hosking. On the usefulness of liveness for garbage collection and leak detection. In *European Conference for Object-Oriented Programming (ECOOP)*, 2001.

[24] R. Hudson and E. Moss. Incremental collection of mature objects. In *International Workshop on Memory Management*, St. Malo, France, September 1992.

[25] R. Jones and R. Lins. *Garbage collection: Algorithms for automatic dynamic memory management*. John Wiley & Son Ltd., 1996.

[26] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetime of objects. *Communications of the ACM (CACM)*, 1983.

[27] D. Moon. Garbage collection in a large Lisp system. In *Lisp and functional programming*, 1984.

[28] E. Ruf. Effective synchronization removal for Java. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[29] M. Seidl and B. Zorn. Segregating heap objects by reference behavior and lifetime. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[30] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Principles of Programming Languages (POPL)*, 2002.

[31] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *SIGMETRICS*, 2001.

[32] Standard Performance Evaluation Corporation (SPEC). SPECjvm98 benchmarks. http://www.specbench.org/osg/jvm98.

[33] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *International Symposium on Memory Management (ISMM)*, 2000.

[34] D. Stefanović, K. McKinley, and E. Moss. Age-based garbage collection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[35] D. Stefanović and E. Moss. Characterization of object behaviour in Standard ML of New Jersey. In *Lisp and functional programming*, 1994.

[36] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[37] D. Tarditi and A. Diwan. Measuring the cost of storage management. *Lisp and symbolic computation*, 1996.

[38] M. Tofte. A brief introduction to regions. In *International Symposium on Memory Management (ISMM)*, 1998.

[39] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Practical Software Development Environments*, 1984.

[40] D. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *Transactions on Programming Languages and Systems (TOPLAS)*, 1992.

[41] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Programming Languages Design and Implementation (PLDI)*, 2001.

[42] P. Wilson. Uniprocessor garbage collection techniques. Accepted for publication in *ACM Computing Surveys*.

[43] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "static-graph" reorganization to improve locality in garbage collected systems. In *Programming Languages Design and Implementation (PLDI)*, pages 177–191, Toronto, Canada, 1991.