A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks

Sylvia Dieckmann and Urs Hölzle
University of California at Santa Barbara
Department of Computer Science
Santa Barbara, CA 93106
{sylvie, urs}@cs.ucsb.edu

TRCS98-33
December 3, 1998

Abstract. We present an analysis of the memory usage for six of the Java programs in the SPECjvm98 benchmark suite. Most of the programs are real-world applications with high demands on the memory system. For each program, we measured as much low level data as possible, including age and size distribution, type distribution, thread-local objects, and the overhead of object alignment. Among other things, we found that non-pointer data usually represents more than 50% of the allocated space for instance objects, that Java objects are fairly small, and that a large fraction of objects is never accessible outside the thread that allocated it.

On some printers, the graphs in this paper apparently can be hard to read, although they look fine on our printer. We are currently working on better plots; in the meantime, color Postscript/PDF files for online viewing are available at http://www.cs.ucsb.edu/oocsb/papers/TRCS98-33.color.ps (or .pdf) shortly after the submission deadline.

1. Introduction

Java has brought garbage collection to the mainstream, being the first truly popular language in the C/C++ tradition that requires garbage collection (GC). Since Java differs in many respects from other languages requiring GC, such as Smalltalk, ML, or Lisp, the GC behavior of Java programs may well differ from that of programs written in other languages.

To understand the GC performance of a system, one must study the allocation behavior of the targeted applications. Every GC implementation leaves room for a waggonload of knobs and levers which impact performance, but tuning is difficult since the right settings depend on the characteristics of the executed program, which again depends greatly on language features and implementation style. To better identify and optimize garbage collectors, implementors need detailed empirical information about the allocation behavior of applications. For that reason, many published studies analyze the allocation behavior in the context of several languages, including Smalltalk, SML/NJ, Lisp, C, and C++ [Ung86, SM94, HMN97, HH+98, Zor89, ZG92, DDZ94]. However, no in-depth analysis of Java programs has been published to date, in part because of the lack of a standardized benchmark suite.

In this paper, we present the first in-depth analysis of the memory usage of realistic Java programs. Our study is based on the programs of the SPECjvm98 benchmark suite recently released by the System Performance Evaluation Corporation [SPEC98]. Most programs are real-world applications with high demands on the memory system.

For each program, we measure as much low-level data as practical. To test the generational hypothesis, we measure age distributions. To allow implementors to judge the impact of segregating objects by type or size, we analyze the heap composition and identify object groups (e.g., reference-free instance objects) which might benefit from a special treatment. To determine the impact of 8-byte object alignment, we simulate its effects. Since every system is different and there are so many possible GC variants, we generally refrain from making recommendations based on the data. Instead, our objective is to provide the GC community with detailed data that allows researchers to predict the impact of many GC implementation decisions for Java applications.

The remainder of this paper is organized as follows. Section 2 discusses related work and previous studies of allocation behavior, and section 3 describes the benchmarks. Section 4 presents our experimental setup. In Section 5 we discuss the results including our observations on object lifetimes, reference density, heap composition, thread locality, and others. Section 6 compares our numbers to those reported for other languages where possible, and section 7 summarizes our results.

2. Related Work

When implementing a language with garbage collection, it is essential to understand the expected allocation behavior. Since allocation patterns not only depend on the executed applications but also more generally on language characteristics, researchers have studied this question independently for several programming languages. Most of the earlier papers focus on lifetime and survival rates in order to estimate the overhead for generational GC. Later, after the basic characteristics of GC were understood, researchers became more interested in segregation approaches, special allocation strategies and others.

Ungar, for example, who first implemented Generation Scavenging [Ung84] in Berkeley Smalltalk (BS), analyzed the dependencies between survival rate and nursery size for Smalltalk-80 [Ung86]. Baker suggested theoretical models to explain allocation behavior [Bak94, Bak93]. Hayes analyzed survival rates for long-lived objects in Cedar, a Modula-like language, and found that objects that survive a certain age tend to die in clusters [Hay91].

Zorn presented statistical numbers on eight large Lisp applications analyzed with an object-level runtime system simulator [Zor89]. Unlike in most other studies (including ours), Zorn used memory reference counts as a metric for object lifetimes. More recently, Zorn et al. have studied large C/C++ programs, often in the context of lifetime prediction and memory allocation [BZ93, ZS98, GJS96, ZG92, ZG94].

Stefanovic and Moss analyze the allocation behavior of SML/NJ [SM94]. Gonçalves discusses object age distribution in his study on cache performance [Gon95]. We discuss these studies further in section 6.

Nettles et al. developed Oscar [MHN97], a language-independent GC testbed that can be used to analyze object allocation behavior. Unlike our simulator, Oscar does not trace heap activity but records frequent heap snapshots. Hicks et al. used Oscar to analyze both SML/NJ and Java applications [HMN97, HH+98] but mostly focus on execution time. Except for those studies, Java has not yet been the focus of an in-depth allocation behavior study. To the best of our knowledge, no other analysis of the SPECjvm98 benchmark suite has been published yet.

3. Benchmarks

All of our measurements are based on six programs from the SPECjvm98 benchmark suite, released in August 1998 by the System Performance Evaluation Corporation (SPEC). SPEC is a nonprofit organization of hardware vendors whose objective is to establish a standardized set of vendor-neutral, relevant, application-oriented benchmarks applicable to the newest generation of high-performance computers. Other popular SPEC benchmarks measure various system aspects such as CPU, NFS, and Web server performance.

SPECjvm98 is shipped as a set of Java class files and is intended to measure the efficiency of Java Virtual Machine (JVM) implementations, i.e., the combination of JIT compiler, runtime system, OS, and hardware platform. The individual programs were chosen by the SPEC member companies based on several criteria including high bytecode content, flat execution profile (no tiny loops), repeatability, heap usage and allocation rate, and either I-cache or D-cache misses on the reference platform. Most of the tests represent real applications and use both, integer and floating-point computation, library calls, and some I/O; however, AWT (window), network, and graphics are not covered in this suite. As a result, the SPEC benchmarks all execute little native code in the Java System Classes, and no program contains application-specific native code. All programs except one (mtrt) are single-threaded.

Program	Description	class file size (Kbytes)[a]	max. live heap (Mbytes)	total allocation (Mbytes)[b]	time (sec)[c]	heap loads * 10^6	% of loads for references	heap stores * 10^6	% of stores for references	load / store ratio
check	test JDK and Java features	5.8	n/a			n/a				
compress	Utility to compress/uncompress large files, based on Lempel-Ziv method; performs several passes over same data	17.4	6.7	105	1175	2,423	49	592	0	4.1
db	Small data management program; performs DB functions on memory resident database	9.9	7.2	231	505	712	56	42	70	16.7
jack	Parser generator with lexical analysis, early version of what is now JavaCC; performs several passes over same input	129.4	1.2	61	455	627	49	289	46	2.2
javac	The JDK 1.0.2 Java compiler compiling 225,000 lines of code	548.3	6.5	111	425	331	42	92	21	3.6
jess	Java expert system shell; based on NASA's CLIPS expert system	387.2	1.1	161	380	341	57	30	26	11.2
mpegaudio	MPEG-3 audio stream decoder	117.4	insignificant		1100	n/a				
mtrt	Dual-threaded raytracer	56.5	3.0	147	460	372	54	54	6	7.0

Table 1. SPECjvm98 programs

[a] Total size of all class files that where delivered as part of the actual application. These numbers do not include the harness code or JVM system classes.

[b] These numbers are based on our simulation and exclude space consumed by alignment, handle space, extra header words etc.; see section 4.3 for a details. The SPECjvm98 documentation reports significantly higher numbers, but we have verified that real JVM implementations (e.g., Sun's JDK 1.2) closely correlate with the numbers given here.

[c] Run time on the SPEC reference machine, a 133MHz IBM PowerPC 604 running an interpreter.

SPECjvm98 consists of eight different tests (see Table 1); seven are used for computing the performance score, one (check) validates the correctness of the VM. This test does not contribute to the result but must be executed correctly in order to obtain a valid score; we omit it from all our numbers. We also exclude mpegaudio since it barely allocates any data.

For all benchmarks, SPEC provides three different inputs referred to as "problem size 100, 10, and 1". Although the input names may suggest so, SPECjvm98 does not scale linearly (i.e., input size 10 does not run in 1/10th the space or 1/10th the time). Only the largest input may be used to publish benchmark results; all our runs use this input unless mentioned otherwise. Some of the programs (compress, jack, javac) iterate multiple times over the same input, which explains the repetitive shape of some graphs (for an example, see Figure 3).

In order to produce valid results, SPEC requires the user to run all applications through a harness which sets up the environment and times the experiment. In this study we too use the harness to start our programs and therefore add a constant but small amount of mostly long-lived data to all our results.

3.1 Program Descriptions

We now describe the six analyzed programs in more detail.

compress implements file compression and uncompression. It performs five iterations over a set of five tar files, each of them between 0.9Mbytes and 3Mbytes large. Each file is read in, compressed, the result is written to memory, then read again, uncompressed, and finally the new file size is checked.

In every cycle, compress allocates two large byte arrays for input and output; other than that, it allocates very few other heap objects. As a result, the live profile compress looks somewhat odd—rather than a continuous curve we see several vertical bars only, because we plot large object allocations as single data points followed by a gap whose length corresponds to the age of the object. Since compress has no long sequence of small allocation requests, no continuous curve can form. (Section 5.1 discusses our treatment of large objects in more detail.)

Although we believe that compress is not a typical representative of an object-oriented application we did not drop it from our suite since we assume that Java applications in the style of compress do exist. However, we sometimes exclude it from general statements made in this paper where it doesn't conform with the general trend.

db is the only program in this study that is not derived from a real-world application. It simulates a simple database management system with a file of persistent records and a list of transactions as inputs. The task is to first build up the database by parsing the records file and then to apply the transactions to this set. Accordingly, db has a very distinct live heap profile: the heap size grows linearly during the building of the database but stays at a fixed level for the entire time of the execution of the transactions.

jack is a commercial application (a parser generator) and is therefore shipped without source code. From our traces, we know that jack performs 16 iterations of building up a live heap structure and collapsing it again. According to the SPEC documentation, it repeatedly generates a parser from the same input. Because no data survives between iterations, jack can run in a fairly small live heap.

javac is the JDK 1.0.2 Java compiler iterating four times over several thousand lines of Java code; the source code of jess serves as input for javac.

jess is an expert system which reads a list of facts about several word games from an input file and attempts to solve the riddles. Although the run is computationally intensive and allocates a lot of memory, jess does not store temporary results for a prolonged time. Thus, the live heap stays at a relatively constant level for the entire run of the application, even though new objects are allocated continuously.

mtrt raytraces a picture by dividing the input data in sections and starting a new working thread for every section. It is the only multithreaded application in our suite. Unfortunately, even the largest problem size currently does not create more than two work threads. In addition, the threads do not yield voluntarily which means that our tracer, which is based on a JDK with cooperative thread model, executes them sequentially [Hol98]. We therefore hesitate to consider mtrt a truly multithreaded program.

4. Experimental Setup

Or experimental setup consists of two independent phases (Figure 1). First, an instrumented version of Sun's JDK 1.1.5 VM produces a trace file while executing the benchmark application. In the second phase, a simulator (written in Java) reads this trace and simulates allocation, pointer assignments, and garbage collections while computing the statistics shown in the rest of the paper. The next two sections describe these two components.

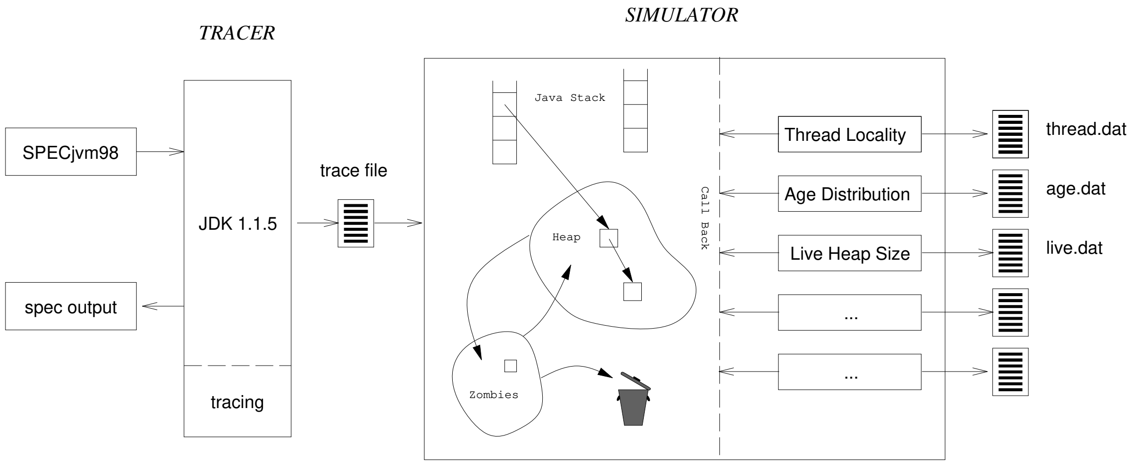

Figure 1. Experimental Setup

4.1 Tracer

We modified the Sun JDK1.1.5 VM to log all runtime information of interest. The recorded events include object creation, updates to the heap, stack, and operand stack, and method invocation. To reduce the trace size, we do not record updates to non-reference variables. Despite this restriction, and even with a reasonably space-efficient encoding of the trace events and subsequent compression with GNU gzip, trace lengths range up to 1.5 gigabytes. Together, the traces for all benchmarks comprise more than 4.5 gigabytes of data.

On object creation, the tracer records the class, bytes allocated (for arrays), and object ID. Since the JDK1.1.5 can relocate heap objects we only use *handle* addresses—not real addresses—for identification. The JDK VM assigns a handle to every heap object to support heap compaction. This handle is nothing more than a forwarding pointer; all object references use it rather than referring to the object directly. If the VM now decides to relocate an object, it updates the forwarding pointer (as opposed to all object references). Consequently, a handle address never changes during the lifetime of its object, which makes it an ideal object identifier.

Together with the recorded heap stores, the object creation data is sufficient to reconstruct heap structures and the complete object allocation history from the trace. The trace also contains method invocation/return events as well as the operands of any bytecode that affects pointers in the stack or operand stack; these trace records allow the simulator to reconstruct pointers from the stack.

We currently ignore pointer stores from C code, since the JDK 1.1.5 VM contains too many places that directly manipulate Java objects without going through the JNI interface. In order to determine the impact of this simplification, and to compensate for it in the simulator, the trace file augments some events with the expected value of the result, even though the simulator should be able to deduce it from the current state. For example, pointer loads record the value being loaded. For the same reason, the trace also records the handles of objects freed by the JDK VM, although we are not interested in (or dependent on) the JVM's GC algorithm. Fortunately, omitting stores from native code has a negligible impact on our results; we will discuss the exact impact in detail in the next section.

4.2 Simulator

The second component of our experimental setup consists of a heap simulator written in Java. The simulator reads and interprets the trace file to reconstruct heap and stack activity, and performs a garbage collection at fixed intervals to determine object lifetimes. For every Java object allocated by the application, the simulator creates a SimObject instance to hold the pointer fields of the application object, to capture a variety of GC-relevant information, and to provide additional space for temporary counters and markers.

While simulating the heap activity of the application, the simulator gathers a variety of statistics. All of our experiments are coded as independent units, each of which creates its own, sometimes partially redundant, output data. Most experiments are event-driven, registering their interest in certain events (e.g., new_object, free_object) so that the simulator calls them back whenever these events occur.

Since most of our experiments distinguish between live and dead objects, the simulator performs exact garbage collections on the simulated heap. Currently, we force a full collection after every 50 Kbytes[1] of allocation; we believe that a finer resolution is unlikely to improve the result but would slow down the simulation significantly. (A typical simulation currently takes about four days on a 300MHz UltraSPARC workstation.)

When the simulator determines that an object is unreachable, it notifies the registered statistics objects and moves the SimObject to a “zombie” region. Because the trace does not contain references from the JVM’s C code, an object might appear dead in the simulated heap but actually is still alive in the real application. Once the trace reports that the JDK GC has indeed freed the zombie object, the simulator discards it. However, if the object is still live, the simulator will discover that the next time it is referenced. In that case, the simulator resurrects the object and notifies the statistics objects to correct their data. As discussed in the previous section, the trace data contains some redundant information to detect these premature deaths and to recover from them.

	total objects allocated	objects resurrected	%	corrected references
compress	6607	162	2.15	3
jess	7924698	226	0.00	3
db	3211569	146	0.00	3
javac	6099430	21336	0.35	78203[a]
mtrt	6587012	106	0.00	3
jack	6863757	16278	0.24	3

Table 2. Error due to simulation

[a] Exclusively due to recovery point in event get_field

Table 2 shows that the simulator often resurrects fewer than 250 objects. Only in javac and jack does it retrieve a larger amount of zombies, but even there those objects make up for less than 0.4% of all allocated objects. In addition, a single object might get resurrected multiple times, so that the actual error would be even smaller than shown. Also, any error will show up only for a limited segment of the simulation, that is, from the point where the simulator falsely collects an object until it gets resurrected (detected error) or the JDK1.1.5 frees the original.

We believe these numbers are a strong indication that the inaccuracies introduced by ignoring native code are minor and do not influence our results. Because none of the applications in SPECjvm98 adds its own native code, only native code in the standard Java system classes could potentially cause a discrepancy.

[1] All graphs in this paper are based on 50 Kbyte intervals. During our analysis we also used 10 Kbyte intervals for better accuracy. However, we found no significant differences between 10 Kbytes and 50 Kbytes.

To verify the correctness of the Simulator, we compared the simulation results to numbers reported by the tracing VM. For example, we ran the Tracer with -verbosegc and compared the result to our simulator data in order to verify total allocation size. We also added extra counters to monitor the overhead of 8-byte alignment, the amount of memory still live at the end of the applications, and many others. All numbers obtained by the Tracer agreed closely with those of the Simulator. When adding experiments to the Simulator, we usually implemented two algorithms and made sure the outcome was identical. For example, we would update the live heap counter dynamically on every death event. In addition, we would periodically scan a snapshot of the live heap and determine the size. Finally, we cross-checked all our experiments for consistency. The sum of all 'live bytes of type X' numbers in one experiment, for instance, must correspond to 'currently live bytes' measured in another experiment.

To make sure that our Tracer did not miss important events, we kept a log of all warnings issued by the Simulator due to resurrection, recovery points and other state inconsistencies. If warnings appeared multiple times, we isolated the problem in the trace file and debugged the Tracer.

4.3 Object Model

To keep this study as general as possible, we only consider aspects that are likely to occur universally in Java implementations. That is, we abstract away effects that are implementation dependent, such as fragmentation, object or field alignment, special object headers, etc. The resulting object model (see Figure 2) is simple yet close to that of a realistic implementation.

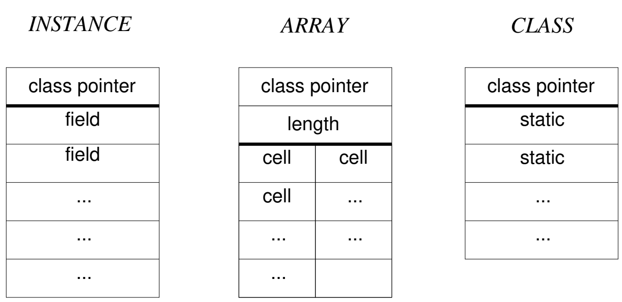

Figure 2. The simulator's object model.

Regular objects contain a class pointer and an arbitrary number of fields; we assume that there are no additional header words. All fields are word aligned and the simulator does not attempt to pack smaller fields. For example, a Point object with integer fields x and y consumes 12 bytes of heap space. Similarly, arrays also contain a class pointer, followed by a 32-bit length field and enough space to host *cell_size x length* array elements, where the cell size (1, 2, 4, or 8 bytes) depends on the array element type. For example, an integer array of size 1, a char array of size 2, and a byte array of size 4 all consume 12 bytes. Since virtually all processors either require 32-bit aligned addresses or impose a significant penalty on nonaligned addresses, we assume that all objects are aligned to 32 bits, so any fractional words (e.g., in a byte array of size 1) are padded up to the next word boundary. We make no further alignment assumptions (such as 64-bit alignment for long or double) since the space impact of such alignments depends heavily on allocator choices (e.g., unified heaps vs. BiBOP allocation).

Similarly, we do not reserve space for an element type pointer in reference arrays even though this type is required for array store checks. Some VMs (e.g., Sun's JDK 1.1.5 VM) add this extra word to all reference arrays. But a VM could also store the element type in the array class object, eliminating the per-array overhead. Since reference arrays are usually fairly large, we don't expect this issue to make a difference one way or the other.

Finally, we assume class objects to have a class pointer and enough space to accommodate the static fields, but we ignore all metadata such as method blocks, field blocks, constant pools, etc. Even though the class metadata might add up to several Kbytes of data, it is up to the JVM implementor to decide where and how it should be stored. For example, the JDK 1.1.5 allocates a 100 bytes struct on the heap for every class object which points to the remaining metadata in the C heap. In any case, the total allocation for all six SPECjmv98 programs is so high that the overhead added for class objects has no influence on any of our results.

5. Experimental Results

5.1 Heap Size and Object Lifetimes

Figure 3 shows the amount of live data for each application; for better readability the graph on the right shows an enlarged subsection of the same data. As is customary in GC-related work, we measure time (x-axis) in terms of Mbytes allocated. This metric is popular because the number of bytes allocated correlates directly with the amount of work that allocator and GC have to invest.

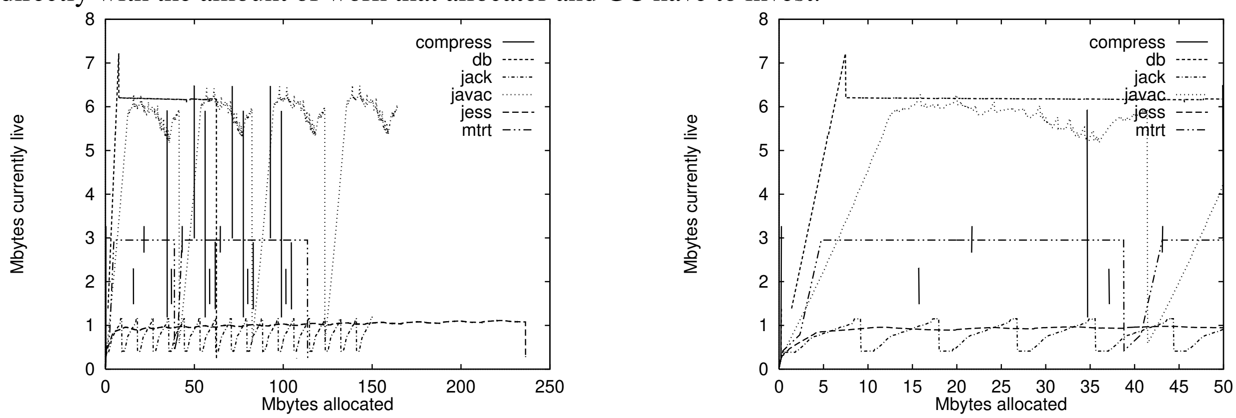

Figure 3. Minimal size of live heap (total and enlarged).

The graphs show the amount of bytes that are currently live over the total amount of space allocated up to that point. For example, after 10 Mbytes of allocation, the live heap of db is slightly larger than 6 Mbytes. The right graph shows an enlarged portion of the left graph.

The peak value of each curve in Figure 3 indicates the minimum heap size required to run the program to completion. The data points of all graphs are spaced 50Kbytes apart since the simulator performs a garbage collection after every 50Kbytes of allocation. jack, for example, generates a very distinct live heap curve which oscillates sixteen times between 0.4 and 1 Mbyte, with peaks spaced at 9Mbytes, because jack allocates approximately 9Mbytes in every of its sixteen iterations. Most of this data dies soon but a little more than 1Mbyte stays alive until the end of one cycle, at which point it collapses into a remainder of 0.4Mbytes.

If an allocated object is larger than 50Kbytes, the graphs will show a gap whose length corresponds to the size of the object just allocated. This effect is especially apparent in compress which allocates mostly large arrays. Because compress allocates virtually no other data, these arrays cause the curve to take the shape of a series of vertical bars were the height of each bar and the distance from the one to the right represents the size of the allocated object. (Refer to section 3.1 for a description of compress.)

Although all but one of the SPECjvm98 programs allocate more than 100 Mbytes of memory, they can all run in a fairly small heap (less than 8 Mbytes) if runtime performance is not an issue. Also, the maximum live heap size is established at a relatively early point in time, which may be important in the presence of automatic heap expansion; none of the applications would require a heap expansion after the first quarter of execution. jack and javac both repeatedly build up large live structures that collapse instantaneously when the phase terminates. db, jess and mtrt, on the other hand, maintain a constant amount of live data until the very end of the application, either because the same data structure is kept alive or, as in jess, because dying objects are continuously replaced with new allocations.

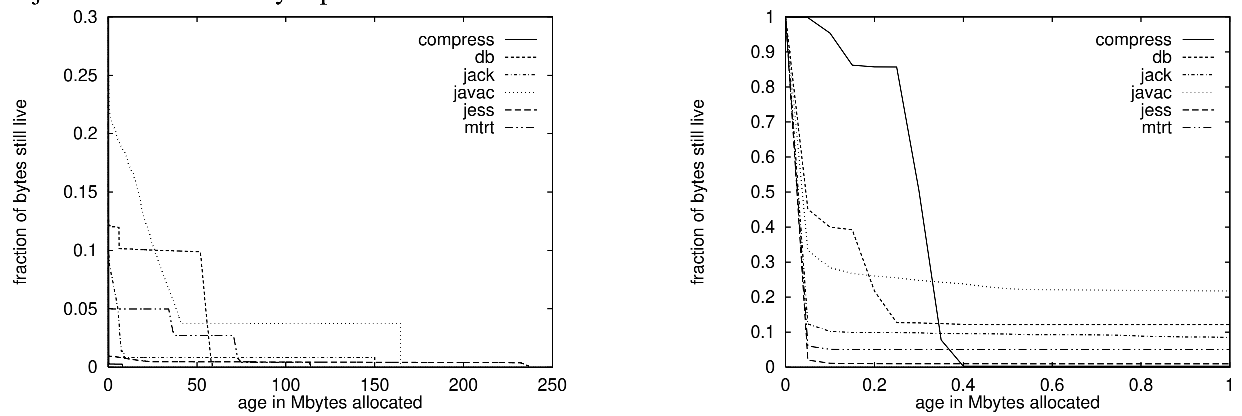

Figure 4. Age distribution in total allocated (total and enlarged).

The graphs show the fraction of bytes that are still live at time X after their creation; time is measured in allocated bytes. All curves must start at 1 since objects are always live immediately after their creation. For example, roughly 27% of all allocated bytes survive for 200 Kbytes in javac. Note that the two graphs are stretched in different dimensions; for better readability we clipped the y-axis of the left graph and the x-axis of the right graph.

Figure 4 shows the object age distributions. As in the previous figure, the x axis is measured in Mbytes of allocation, but for age distribution we treat large allocation requests differently by clipping them at 50 Kbytes. In other words, if a one-megabyte array becomes garbage before the allocation of the next object, its age is 50 Kbytes, not 1 Mbyte. We believe that clipping makes sense because the main reason the GC community is interested in age distribution is to estimate the success of generational GC and to determine the optimal configuration. Without clipping, large objects would otherwise automatically look long-lived even when, in fact, they die shortly after allocation. Although the allocation of a large object uses up heap space, it would not trigger an equal amount of GC work, thus justifying an age definition that presents it as "young".[1]

In the SPECjvm98 suite, clipping only affects compress, which allocates around 105 Mbytes but whose age distribution appears to end at 8Mbytes. Essentially, after allocating a large byte array compress performs long allocation-free computations after which the array becomes garbage. Thus, even though the array is live for many seconds of real time, it appears very short-lived to the garbage collector.

The age distributions of the SPECjvm98 applications confirm the weak generational hypothesis [Gon95] that most objects die young, although the effect is not as pronounced as for other programming languages. For example, Stefanovic and Moss report that only 2-8% of all allocated bytes survive for more than 100 Kbytes in four SML/NJ applications [SM94], and Ungar measured that only 6.6% of Smalltalk objects survive 140 Kbytes of allocation [Ung86]. In contrast, 1%-40% of the SPECjvm98 objects are still live after 100 Kbytes (with jess on the lower and db on the upper end, ignoring compress). Even after one megabyte of allocation, 21% of all allocated bytes are still live in javac, 12% in db, and 8% in jack.

This data implies that Java, too, can benefit from generational GC, which allows to collect younger objects more aggressively. However, the nursery of a Java heap should be substantially larger than for functional languages such as ML or Smalltalk.

The age distribution for older objects is very application-dependent; most applications show clusters of objects dying at roughly the same age, leading to sudden steep drops in the age distribution graph. Only jack and javac show relatively smooth age distributions. This observation is consistent with Hayes' study of the

[1] We are currently experimenting with an alternative treatment for large objects. Instead of clipping objects over 50Kbytes, we now dedicate a part of the simulator heap to a special large object area for allocation requests over 1Kbytes. Since we assume that objects in this area would be treated differently by a garbage collector, we plan to exclude them from normal age distribution graphs.

behavior of old objects [Hay91], and suggests that multi-generation collectors would not necessarily work for Java. Instead, it is worth investigating techniques such as Hayes' key object opportunism.

5.2 Instance Objects vs. Arrays

We now turn to analyses of heap composition, starting with the distinction between instance objects and arrays. A garbage collector may want to treat arrays specially, particularly if they are reference-free (e.g., strings). The left graph in Figure 5.1 shows that all applications initially allocate mostly arrays. Part of these

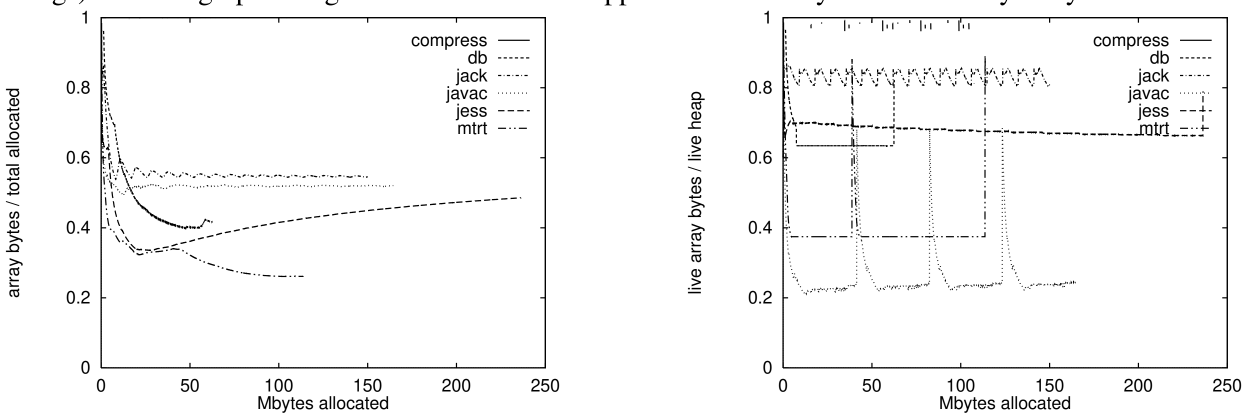

Figure 5. Fraction of array bytes

The graphs show the amount of array bytes (right graph) and live array bytes (left) as a fraction of total bytes allocated (left) and the live heap size (right); time is measured in allocated bytes. For example, after 100 Mbytes of allocation, 40% of jess' heap is used for arrays but close to 70% of all live objects are arrays.

objects can be attributed to JVM initialization; the VM allocates the first 160 Kbytes—mostly byte arrays—before the user application is even started. The SPEC benchmark harness and the actual benchmark setup also appear to require a higher ratio of array allocations. However, after 10-20 Mbytes of allocation most applications stabilize with an array to instance object ratio of around 1:1. Only mtrt allocates more than twice as many instance objects than arrays over the entire run.

The right graph of Figure 5 shows arrays as a fraction of live objects only. Here, the trends are less clear, and we cannot easily generalize them. However, although the fraction of arrays in the live heap varies significantly from 25% to close to 100%, every application appears to maintain a fairly constant ratio over the entire execution. Most applications do show several extreme spikes, but when correlated with Figure 3 it becomes clear that these spikes occur when most instance objects die at the end of an input data set. Apparently, the arrays are live for the entire execution time, thus causing the variations to appear disproportionately large when the live heap contracts sharply.

5.3 Reference Density

We now investigate how much of the allocated space contains references rather than primitive types (e.g., bytes or integers). Again, we distinguish objects by instance objects vs. arrays, since only instance objects can mix reference and non-reference fields. Although our object model includes a class pointer in every object, we do not count it as a reference but rather as a non-reference in this section. The class pointer may well be special-cased in a GC implementation since it is immutable unless class objects are copied. If garbage collection of class objects is switched off altogether (e.g., JDK1.1.5 with -noclassgc), it does not even have to be scanned.

Figure 6 shows that all applications allocate a high percentage of non-reference fields during startup. But even later, most programs allocate less than half of their space for references. Only jess and db allocate a higher fraction of references; the latter ends with a total of 66% allocated for references. Three of the programs, jack, javac, and jess, generally have a balanced live heap with between 45-50% of non-reference bytes. Again, the periodic heap contractions in these programs let these applications appear more irregular than they are. The other three programs also maintain a fairly constant reference density in their live heap with between 65% and 98% of the live space dedicated to non-references.

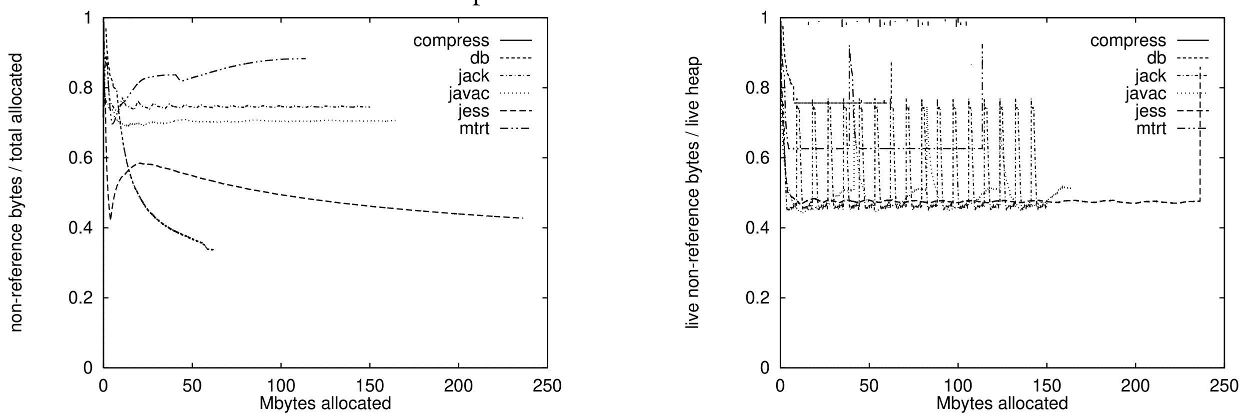

Figure 6. Fraction of non-reference bytes

The graphs show the amount of non-reference bytes (left graph) and live non-reference bytes (right) as a fraction of total bytes allocated (left) and live heap size (right); time is measured in allocated bytes. In jess, foor example, after 50 Mbytes of allocation, 55% are allocated for non-reference fields, but 47% of the live objects at this point are non-reference fields.

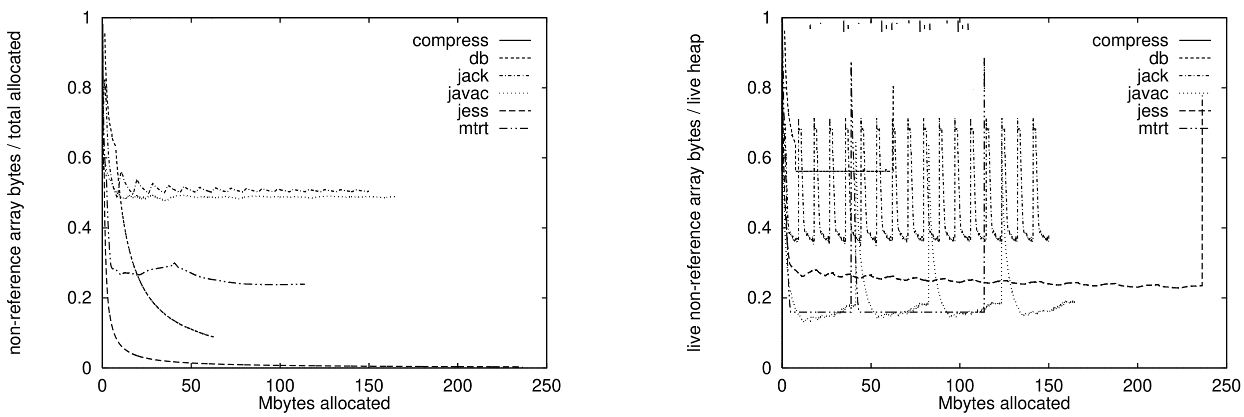

Figure 7. Fraction of non-reference bytes in arrays

The graphs show the amount of non-reference array bytes (left graph) and live non-reference array bytes (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In jess, for example, after 50 Mbytes of allocation, only 1.5% of the space is allocated for non-reference arrays, but over 25% of the live objects at this point are non-reference arrays.

This data helps estimate the number of pointers a collector must scan and update. However, some non-reference fields are easier to skip than others; in particular, non-reference arrays are faster to skip than non-reference fields scattered throughout instance objects. Thus, we now refine the data presented in Figure 6 by separating out arrays (Figure 7) and instance objects (Figure 8).[1]

The behavior for arrays varies widely among programs, with some allocating non-reference arrays almost exclusively (compress), others allocating hardly any non-reference arrays (jess), and the rest in-between.

[1] Unlike in Figure 6, the complements in these graphs do not show *(reference arrays)* and *(reference fields in instances)* but rather *(reference arrays + all instances)* and *(reference fields in instances + all arrays),* respectively.

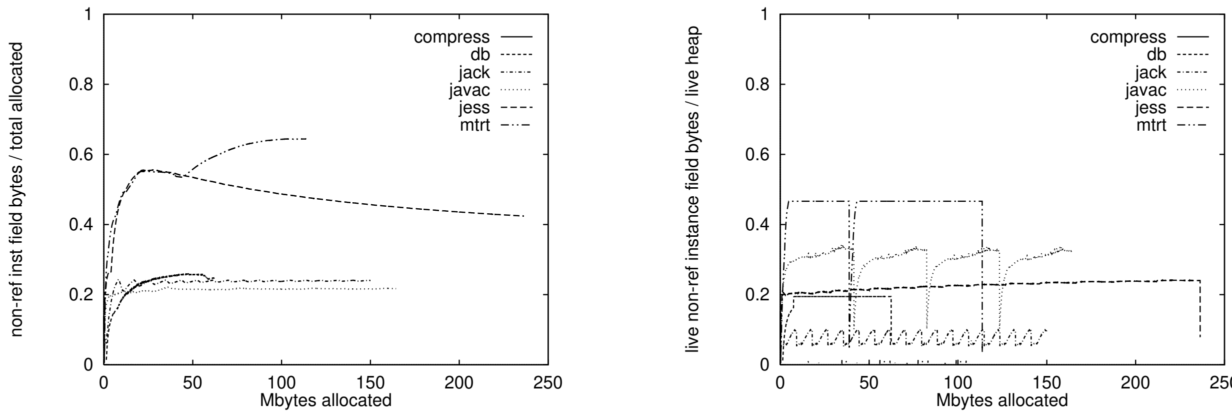

Figure 8. Fraction of non-reference bytes in instances

The graphs show the amount of non-reference instance bytes (left) and live non-reference instance bytes (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In jess, for example, after 50 Mbytes of allocation, over 50% of the space is allocated for non-reference instance field bytes, but only 20% of the live bytes at this point are non-reference instance field bytes.

Non-reference fields from instance objects show a somewhat more uniform pattern, with three applications around 25%, jess and mtrt considerably higher, and compress at zero.

Since we were surprised to see such a high fraction of non-reference fields in instances, we investigated the possibility of segregating instances that contain no references other than the class pointer. However, Figure 9 suggests that segregation is unlikely to work well in the general case; only mtrt creates a significant amount of reference-free instance objects.

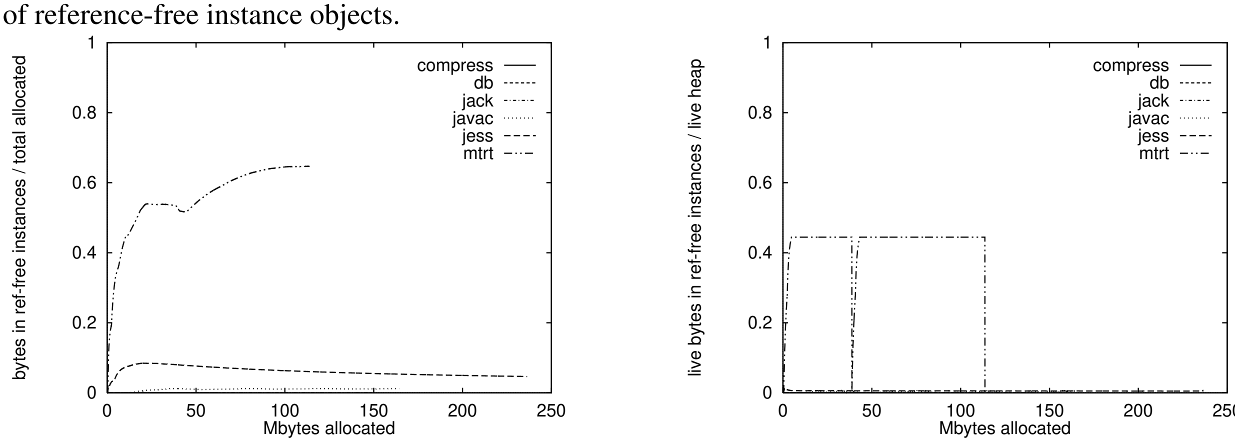

Figure 9. Fraction of bytes from reference-free instances

The graphs show the amount of bytes consumed by reference-free (expect for the class pointer) instance objects (left) and reference-free live instance objects (right) as a fraction of total bytes allocated resp. live heap; time is measured in allocated bytes. In mtrt, for example, after 100 Mbytes of allocation, 65% of the space is allocated for reference-free instance objects; 45% of the live bytes at this point are from reference-free instance objects.

5.4 Heap Composition

Since the previous section revealed that arrays and even non-reference arrays are fairly frequent, we now show what kinds of arrays occur most frequently.[1] None of the SPECjvm98 programs allocates a significant number of arrays of type boolean, double, float, int, or short. In Figure 10 as well as in all following

[1] Note that Java does not have true multidimensional arrays; for example, Java represents a two-dimensional int array as a one-dimensional *reference* array whose elements are int arrays (the rows of the two-dimensional array).

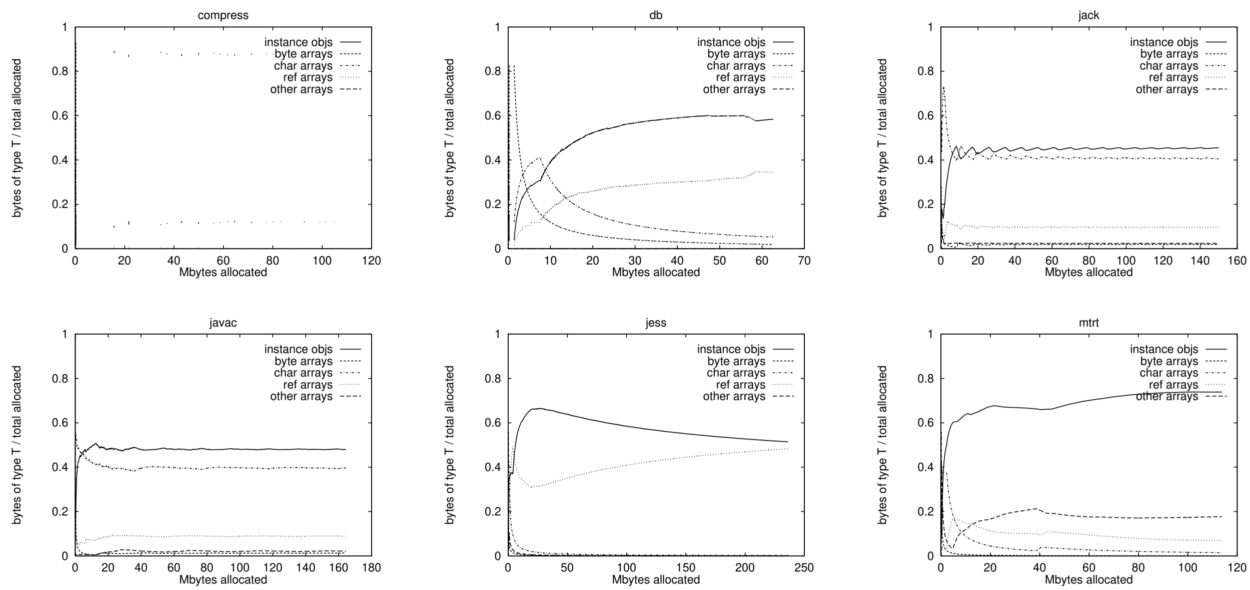

Figure 10. Heap composition for total allocated.

For every application the graphs show the fraction of the total allocated space that is dedicated to a certain object type; time is measured in allocated bytes. For example, after 100 Mbytes of allocation, the heap for javac consists of 48% instance objects, 40% char arrays, 9% reference arrays, 1% byte arrays, and 2% other array types.

graphs, these arrays are summarized as "other arrays." All programs allocate less than 2% for those arrays except for mtrt (up to 20%).

Other object kinds (instance objects, byte, char, and reference arrays) are all fairly common throughout the entire suite, although individual programs may not use certain types of arrays. Only instance objects are represented with at least 45% in all but compress. Often, instance objects and one other array type consume 85-99% of all allocated space. However, the dominant kinds depend entirely on the application: db and jess contain a high fraction of reference arrays, jack and javac both allocate around 40% for char arrays, and mtrt allocates instance objects almost exclusively. For some programs, the distribution stabilizes after some time (jack, jess) but not in others (db, mtrt). An informal study of other applications confirmed this trend: many applications allocate the bulk of their space for very few kinds, with one or two of them dominating the heap. These results suggest that it may be worthwhile to segregate non-reference arrays.

The picture changes when taking only live objects into consideration (Figure 11). As seen previously, the "live only" graphs contain some noise caused by the heap contractions at the end of each repetition. Ignoring these spikes, the live heaps show a relatively even distribution that often differs markedly from the numbers presented in Figure 10. For example, db allocates 58% of its space for instance objects, 34% for reference arrays, and only 6% for char arrays, but appears to keep most of those char arrays alive. Thus, its live heap contains about 50% char arrays but only 13% of reference arrays. In general, char arrays are more common in the live heap than their percentage of total allocation might suggest.

The differences between the composition of the live heap versus the total allocated suggest that basic types have different age distributions. For example, one would intuitively expect char arrays, which often represent strings, to be rather small and either very short-lived (for temporary results) or permanent. In the next few graphs we investigate age distribution and average object size separated by object kind.

Figure 12 and Figure 13 both show the same numbers (age distribution), but the former organizes the data per application whereas the latter presents it by object kind for the three most frequent kinds. To make the graphs easier to read, we eliminated any line that would represent less than 5% of total allocation; in that case, the line's label is marked with the letter X. However, is still useful to keep in mind that not all object

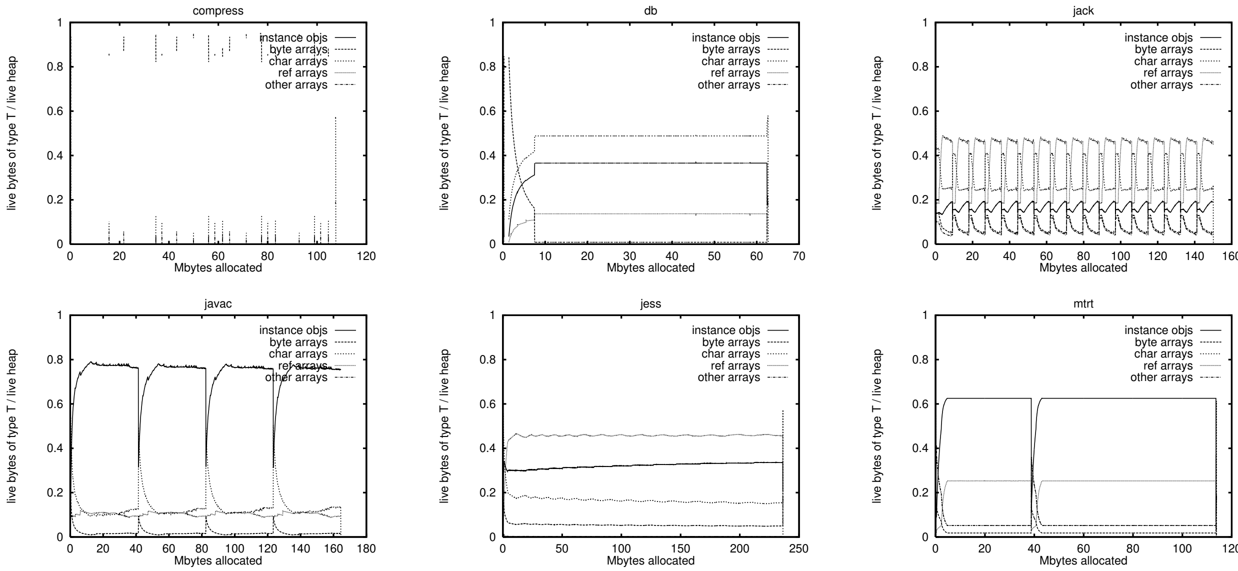

Figure 11. Heap composition for live only.

For every application the graphs show the fraction of the current live heap that is dedicated to a certain object type; time is measured in allocated bytes. For example, after 100 Mbytes of allocation, the live portion of the heap for jess consists of 46% ref arrays, 32% instance arrays, 16% char arrays, 5% byte arrays, and 1% other array types.

kinds are equally important, and to correlate these graphs with the actual heap composition shown in Figure 10.

As discussed in section 5.1, a large fraction of objects dies very young. In jess, for example, most objects don't even survive the first 50Kbytes of allocation which at this resolution makes the upper graph in Figure 12 appear blank. Often, most longer-living data is of one type. db keeps mostly char arrays, jack retains reference arrays, and mtrt uses some longer living arrays of various types. Again, the age distribution graphs show marked steep drop; only jess shows a smooth age distribution (especially for instance objects and reference arrays). It might be worthwhile to use an adaptive strategy which recognizes those long-lived types and collects them less aggressively.

Even compress seems to keep only instance objects for a prolonged time. However, in this case the numbers might be slightly misleading due to the fact that we clip large object allocations to 50Kbytes when measuring the age. compress uses several large byte arrays for arithmetic computation but it does not allocate more space during the lifetime of each of those arrays. Consequently, the byte arrays seem to die young.

Figure 13 shows that reference arrays tend to survive longer than instance objects and especially char arrays. They also have a smoother age distribution than the other two types. Character arrays, on the other hand, either die very young (as in jack and javac) or to stay alive for most of the execution time (as in db). This observation suggests that segregating all arrays into type-specific heaps and applying different collection strategies to them might pay off. However, our benchmark suite may be too small to draw reliable general conclusions about type-specific age characteristics.

5.5 Object Size

Table 3 lists the average object sizes, computed separately for each of the five frequent object kinds. Again, these normalized numbers can be misleading in cases where they are based on a small sample size. Thus, for each type the table includes the percentage of total allocation per entry, and entries based on less than 5% of total allocation are shaded.

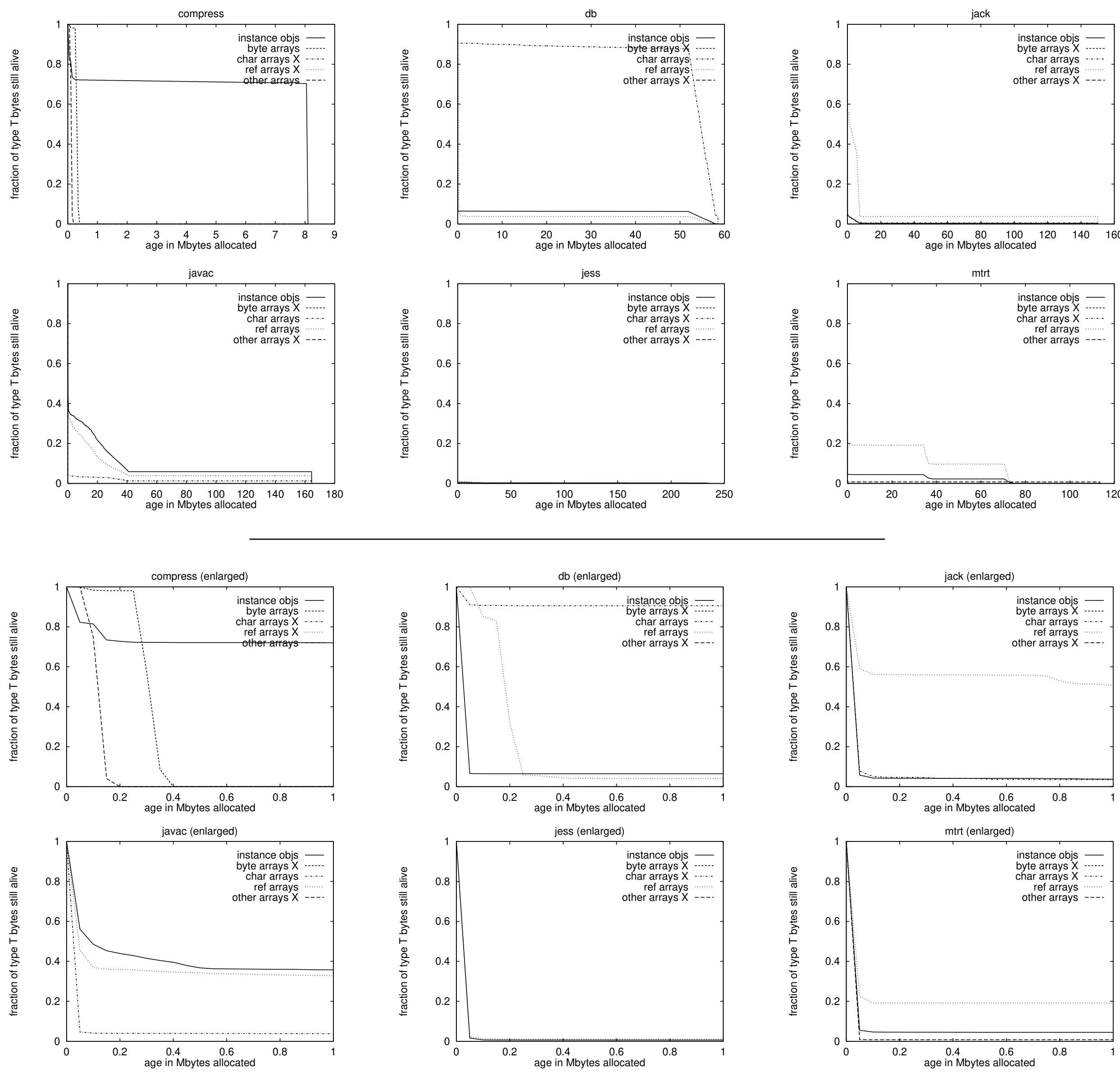

Figure 12. Age distribution. (Lower two rows show enlarged graphs.)

For every application the graphs show the fraction of bytes of a certain type that are still live at time X after their creation; time is measured in allocated bytes. For example, 22% of all instance object bytes, 13% of all reference array bytes, and 3% of all char array bytes survive the first 20 Mbytes of allocation in jess.

For better readability we prune all object types that represent less than 5% of the total allocation and mark their labels with the letter X.

Instance objects clearly are the smallest objects on the heap. Depending on the application, their average size varies between 16 and 23 bytes. But char arrays are also fairly small, with an average size between 26 and 42 bytes. All other array types are less predictable; depending on the application they can range between a few bytes and several Kbytes. Given the small average object size, it appears imperative for implementors to keep the number of header words to a minimum; for example, even a single extra header word per object will increase mtrt's heap by about 20%.

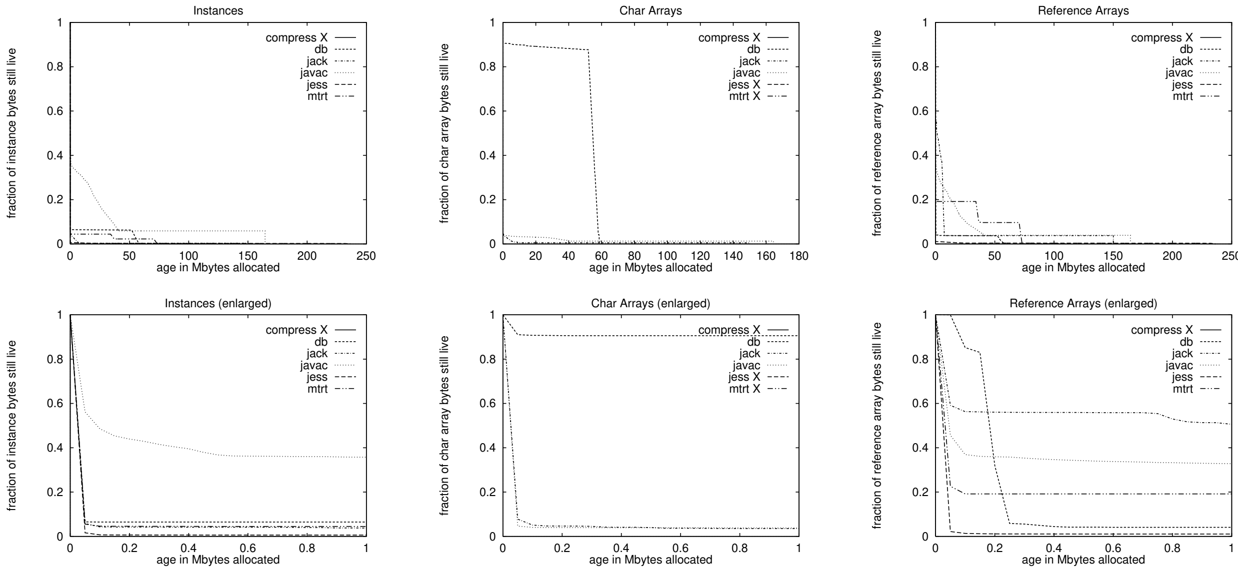

Figure 13. Age distribution for selected types. (Second row shows enlarged graphs.)

For every type the graphs show the fraction of bytes in any applicaton that are still live at time X after their creation. For example, 90% of all char object bytes in db, 3% in javac, and 0.5% in jack survive the first 20 Mbytes of allocation.

For better readability we prune all object types that represent less than 5% of the total allocation and mark their labels with the letter X.

	instance objects			byte arrays			char arrays			ref arrays			other arrays			sum		
	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total	mean	median	% of total
compress	17	16	0	238Kb	24	87	115	46	0	948	412	0	170Kb	1.9Kb	13	11.4Kb	20	100
db	12	12	58	1.2Kb	9	2	27	22	6	1.3Kb	48	34	227	170	0	19	12	100
jack	17	16	45	10	9	2	26	10	41	92	48	10	29	24	2	22	16	100
javac	19	16	48	460	264	1	42	40	40	41	12	1	140	120	2	27	20	100
jess	23	24	51	74	9	0	64	40	0	44	48	48	385	20	0	30	24	100
mtrt	16	16	74	151	9	0	64	72	2	32	32	7	20	20	18	17	16	100

Table 3. Average Object Sizes in bytes.
A cell is shaded if this type comprises less than 5% of the total allocated heap space

5.6 Object Alignment

Since all program heaps are dominated by relatively small objects, object alignment can add a significant space overhead. All numbers in this paper are based on 32-bit aligned object sizes as explained in section 4.3. However, real JVM implementations may often align objects to other boundaries to simplify memory management or to satisfy hardware alignment requirements. For example, on many processors 8-byte data (i.e., long or double) must be 8-byte aligned (e.g., on SPARC, Alpha, and PowerPC); otherwise unaligned data accesses may incur significant access penalties (e.g., on Intel Pentium II processors). Sun's JDK 1.1.5 VM, for instance, aligns all objects to 8-byte boundaries. This extra alignment increases the heap size by up to 19% (see Table 4). The programs that suffer most from alignment, jack and mtrt, both have a high percentage of small objects. The reverse implication (programs with many small objects show high overhead) does not necessarily hold since the size of frequently-allocated objects may be a multiple of eight.

	total allocated (Mbytes)	8-byte aligned (Mbytes)	increase (Mbytes)	% of total allocated
compress	105	105	0.01	1.1%
db	231	240	9.27	4.1%
jack	61	73	11.57	18.9%
javac	161	171	10.25	6.4%
jess	111	115	4.26	3.8%
mtrt	147	165	18.62	12.7%

Table 4. Heap size increase due to 8-byte alignment

5.7 Potential Thread Locality

Java provides support for multithreaded applications, which can add a considerable burden to several aspects of the JVM implementation, including the allocator and the garbage collector. For example, the allocator must synchronize concurrent allocation requests, and the collector may need to protect against concurrent mutators. Some of this overhead occurs even when the application is strictly single-threaded. Typically, the collector also needs to suspend all mutator threads before starting a collection. Segregating thread-local data from global data could improve garbage collection efficiency by allowing per-thread collection. In this section we investigate the possibility of *thread-local heaps*.

What is a thread-local object? Several definitions appear plausible, for example, objects referenced by a single thread only, or objects reachable by a single thread only. In this paper we concentrate on the latter definition—an object is *thread-local* if it is provably unreachable from other threads; otherwise it is considered *global*. In accordance with the Java Language Specification [GJS96], every object is allocated in a well-known thread context and is only accessible by this thread initially.[1] In addition, no other thread can obtain a reference to a local object *l* unless the creating thread explicitly gives it away (makes it global) by:

- assigning the reference to *l* to a class variable;
- assigning the reference to *l* to an already global object; or by
- turning another local object from which *l* is reachable into a global object (transitive closure).

Even when an object and its transitive closure are global, they may never actually be accessed by another thread, which makes our definition conservative. Once an object becomes global, we never revert it to local even if it later becomes again unreachable by other threads.

The remaining figures in this sections show various aspects of thread locality. Since an object can change from local to global at any point in time, a sudden drop in the percentage of local object bytes might be caused by a single large object becoming global, by a connected structure becoming global as part of a transitive closure, by the allocation of a large global object, or, when looking at live objects only, the death of local objects.

Local objects clearly dominate allocation (Figure 14). This result is surprising even though most of the SPEC applications are single-threaded, because it means that most objects are never reachable from the global heap. All applications allocate some global objects during startup, but soon the amount of global data drops to less than 1% of the total allocated space. Only jack maintains a constant level of around 8% of global data.

In contrast, the live heap as represented in the right graph of Figure 14 presents a very different picture. Although local data still dominates most live heaps, the actual fraction is much lower. The heap of jack

[1] With the exception of objects that extend java.lang.Thread or implement java.lang.Runnable.

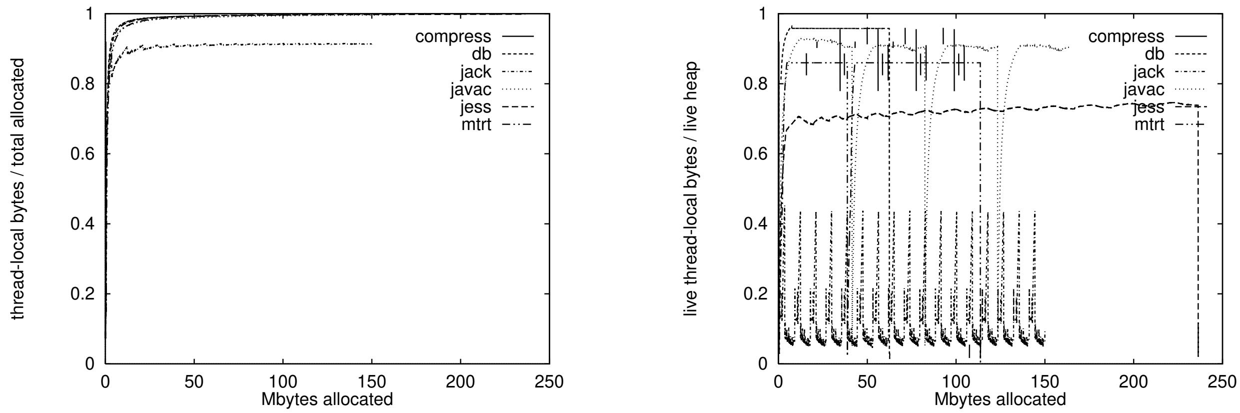

Figure 14. Fraction of bytes from thread-local objects

The graphs show the amount of thread-local bytes (left) and live tread-local bytes (right) as a fraction of total bytes allocated resp. live heap. For example, after 50 Mbytes of allocation, 99% of the heap in jess is used for thread-local objects; 70% of the live bytes at this point are dedicated to thread-local objects.

We consider an object thread-local if it can not be accessed by more than one thread, even though the application might currently be single-threaded.

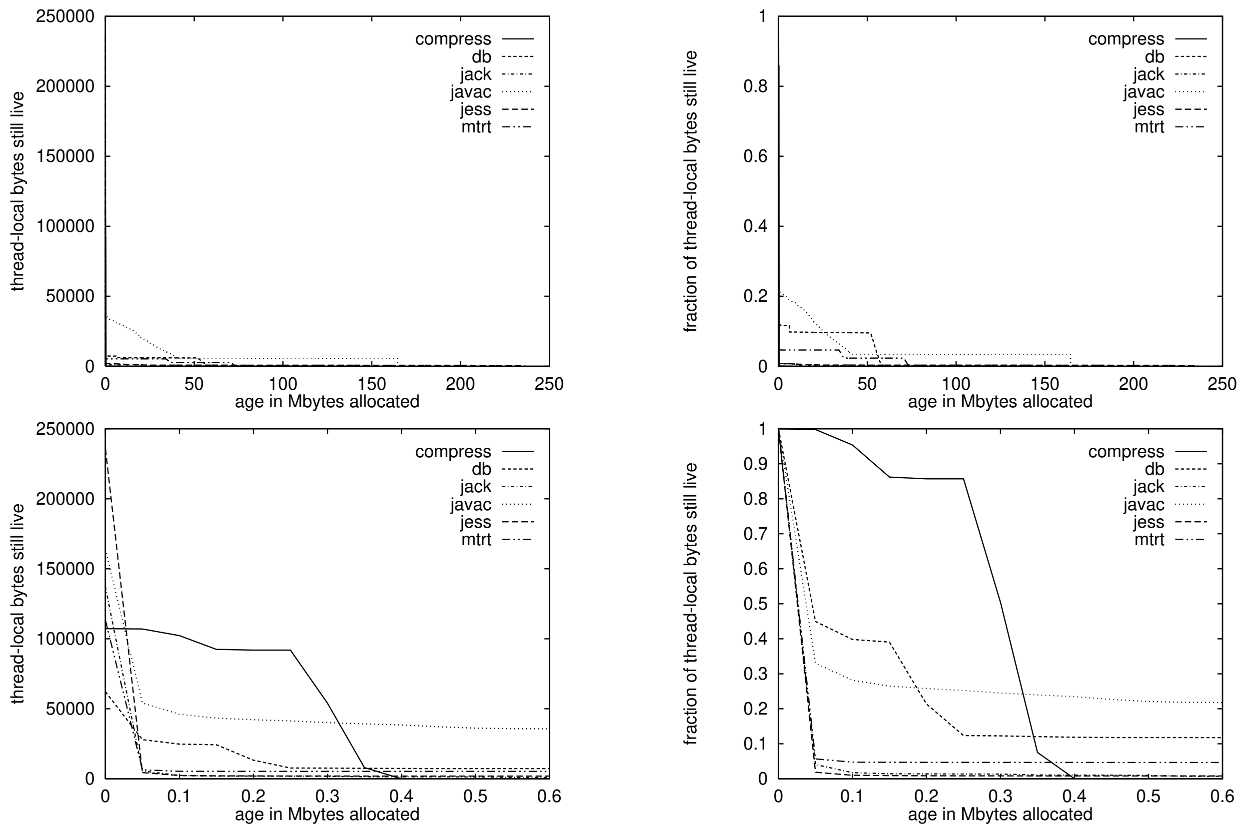

Figure 15. Age distribution for thread-local objects (absolute and relative numbers; enlarged graphs on bottom)

The graphs show the raw number (left) and the fraction (right) of thread-local bytes that are still live at time X after their creation. For example, 45 Kbytes (28%) of all thread-local bytes in javac survive the first 1 Kbyte of allocation.

We consider an object thread-local if it can not be accessed by more than one thread, even though the application might currently be single-threaded.

contains only between 5% and 43% local data, but all other programs still contain at least 50% local data. In general, global data appears to be more common on the live heap, and to live longer, surviving the regular heap contractions.

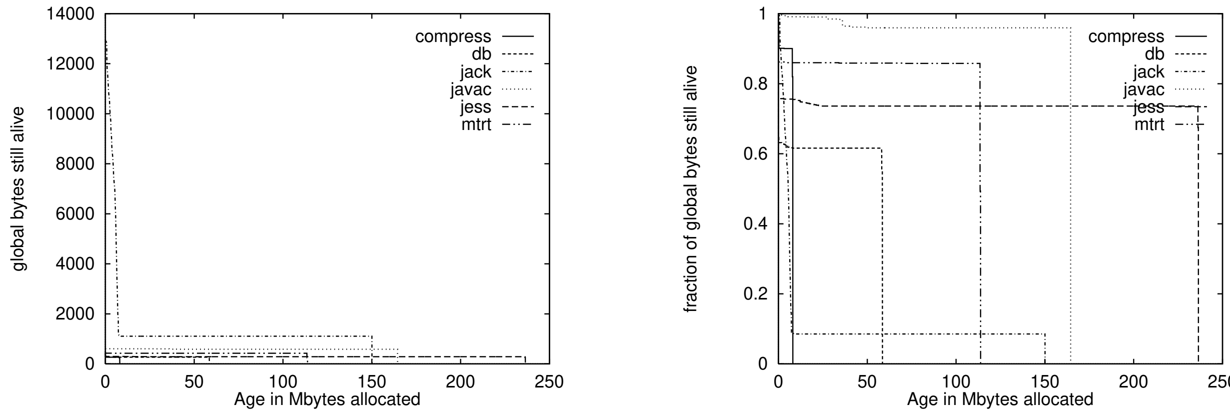

Figure 16. Age distribution for global objects (absolute and relative numbers)

The graphs show the raw number (left) and the fraction (right) of global bytes that are still live at time X after their creation. For example, 584 bytes (96%) of all global bytes in javac survive the first 50 Mbytes of allocation.

We consider an object global if it can be accessed by more than one thread, even though the application might currently be single-threaded.

Indeed, Figure 15 and Figure 16 confirm this observation. Although global data is far less frequent, it has a significantly higher life expectation. It appears that when an object becomes global, it is likely to live on for a very long time if not until the end of the application. Intuitively, this behavior could result from global roots (i.e., static fields) that are rarely updated. Unless a static field is overwritten, anything reachable from it will remain live until the end of execution (or until the class is unloaded, which does not happen for the programs in SPECjvm98).

The locality results must be interpreted with care, since we measured mostly single-threaded applications. However, we can expect at least some families of programs (such as server applications) to be highly multi-threaded with strong thread locality (e.g., pools of independent worker threads). In this case, using thread-local heaps might improve GC performance.

6. Comparison With Other Studies

In general, it is difficult to compare empirical GC studies because each study uses different assumptions (e.g., nursery size, large object area), different techniques (simulation, code instrumentation etc.), different metrics, different benchmarks, and different languages. Nevertheless, we attempt to make some comparisons below.

Barrett and Zorn report lifetime quantiles for five allocation intensive C programs [BZ93]. They find that 75% of all objects survive less than 849-32,000 bytes, and that 91%-100% of all objects are "short-lived", i.e, die within 32 Kbytes of allocation. Stefanovic and Moss report a similarly high object mortality for SML/NJ [SM94]; with the smallest nursery size setting of 32 Kbytes, the three ML applications presented have survival rates of approximately 2%, 7%, and 9%. In addition, the survival rate drops dramatically if the nursery size is increased to 250 Kbytes. The four Lisp programs measured in [Sha88] have a slightly lower mortality: between 5% and 25% survive the first 32 Kbytes, and approximately 2% to 8% are still live after 1 Mbyte. In comparison, Java objects appear to be more tenacious, with up to 99% of objects surviving 32 Kbytes of allocation and more than 10% still alive after 200 Kbytes.

The eleven benchmarks in the study of C/C++ programs by Detlefs et al. [DDZ94] have average object sizes between 15 and 249 bytes, with six programs over 30 bytes. In comparison, only one of our six benchmarks (compress) exceeds an average objects size of 30 bytes. Although [WJ+95] do not measure the average object size, they distinguish popular object sizes when presenting the memory profile for five C programs.

These programs, too, have larger object sizes than our Java examples with objects of several kilobytes being fairly common.

A few studies report numbers on type composition, such as Shaw for Lisp [Sha88] and Gonçalves for ML [Gon95]. Stefanovic and Moss analyze nursery survival rate depending on object kind in ML [SM94]. Since all three studies use different categories for types (e.g., Cons, Vect/String, Float, Object, Ratio, Symbol, and Fundef in Lisp), their numbers are difficult to compare to ours. However, it appears that the four Lisp programs analyzed in [Sha88] allocate less space for arrays than typical Java applications (2%-48%). This effect seems even more pronounced in ML; none of the 10 benchmarks studied in [Gon95] uses more than 1% for arrays. In addition, three of them use over 30% of their space for reference-free objects (real).

7. Conclusions

We have analyzed the memory allocation behavior of six large Java programs from the SPECjvm98 benchmark suite. All applications allocate significant amounts of memory (between 60Mbytes and 230Mbytes) but can run with a minimal heap size of less than 8Mbytes. Our observations confirm the weak generational hypothesis that young objects are most likely to die, although the result is less pronounced than in other languages—after one megabyte of allocation, up to 21% of all Java objects are still live.

Regarding heap composition, we found that both arrays and instance objects contain a high fraction of non-reference fields. However, only one application (mtrt) allocates a considerable number of instance objects containing no references except for the class pointer. When analyzing the heap for object types, we found that one kind of object (e.g., instance objects, or byte arrays) often dominates allocation, but none dominates all applications.

Eight-byte alignment, a technique used in many real JVM implementations, can increase the allocation rate by up to 19%, with a median of 5%. Extra header words (on top of the class pointer) are even costlier: given average object sizes around 20 bytes, a single extra word would increase the allocation rate by about 20%.

Despite the fact that all but one of our test programs are single-threaded, we were surprised to see that most of the allocated data (>99%) is strictly thread-local. (We consider an object thread-local if no other thread can access it.) Those objects that become global are very likely to survive for a long time.

We hope that this information will prove useful to JVM implementors when optimizing Java garbage collection and memory allocation.

Acknowledgments

We thank Steffen Grarup, Mark Reinhold, and the members of the OOCSB group at UCSB for valuable comments on earlier versions of this paper. We also thank Derek White who introduced us to the idea of thread local heaps for Java.. This work was funded in part by Sun Microsystems, the State of California MICRO program, and by the National Science Foundation under CAREER grant CCR96-24458.

References

[Bak93] Henry Baker. 'Infant mortality' and generational garbage collection. In *ACM Sigplan Notices*, volume 28(4) of ACM Sigplan Notice, April 1993, pages 55-57.

[Bak94] Henry Baker. The thermodynamics of garbage collection. In *ACM Sigplan Notices*, volume 29(4) of ACM Sigplan Notice, April 1994, pages 58-63.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation,* volume 28(6) of ACM SIGPLAN Notices, Albuquerque, NM, June 1993, ACM Press, pages 187-196.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications,* volume 21(11) of ACM SIGPLAN Notices, Portland, OR, October 1986, ACM Press, pages 119-130.

[DDZ94] David L. Detlefs, Al Dosser, and Benjamin G. Zorn. Memory allocation costs in large C and C++ programs. In *Software Practice and Experience*, 24(6), 1994.

[GA95] Ran Giladi and Niv Ahituv. SPEC as a performance evaluation measure. In *Computer*, volume 28(8), August 1995. p.33-42.

[Gon95] Marcelo José de Rezende Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection.* Ph.D. thesis, Princeton University, May 1995. Technical Report CS-TR-492-95.

[GJS96] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. Addison-Wesley, 1996.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *Proceedings of OOPSLA'91 Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of ACM SIGPLAN Notices, Phoenix, Arizona, October 1991. ACM Press, pages 33-46.

[Hay93] Barry Hayes. *Key Objects in Garbage Collection.* Ph.D thesis, Stanford University, March 1993.

[HH+98] Michael W. Hicks, Luke Hornof, Jonathan T. Moore, and Scott M. Nettles. A study of Large Object Spaces. In *Proceedings of the First International Symposium on Memory Management,* Vancouver, October 1998, ACM Press, pages 138-145.

[HMN97] Michael W. Hicks, Jonathan T. Moore, and Scott M. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of International Conference on Functional Programming,* Amsterdam, June 1997.

[Hol98] Allen Holub. Programming Java threads in the real world, part 1. In *JavaWorld*, September 1998. http://www.javaworld.com/javaworld/jw-09-1998/jw-09-threads.html

[JL96] Richard Jones and Rafael Lins. *Garbage Collection.* John Wiley & Sons, 1996.

[LH83] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. In *Communications of the ACM*, 26(6):419-29,1983.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[MHN97] Jonathan T. Moore, Michael W. Hicks, and Scott M. Nettles. Oscar: A GC testbed. Position paper for *OOPSLA'97 Workshop on Garbage Collection and Memory Management.* Atlanta, GA, October 97.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* Ph.D thesis, Stanford University, 1988. Technical Report CSL-TR-88-351.

[SPEC98] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation, Release 1.0.* August 1998. Online version at http://www.spec.org/osg/jvm98/jvm98/doc/index.html

[SM94] Darko Stefanovic and J. Eliot B. Moss. Characterization of object behavior in Standard ML of New Jersey. In *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming*. ACM Press, June 1994.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *ACM Software Eng. Notes/SIGPLAN Notices Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, volume 19(5) of ACM SIGPLAN Notices, April 1984.

[Ung86] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* ACM Distinguished Dissertations. MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[Ung86] Paul R. Wilson. Uniprocessor garbage collection. In *Proceedings of International Workshop on Memory Management (IWMM'92), volume 637 of* Lecture Notes in Computer Science, Springer Verlag, 1992.

[WJ+95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, volume 986 of Lecture Notes in Computer Science, Kinross, Scotland, September 1995. Springer-Verlag.

[Zor89] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. Ph.D. thesis, University of California at Berkeley, March 1989

[ZG92] Benjamin G. Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, volume 27(12), pages 71-80, December 1992.

[ZG94] Benjamin G. Zorn and Dirk Grunwald. Evaluating models of memory allocation. In *ACM Transactions on Modeling and Computer Simulation*, 4(1), 1994.

[ZS98] Benjamin G. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.