The Causes of Bloat, The Limits of Health

Nick Mitchell Gary Sevitsky
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532 USA
{nickm,sevitsky}@us.ibm.com

Abstract

Applications often have large runtime memory requirements. In some cases, large memory footprint helps accomplish an important functional, performance, or engineering requirement. A large cache, for example, may ameliorate a pernicious performance problem. In general, however, finding a good balance between memory consumption and other requirements is quite challenging. To do so, the development team must distinguish effective from excessive use of memory.

We introduce *health signatures* to enable these distinctions. Using data from dozens of applications and benchmarks, we show that they provide concise and application-neutral summaries of footprint. We show how to use them to form value judgments about whether a design or implementation choice is good or bad. We show how being independent of any application eases comparison across disparate implementations. We demonstrate the *asymptotic* nature of memory health: certain designs are limited in the health they can achieve, no matter how much the data size scales up. Finally, we show how to use health signatures to automatically generate formulas that predict this asymptotic behavior, and show how they enable powerful limit studies on memory health.

Categories and Subject Descriptors D.2.8 Metrics [*performance measures*]

General Terms Memory Footprint, Data Structure Design, Characterization

Keywords bloat, memory footprint, metrics, limit studies

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

1. Introduction

It is fairly easy these days to design and implement a data model in a way that consumes a large amount of memory. For example, we have seen Java server applications that require a gigabyte of memory to support a few thousand users. That a data structure is big is of course a sign for concern. We propose that the *health* of a data structure's use of memory depends not so much on its size, but rather on the relationship between actual data and structural overhead.

The makeup of memory depends on the design of data types, and the way instances of these types are glued together into collections [19, 20]. Consider an example data model with entities E_1, E_2, and E_3; each instance of E_3 has a single byte field of actual data, each E_2 contains one or more E_3 instances, and each E_1 contains one or more E_2 instances. Developers must choose how to map this model to a physical implementation. In an object oriented language, one typically maps each entity to a class, and uses a standard collection, such as the Java LinkedList, to store the multi-valued relationships. Unfortunately, this implementation results in an extremely unhealthy heap. With fan-out of 1000 at each level, this implementation's fraction of actual data is only 10%. Furthermore, as Section 4 shows, the fraction of actual data *can never be better than 10%* — no matter how many instances of E_2 or E_3 there are to amortize bookkeeping overheads.

In this paper, we first introduce a scheme for exposing the underlying causes of such poor health. Second, we leverage this scheme to introduce a way to determine whether a data design will ever be good.

The Causes of Bloat Depending on the data type design, each instance will use its bytes for a variety of purposes. Section 2.1 introduces an *instance health* categorization system that divides the bytes of each instance into one of four *application-neutral* [18] categories: primitive data, header bytes, pointers, and null pointers. For example, each instance of E_3 above has only a single byte of primitive data, but 12 bytes of JVM-imposed object header.

Depending on the collection design, each instance will serve one of a number of purposes. For example, each in-

stance of LinkedList serves as a wrappers to the collection as a whole. A consequence of this design is that, when LinkedLists are nested [2], the wrapper cost is magnified. Section 2.2 introduces a *collection health* categorization system that divides instances into one of four application-neutral categories based on the role they play in collections.

Combining these two classification schemes yields deeper insight into whether bytes are serving a useful purpose. For example, when we contrast the two primitive fields in each LinkedList instance, used for bookkeeping purposes, with the primitive byte field in E_3, we see that not all primitive data should be judged equally. The role of an object can give valuable clues into the purpose of its bytes. Section 2.3 introduces the *health signature*, the composition of the collection and instance health categorizations; it is formed by intersecting the two categorizations. We present health signatures from a variety of applications and benchmarks.

To tie questions about features of a design to their memory consequences, Section 3 introduces *health judgment schemes*. A judgment scheme maps a set of features to the design's health signature. Features of interest will vary depending on the analysis. We introduce two schemes. The *overhead* judgment scheme highlights, as features of the design, the various ways overhead is introduced. For example, we can analyze the amount of pointer overhead that a design incurs. Section 3.1 shows that a HashMap of 2-characters Strings will devote 29% of its space to pointer overhead.

The *scaling* judgment scheme is focused on collections, and the fixed and per-element costs they incur. The memory impact of collection choices can be difficult to predict, particularly when using standard collections whose implementations are hidden. The scaling judgment scheme helps to surface these costs. For example, Section 3.3 shows a real application that devotes a surprising 74% of its memory to collection fixed and per-element costs.

Another benefit of judgment schemes is as a concise health summary. We present these summaries for dozens of applications, to illustrate the great variety of health characteristics. The application neutrality of the health signature enables comparison of these disparate applications. In addition, Section 3.3 provides a detailed example of how each scheme offers distinct insights into a real-world memory problem.

The Limits of Health From a health judgment, we can derive a health metric: the ratio of actual data to the total. We have observed that this ratio has asymptotic behavior. For example, our LinkedList of LinkedLists of E_3 objects will never contain more than 10% actual data; the application discussed in Section 3.3 can never contain more than 17%. Section 4 demonstrates this behavior, and presents a way to derive *scaling formulas* that predict it. We show how the scaling properties of a nested data structure depend upon the structure of that nesting. We introduce the *content schematic*, a concise summary of this nesting structure. The content schematic of a data structure identifies the distinct contexts in a data structure that may vary in size as the application runs. We show how to combine a scaling judgment scheme with the content schematic in order to automatically derive formulas. We present several example studies to demonstrate the power of scaling formulas in pinpointing the limits of health for a given design.

Summary This paper presents the following contributions:

- two application-neutral systems that classify the role of bytes in objects, and the role of objects in collections
- health signatures that distinguish the role of bytes based on the role of objects in collections
- a judgment scheme to distinguish the sources of overhead in a design, and a second scheme to expose the design's scaling properties
- a technique for finding and summarizing the data structures in a heap snapshot into *content schematics*
- an algorithm for automatically constructing scaling formulas to gauge the asymptotic health of a data structure

2. The Health Signature

We summarize a heap snapshot, or a subset of a snapshot, to identify potential inefficiencies in the way memory is structured. A *health signature* is a two-dimensional summary using two distinct categorizations of memory:

- **instance health**: for each object, categorize its bytes according to the function those bytes serve.
- **collection health**: for each data collection, categorize its constituent objects according to the function each serves in implementing the collection's functionality.

In both cases, the categories are application-neutral, in the sense that they are not in terms of data types, or any other such application-specific artifacts. We begin by introducing the categories, and then show how to derive health signatures from them.

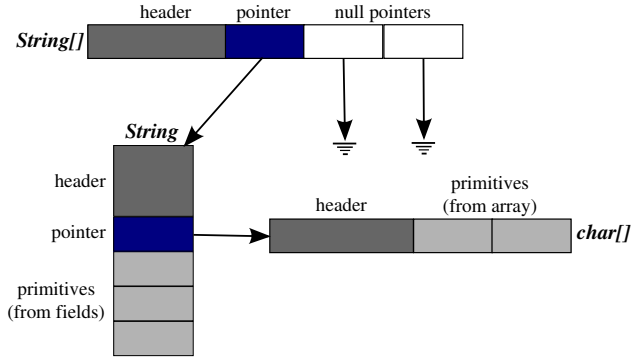
2.1 Instance Health

Every instance of a data type consumes a certain number of bytes in the runtime heap. Some of these bytes store the type's instance variables, while some store information needed by the underlying runtime, such as for garbage collection or lightweight synchronization. We categorize the bytes of an object in this way: by what purpose those bytes implement or facilitate.

Table 1 shows the four categories of instance health. The first category covers all primitive data, whether from primitive fields of objects, or from arrays of primitive data. In Java, each primitive element consumes one, two, four, or eight bytes. The second category accounts for the bookkeeping information that the runtime sets aside to help manage each object, commonly termed "object header". The runtime

category	explanation	# bytes
primitive	prim. array elements, prim. fields	1–8
header	space imposed by VM	12–16
pointer	references between objects	4
null	unused pointers	4

Table 1. The categories of instance health, with sizes for a typical 32-bit Java VM.



(a) An data structure composed of an array of strings, showing one string with its three primitive fields and that string's 2-element character array.

primitive	header	pointer	null	**total**
16	40	8	8	72

(b) The instance health categorization of this data structure, with category sizes in bytes.

Figure 1. An example of instance health.

uses this header to facilitate tasks such as garbage collection, lightweight synchronization, and reflection. The size of each object header is usually independent of the instances type; many Java virtual machines impose a 12- or 20-byte object header, depending on whether it uses 32- or 64-bit addressing. The header of array instances differs from that of object instances. For example, in Java, it is common to have a twelve-byte object header, with an additional 1–4 bytes to keep track of an array's length.[1]

The final two categories cover bytes set aside for pointers between objects. Each pointer slot, whether from a field of an object or from an array of pointers, consumes one word (either 32 or 64 bits on most contemporary virtual machines). This is the same even if the pointer is null. Null array slots occur in the common case of an application that allocates an array with some default capacity, but only makes use of a smaller number of elements during the course of its run. The third category includes bytes from pointer slots that refer to extant objects, and the fourth category includes the null case.

[1] There are two other cases of per-instance variation of object header size: fragmentation due to object alignment, and hashcodes stored in object headers. This information is available only in some JVMs. In cases where we do have it we include it in the object header category.

category	explanation	example
head	head of a collection	HashMap, String
array	array backbone	HashMap$Entry[]
entry	list-style element	HashMap$Entry
contained	anything else	char[]

Table 2. The categories of collection health.

Figure 1 illustrates an instance health categorization. The figure shows an array of Java String objects, the one String in that array, and the character array object which underlies that String. On a 32-bit platform with 12-byte object headers and 8-byte alignment, this example will consume a total of 72 bytes: 40 from object headers, 8 from pointers, 8 from null pointers, and only 16 from primitive data. Furthermore, of those 16 bytes, only 4 come from the true data of this structure: the two characters in the String's character array. To come to this conclusion requires a categorization of objects that distinguishes the String from its character array.

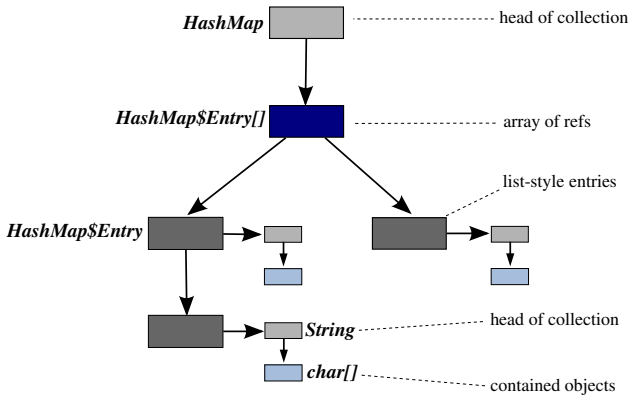
2.2 Collection Health

We now address sources of bloat that stem from the way individual objects are glued together into larger collections. Inside every collection will of course be the actual data intended to be collected together. Unfortunately, the actual data may only occupy a small fraction of the collection's total size. Everything else represents the infrastructure costs. This dichotomy, between actual data and collection infrastructure, is the basis for our second categorization scheme. Whereas the instance health breaks down the bytes of each instance, the collection health analysis categorizes the objects of a collection.

Each instance inside a collection serves one of four primary roles [15]. Table 2 shows these four categories of collection health. First, every collection consists of a number of "backbones" that allow the collection to contain a variable number of objects. Some backbones are `array`s, which are commonly used when append-only random access is needed. Other backbones have a chained or recursive structure, to allow for efficient random insertion and deletion. We label these as `entry` instances. For example, in the Java standard libraries, instances of the data type LinkedList$Entry fall into this category.

Next, many data structure implementations include a data type that represents the collection as a whole; in the Java standard libraries, such data types include HashSet and LinkedList. In fact, every implementation in the Java collections library was designed so as to devote one instance to serve this role. The `head` category includes these heads of collections. In general, we include in this category any instance that points to an `array` or `entry` instance. The final category, `contained`, includes what remains; it includes the non-infrastructure instances inside collections.[2]

[2] It also includes the relatively few instances outside any collection.



(a) A data structure composed of a hash set containing three strings, and showing each string with its 2-entry character array.

contained	head	array	entry	**total**
48	144	76	96	364

(b) The collection health categorization of this data structure, with category sizes in bytes.

Figure 2. An example of collection health.

Figure 2 shows an example collection health categorization. This heap contains a HashMap whose keys and values are Java Strings, and the Strings in turn contain character arrays. We consider the case where the key and value point to the same object,[3] and show only a single edge to the Strings, to reduce clutter in the figure. The top-most node in the figure, representing the HashMap , falls into the `head` category. The Strings also fall into the `head` category, since they refer to the arrays of primitive characters. Using instance sizing information from IBM Java 1.5, this heap consumes a total of 364 bytes. Of that, 144 bytes, 40% of the total, belongs to the `head` category: three strings, each 32 bytes, and one hash map; 76 bytes belongs to the `array` category, from the four-element array; 96 bytes, 26% of the total, belong to the `entry` category: three hash map entry objects; and 48 bytes, only 13%, belong to the `contained` category: three 2-element primitive arrays.

2.3 Composing the Two Classifications

The composition of instance health and collection health lets us study the role of an instance's bytes in the context of a collection. For example, bytes that seem reasonable (e.g. primitive data) may come mostly from objects that serve an infrastructure role, and that one did not realize would incur such a great memory overhead. Conversely, bytes that seem to be excessive (e.g. pointer bytes) may come from infrastructure objects that enable a useful function (e.g. offering random insertion and deletion).

With four categories in each categorization system, a health signature is a 4×4 matrix. Each entry is the number of bytes in the intersection of a collection health category and an instance health category.

[3] This is a common situation used to pool objects, since the Java HashSet does not support a `lookup()` method.

	primitive	header	pointer	null	**total**
contained	12	36	0	0	48
head	56	60	16	12	144
array	0	12	8	56	76
entry	12	48	16	20	96
total	80	156	40	88	364

Table 3. The health signature for the example in Figure 2.

In Table 3 we show the health signature for the data structure in Figure 2. The leftmost column shows the distribution of primitive bytes, which at the surface we might consider to be the "actual data" of the data structure. We see, however, that most of the primitive bytes are found in `head` objects. Each String in Java contains three primitive fields for bookkeeping purposes: a memo of the string's hash code, the string's length, and an offset, in case the underlying characters are shared. Each HashMap, also a `head` instance, devotes five primitive fields to bookkeeping. The only actual data in this data structure is found in the String's character array, the 12 bytes shown in the `contained-primitive` cell. Similarly, each HashMap has four pointer fields, three of which are null by default. These are used for caching in cases which may not occur. We contrast them with the 56 bytes in the `array-null` category, a sign that an array may have been sized too large. By taking into account the context in which bytes are used, the health signature allows us to make these finer distinctions about the ways a design spends its bytes.

Figure 3 presents health signatures from four snapshots, and illustrates the very different signatures that show up in practice. Figure 3(a) shows the health signature of a DaCapo benchmark called antlr. The signature indicates that antlr devotes 70% of its bytes to primitive bytes within contained objects; the number of bytes spent in pointers, arrays and list-style entries are all small fractions of the total heap size. Figure 3(b) shows the very different health signature of javac, a benchmark from the SPECJVM98 suite. In contrast to antlr, javac spends only 20% of its bytes in the primitive-contained intersection. In addition, object headers and pointers show up in much greater concentration. The health signature of Figure 3(c) comes from application S. Its health signature seems to indicate some severe problems with memory health. In particular, contained objects do not consume a plurality of the heap, and primitive data appears mostly in heads of collections. The fourth snapshot, from a J2EE [22] server application, has characteristics in common with each of the other three applications.

2.4 Implementation Details

To automatically categorize by instance or collection health, our implementation analyzes heap snapshots. To collect

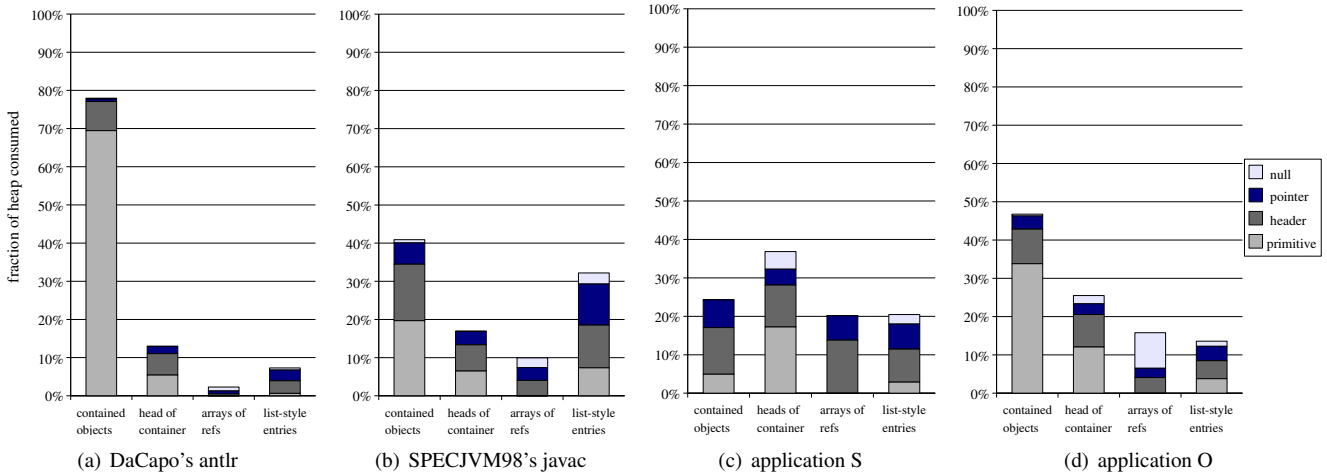


Figure 3. Four example health signatures.

snapshots, we used the built-in facilities of Java virtual machines (JVM) to trigger writing a snapshot to disk. In some cases, the JVM produced a snapshot upon heap exhaustion. In other cases, we explicitly requested them.

2.4.1 Instance Health Implementation

Membership of bytes in a category is a local property of each object. Therefore, categorizing bytes is mostly a straightforward task: for each instance, count up the header and pointer bytes, and consider primitive bytes to be the instance's total size minus that sum. Depending on the richness of information provided by the heap snapshot format, however, there may be several wrinkles.

First, it may not be possible to distinguish null field slots from primitive data bytes with certainty. For example, most heap snapshot formats do not describe the field layout; rather, they specify the size of each type's instances, and, for each instance, its number of outgoing non-null references. In this situation, we conservatively estimate the number of null references by computing the maximum number of non-null references, over all instances of each type.

Second, it is usually infeasible to detect uninitialized primitive data. Therefore, the numbers reported in this paper are based on the assumption that the bytes of a primitive array always contain meaningful data.

2.4.2 Collection Health Implementation

The heap snapshot is a graph, where nodes correspond to objects and edges correspond to references between objects. We first compute a spanning forest over the graph. The forest we use is based on the dominator relation [15] because it eliminates back edges in a reliable way — one that does not depend on an arbitrary ordering of graph roots. We categorize data types based on a purely structural analysis of the spanning forest (so that "refers to" below means in the spanning forest). From the categorization of types, it is trivial to categorize objects, based on their type.

Every array of reference type belongs to the `array` category. Now consider a type having an instance that immediately refers to a different instance of that type. All instances of such types belong to the `entry` category. This strategy under-approximates the set of all list-style and recursive data types. Computing this property in general from only a structural analysis of the graph is a challenge for future research. Then, we categorize as `head` any type with an instance that immediately points to to an object whose type is either categorized `array`, `entry`, or is a primitive array. Any remaining data types are considered to be `contained`.

This implementation maps every object to a single category, except in the case of nested collections. Whenever a collection contains other collections, each collection `head` is also a `contained` object. For this paper, we prioritize the categories, to ensure that each object falls into one category: we give priority to the `array` and `entry` categories, followed by `head`, and then `contained` objects. In this way, all of the LinkedList objects in a list of lists will be considered to be heads of collections.

3. Judgment Schemes

For a developer experienced with a code base and with memory health analysis, a health signature can provide a necessary level of detail for problem resolution. For developers new to the code or to memory analysis, a summary that incorporates some amount of a priori judgment may be necessary. A health signature imposes no value judgments on whether an application's use of memory is good or bad. Moreover, a health signature is a two-dimensional summary. A one-dimensional summary can make deviations from expected norms quickly apparent. It can also enable bulk comparisons across versions, data structures, diverse applications, and varying load scenarios.

	primitive	header	pointer	null
contained	data	small objects		pointer overhead
head	primitive overhead		glue	
array				
entry	data			

Table 4. The overhead health judgment scheme.

overhead judgment	size
data	24
primitive overhead	56
small objects	156
pointer overhead	104
collection glue	24
total	364

Table 5. The overhead judgment scheme applied to the example in Figure 2.

In this section we introduce *health judgment schemes*, one-dimensional summaries of memory health signatures. Health judgment schemes can be designed for specific analysis tasks, combining elements in health signatures to bring out certain features. We demonstrate two health judgment schemes. An *overhead judgment scheme*, described in Section 3.1, highlights common ways in which overhead is introduced into a design. A *scaling judgment scheme*, described in Section 3.2, can help evaluate collection choices and can expose how a data structure will scale. In Section 3.3 we apply these two judgment schemes to a real-world memory problem.

3.1 The Overhead Judgment Scheme

We introduce the *overhead judgment scheme* to help distinguish the data in a data structure from the structure's overhead. We consider the ways in which space for primitive fields, object headers, and pointers may be introduced into a design. Table 4 shows how this scheme judges the categories from a health signature.

A design may use primitive fields to store actual data or bookkeeping information. As we saw earlier with String, we can use the role an object plays in a collection to help distinguish the purpose of its primitive fields. The overhead judgment scheme classifies `contained-primitive` bytes, such as the characters in a String's character array, as *data*; it classifies `head-primitive` bytes, like those in the String proper, as *primitive overhead*. We also consider `entry-primitive` bytes as data, rather than primitive overhead. We do this so as not to penalize applications that combine `entry` and `contained` functionality into a single type.[4]

Object headers and some pointers are introduced during class design when deciding which fields to include in a class, where to use subclassing, and where to delegate fields to separate classes. We classify bytes spent on object `header` as an indicator of *small objects* – instances with few fields, or arrays with few elements. We classify `pointer` and `null` bytes as *pointer overhead*, an infrastructure cost of a highly delegated or highly interconnected design. Pointers may also be employed for a different purpose: to maintain one-to-many relationships. We make this distinction by classifying non-null `pointer` bytes in `array` or `entry` objects as *collection glue*. We classify `null` pointers from `array` and `entry` objects, however, as pointer overhead; they are the overhead of oversized arrays or very short linked lists.

Table 5 shows the overhead judgment scheme applied to the data structure in Figure 2. The large values for small objects and pointer overhead highlight the degree of delegation in this design.

Figure 4 shows the application of this scheme to a diverse assortment of snapshots. Table 6 lists the applications in our corpus. We study snapshots from a large number of real applications: under-test and deployed servers, and standalone and Eclipse-based [12] clients. These heap snapshots range in size from 50 megabytes to 2 gigabytes; the largest snapshots contain as many as 20-40 million live objects.

Our corpus also includes snapshots from two benchmark suites, SPECJVM98 [24] and DaCapo [3] (using dacapo-beta051009.jar). For each benchmark, we configure it to run its largest configuration, and acquire several dozen heap snapshots during the run. We generally pick the largest snapshot from among those collected.[5] It is worth noting that none of the benchmarks have heaps bigger than 10 megabytes. Most of the benchmark snapshots were considerably smaller than that.

3.2 The Scaling Judgment Scheme

The *scaling judgment scheme* distinguishes contained objects from collection implementations, exposing how each contributes to overhead. Table 7 shows how it is derived from the health signature. The primary distinction is between `contained` objects (the first row in the table) and `head`, `array`, and `entry` objects (the remaining rows). Within each we define two finer categories. For contained data, we label as *data* any primitive field data, just as we did in the overhead judgment scheme. We label the rest of the bytes – object headers and null and non-null pointers – as *data overhead*. For objects implementing collection infrastructure, we divide their bytes into *fixed* and *variable collection overhead*. Fixed collection overhead includes all of the bytes in head-of-collection objects, and the object header portion of arrays

[4] The Java LinkedList, Hashtable, etc. use distinct data types for these two roles; the GNU Trove [9] collection classes often do not. A collection health categorization that better distinguished these scenarios would allow for more precise health judgment for `entry-primitive` bytes.

[5] In cases where one snapshot was marginally smaller but considerably more unhealthy, we chose to present, in Figure 4, the unhealthy one.

application	description
A, P	Eclipse-based programs
C, G, H, I, L, M	financial services server
E, Q	catalog management server
B, N, F	Java clients
D, J, K, O, R	collaboration servers
S	standalone Java program
serverbench	application server benchmark
dacapo	the DaCapo benchmark suite
specjvm	the SPECJVM98 benchmark suite

Table 6. Our corpus of Java heap snapshots, taken from real applications and benchmarks. The real applications have 50MB to 2GB of live objects. The DaCapo and SPECJVM benchmarks are both relatively tiny in this regard.

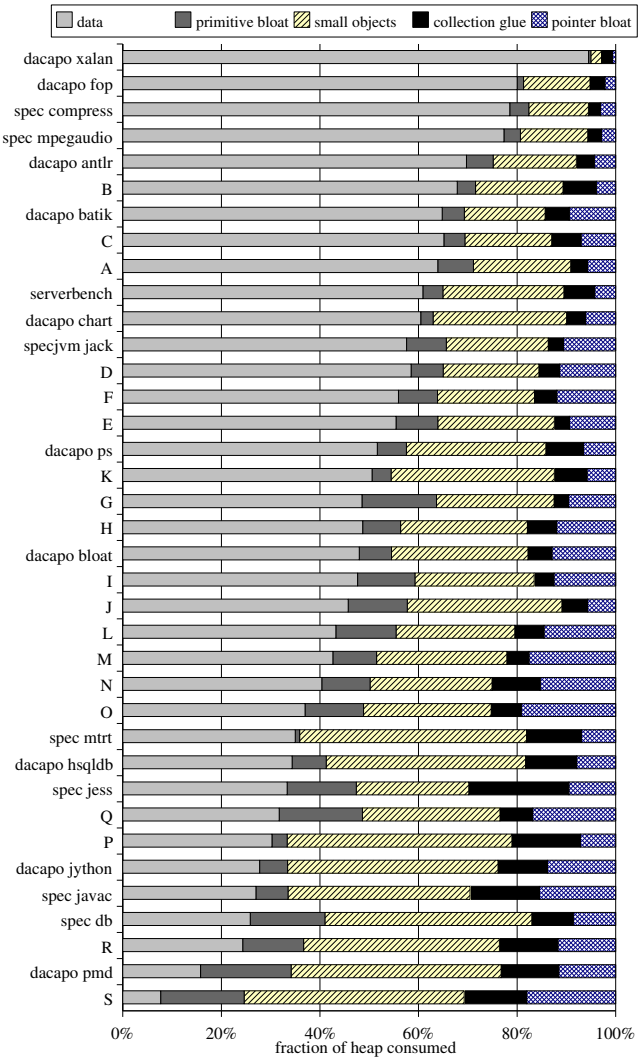


Figure 4. A judgment scheme, in this case the overhead judgment scheme, allows for quick comparison of the health of disparate applications. For example, application Q has 780M of live objects, of which only 234M is actual data.

	primitive	header	pointer	null
contained	data	data overhead		
head	fixed collection overhead			
array				
entry	variable collection overhead			

Table 7. The scaling health judgment scheme.

scaling judgment	size
data	12
data overhead	36
fixed collection overhead	156
variable collection overhead	160
total	364

Table 8. The scaling judgment scheme applied to the example in Figure 2.

of references. [6] Variable collection overhead includes bytes allocated for the elements of arrays of references, whether null or non-null, and all of the bytes of `entry` objects.

Table 8 shows the scaling judgment scheme applied to the example in Figure 2. This breakdown highlights the degree to which memory is devoted to collection overhead, both fixed and variable.

3.3 An Example: Applying Health Judgment Schemes

In this section we walk through an example from a real-world application with large footprint problems, to demonstrate the power of the two health judgment schemes. The example also illustrates the variety of memory problems seen in industrial applications. The application is a planning system, application S in Table 6. It was at the prototype stage of development, and as such, the primary concern was producing a working algorithm quickly. Like many industrial applications, even those much closer to production, memory usage was not taken into account until a problem appeared.

The primary data structure is a level graph where the same vertex may appear with different edges at multiple levels of the graph. We explore one particularly expensive data structure: the representation of the edges. Figure 7 shows a schematic view of that portion of the data, which cost 42MB in this run. Each node in the figure represents instances having the same ownership context; some intermediate classes have been elided (explained formally in Section 4.2). The node labels reflect the cost of all instances at that node, including those of elided classes. Edges are labeled with their average fan out. In the application's level graph, each edge is stored as an Edge object pointing to its source and target vertices. There are two indexes for looking up the edges for a (vertex, level) pair: one for in- and one for out-edges. The indexes are implemented as HashMaps (the left-most

[6] In Java there are no per-array primitive fields. If there were, they would be included in this category.

scaling judgment	outer HashMap	Arrays$ArrayList	HashSet
data	2.6M	0.3M	2.4M
data overhead	9.0M	0.8M	8.2M
fixed collection overhead	7.4M	3.0M	4.4M
variable collection overhead	24.5M	0.8M	21.2M
total	43.5M	4.9M	36.2M

Table 9. The scaling judgment scheme applied to a large data structure of application S. The size of each category represents the sum of bytes categorized that way for all objects under the given collection.

HashMaps in the figure). The key is a 2-element ArrayList, containing an Integer level number and a pointer to a vertex. The value is a HashSet of Edges. In this run, there were approximately 148,000 edges.

The first column of Table 9 shows the scaling judgment scheme applied to this data. We can see that a very small portion of the bytes, 6%, are devoted to actual data, while 74% are devoted to collections. While we would expect an index, especially one mapping each key to many values, to involve a significant amount of overhead due to collections, we would not expect it to so completely swamp the data. What is more telling is how much memory (7.4MB or 17%) is devoted to collection fixed overhead costs. According to the developer, this was a production-scale run, not a small test. We would expect the collection fixed costs to be amortized across a run with 148,000 edges. The fact that they are so large suggests that some of these costs are being magnified rather than amortized.

We next compute scaling judgments over the two subordinate data structures, first the keys and then the values, since each may have different characteristics. According to the developer, the Arrays$ArrayList class was chosen not because a one-to-many relationship was needed, but rather for expediency of coding, since it enabled a coding idiom where test cases with constant vertex/values pairs could be coded in a single line. The second column in Table 9 shows the cost of using a collection class for this purpose: high fixed and variable collection overhead costs. The high fixed component (73% of the keys' cost) is due to requiring two instances, an ArrayList and array, to store just two elements. The use of an ArrayList also required the programmer to box the level number into an Integer, adding to the data overhead and variable collection costs.

The overhead judgment scheme can provide another view into this design, shedding light on overall overhead costs across contained and collection data. The first column in Table 10 illustrates this analysis for the keys. The high value for small objects is a consequence of the two levels of delegation, from Arrays$ArrayList to Object array to Integer. The developer can easily replace this design with a single class: Pair {int level; Object vertex;}. The second column in Table 10 shows the reduction that would be achieved.

The third column of Table 9 shows the scaling judgment over the values: the HashSets and the Edges they

overhead judgment	(current) Arrays$ArrayList	(proposed) Pair
data	0.3M	0.3M
primitive overhead	0.3M	0M
small objects	3.2M	1.1M
pointer overhead	0.3M	0.3M
collection glue	0.8M	0M
total	4.9M	1.7M

Table 10. The overhead judgment scheme helps to compare current and proposed implementations of a HashMap's keys in a large data structure of application S.

hold. Notice that the majority of the cost is due to collection overhead here as well, with significant fixed and variable components. The standard Java HashSet, which stores a set of unique values, is implemented by delegating to the more general HashMap, which in turn points to an array of HashMap$Entry objects that point to keys and values, and to each other.[7] The delegation of HashSet to HashMap raises the fixed collection overhead for each of these nested HashMaps; the overgenerality of HashMap raises both the fixed and variable overhead costs: the HashMap has bookkeeping fields that are not relevant to this use, and all of the inner HashMap$Entry objects point to the same (unused) singleton value. Note that overall, a HashMap$Entry, containing four fields, is larger than the Edge object that it is indexing. This is one of the reasons behind the high variable collection overhead cost in our data structure. Note also that the fixed collection overhead cost is high because each HashMap holds only 4–5 elements on average.

The choice of ArrayList seemed to the developer like an inappropriate choice once the costs were known. The choice of HashSet seemed more reasonable for the intended use. The developer wanted the code to be safe by guaranteeing uniqueness, and assumed that the widely-used standard Java HashSet would be well optimized. One can easily imagine refactored HashSet and HashMap classes that shared code while optimizing for these two different uses.

[7] We use framework knowledge to classify the HashSet as head of collection, though it would not normally be classified as such solely on the basis of structure.

4. Asymptotic Analysis

Ideally, health should improve as input size increases, because fixed infrastructure costs are amortized over increasingly large amounts of data. Unfortunately, with multiple data structures and nested collections, this may not be the case. In many situations, the fraction of actual data in an application reaches an asymptotic value. In this section, we first present a simple example that illustrates the asymptotic behavior of memory health. To model this behavior, we introduce a definition of a data structure and a summary of its internal structure. We then provide an algorithm for automatically constructing formulas, on a per-data structure basis. Each formula predicts how a data structure's memory health changes as the sizes of its components vary. Finally, we give examples of scaling studies that are enabled by having these formulas, and demonstrate them on an application from our test suite.

4.1 Observations of Asymptotic Behavior

Consider a linked list that stores data structures each of which has a collection health categorization of 4 bytes of `contained`, 24 bytes of `head`, and 8 bytes of `entry`. The linked list itself imposes 4 bytes of `entry` per contained structure and 12 bytes of `head`. We focus on these categories to keep this motivating example simple. With one contained structure, the relevant portion of the collection health categorization, (`contained`, `head`, `entry`), will be $(4, 36, 12)$; with two contained structures $(8, 60, 20)$; with n entries, $(4n, 24n + 12, 8n + 4)$. In absolute terms, all three categories approach infinity. However, any one category, relative to the whole, approaches an asymptotic value. When we normalize, we get $(8\%, 69\%, 23\%)$ with one entry, and $(9\%, 68\%, 22\%)$ with two entries. In the limit of large n, this will approach $(11\%, 67\%, 22\%)$.

OBSERVATION 1. *The ratio of any health signature category to the sum of all health signature categories approaches an asymptotic value. This value is governed by the health of the collections and contained structures.*

To illustrate asymptotic behavior in more depth, we experiment with three variations of a collection of collections of primitive bytes (recall this example from the introduction). The first uses the standard Java LinkedList collection for both the outer and the inner collections, and a distinct data type T to store the primitive byte field: a LinkedList of LinkedLists of instances of T. The second has the same shape, but replaces the Java list implementation with one from GNU Trove [9]. Trove avoids the `entry` costs, by requiring that T implement the next and previous pointers itself. The third uses an array of primitive byte arrays.[8]

Figure 5 shows the asymptotic behavior of these three implementations, as the size of the inner and outer collections vary. We use the overhead judgment scheme to categorize the bytes of each scenario. The x-axis of each chart plots an increasing size of the outer collection; the four columns of charts show the behavior of an inner collection size of, in order from left to right, 10, 10^2, 10^3, and 10^4. We have annotated the fraction of `data` in each chart. For example, the standard Java LinkedList approaches 8–10% `data` and the Trove implementation approaches 8–17% `data`. Because the design of these structures is so poor, Trove improves health by only a small amount. In contrast, the array of primitive byte arrays scales in a much more healthy way. The asymptotic health ranges from 34% to 98%. In addition, with large enough inner collections, the health *improves* as data size increases.

[8] A two-dimensional primitive byte array would further reduce bloat.

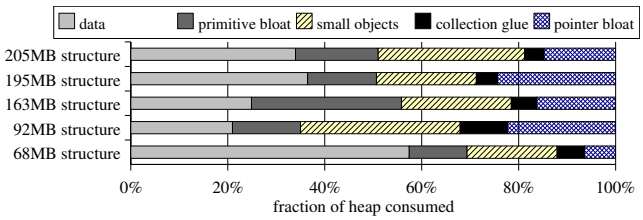


Figure 6. The overhead health judgments for the four largest data structures of application O

.

The health of the whole heap varies depending on how each data structure within the heap changes. The results shown in Figure 5 and in previous sections were the result of analyzing the heap as a whole. An unhealthy data structure may become more healthy as it increases in size. But, if a second, healthier data structure stays fixed in size, while an unhealthy one increases in size, the combined effect of the two will show the downwards trend, as in most of the experiments of Figure 5. In those examples, the JVM itself includes some unchanging data structures that, compared to the ones experimented with, are relatively healthy.

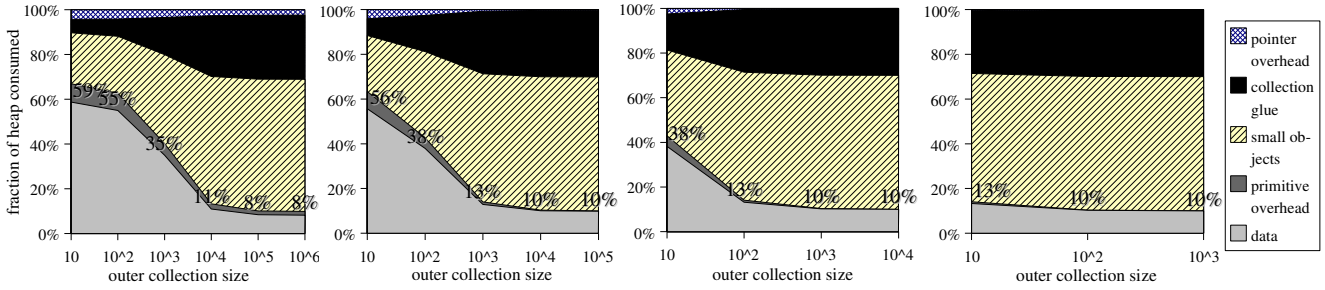
4.2 The Content Schematic

To study the asymptotic behavior of health, we must analyze at a finer granularity than the whole heap. We show how to decompose the heap into data structures, and how to arrange each data structure into a tree of regions that may possibly change in size. We refer to this data structure summary as a *content schematic*.

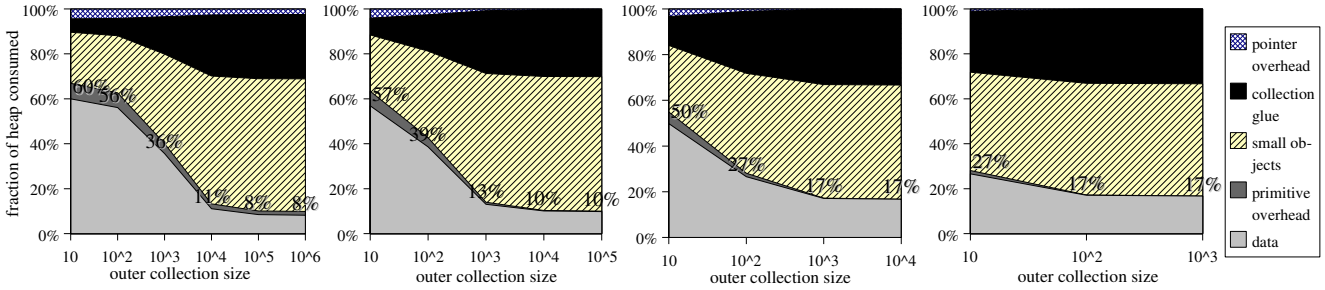
First, we define data structures by unique ownership, drawing on earlier work [15].

DEFINITION 1. *Given a heap snapshot considered as an object reference graph G, a* data structure of G rooted at type t *is the equivalence class of trees in the dominator spanning forest of G whose root node has type t.*[9]

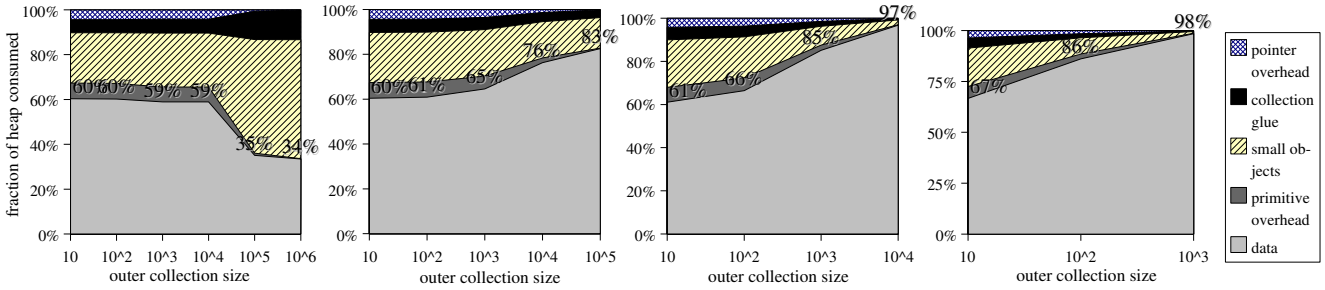
[9] In the case of diamond structures or root sharing, we refer the reader to [16] and [15]. For these situations, we chose to use the heuristic for handling shared ownership described in [16].



(a) Java's LinkedList of LinkedLists of Objects, each with a single byte field



(b) GNU Trove's TLinkedList of TLinkedLists of Objects, each with a single byte field



(c) Array of primitive byte arrays

Figure 5. The overhead judgment scheme illustrates the asymptotic behavior of memory health. The three rows show three implementations of a collection of collections of bytes. The four columns show inner collections of size 10, 100, 1000, and 1000. The x-axis of each plot corresponds to increasing the size of the outer collection.

Memory health varies widely across the data structures of an application. For example, Figure 6 shows the overhead judgment scheme applied to the four largest data structures of application O. Each has a distinct health signature.

To model the scaling properties of a data structure, we must identify its expansion points. To do so, we leverage the *backbone-folding* and *type-indistinguishable* equivalence relations introduced in [15] to group objects into "regions". The backbone-folding equivalence relation folds any `array` or `entry` object into the nearest `head`, and any `contained` object into the nearest parent `contained` object that is a child of an `array` or `entry` object. In this way, container-related infrastructure objects are grouped with the owning container, and the constituents of contained data structures are similarly unified with the head of contained structures. The type-indistinguishable relation creates a class for `head` objects that share the same path-to-root of `head` and `contained` data types; similarly, it creates a class for `contained` objects with the same paths-to-root.[10]

DEFINITION 2. *Given an object i that dominates a set of objects O, a* region of *i is the subset of objects in O equivalent according to the backbone-folding and type-indistinguishable equivalence relations. The* heads *of a region are those objects at the head of a chain of backbone folding. When we refer to the* region type *of a region, we mean the type of the head objects. Finally, the number of* elements *in a region is the number of head instances of that region.*

Under this definition, observe that a region is capable of changing in size if the region type of its parent region is

[10] The composition of these two equivalence relations is similar to the calling context tree [1], except in how it deals with recursive structures, and how it folds contained structures together.

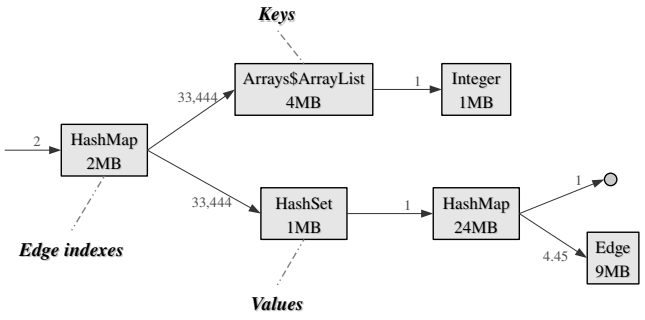


Figure 7. The largest regions in a 40MB data structure of application S. The average number of elements of each region is shown on the incoming edge label. Each region is labeled with the size of objects that fall into its equivalence class. Small regions are drawn as circles.

classified `head`. Furthermore, since we define regions over the dominator forest, the regions under any given object form a tree.

DEFINITION 3. *For a region r, the* parent $p(r)$ of r *is the region headed by objects that most closely dominate objects that point to the heads of r. The* average *number of elements of r is the ratio of the number of elements in r to the number of elements in $p(r)$. Given a data structure i, the* content schematic of i *is the tree of regions comprising i.*

For example, Figure 7 shows the content schematic for a subset of the largest data structure in application S (the same subset studied in Section 3.3). The figure labels each node with the region type of that region, and with the sum of the instance sizes of objects that fall into the equivalence class of that region. The figure labels an edge between regions r_1 and r_2 with the average number of elements in r_2.

4.3 Scaling Formulas

The health of a data structure depends upon the contributions of its constituent regions. To construct a formula that predicts its health, we first define the incremental contribution of every region to the cumulative health.

DEFINITION 4. *Given a region r, the* base health signature *of r is the health signature $\mathcal{H}$ computed over just those objects that belong to r. The* per-element base health signature *of r is its base health signature normalized by the average number elements of r.*

For every region in a data structure, we generate a formula that predicts its cumulative health — i.e. the health of that region and all its descendants. In this paper, we derive one formula for the `primitive-contained` category of the health signature, and a second formula for the other, infrastructural, categories. Observation 1 hypothesized that the ratio of either of these to the total has asymptotic behavior.

DEFINITION 5. *Given a region r, let the cumulative amount of actual data in r be D_r. Let the cumulative amount of overhead be J_r. We define the* scaling formula of r *to be:*

$$\mathcal{S}_r = \frac{J_r + D_r}{D_r} = 1 + \frac{J_r}{D_r}$$

Note that we have defined $\mathcal{S}$ to be the total number of bytes divided by the data bytes, rather than the other way around. We do this only because it leads to simpler formulas.

To explore the limits of memory health, and to make predictions of health under various conditions that might not have been captured by any dynamic run, we express the scaling formula symbolically. A symbolic scaling formula is a function of three main data structure properties: the data structure's nesting of regions, the health of collections, and the health of contained objects. Table 11 summarizes the independent variables that we consider. We now define these factors and show how to derive the set of symbols for a given region— i.e. the domain of the scaling formula $\mathcal{S}$. After deriving the set of unknowns, we then express D and J in terms of those unknowns.

The Expansion Factors of a Content Schematic The content schematic's nesting of regions in part governs how its health changes. For a given content schematic, we identify three ways in which the nesting structure governs changes in a data structure's health. We term these governors of size *expansion factors*.

First, as the application runs, the number of elements in each region may change. For example, this kind of expansion factor captures the number of distinct keys or values in a hash map. Observe that this quantity is the same as the average number of elements in a region. We term these *primary* expansion factors.

Second, certain collection implementations may overlap the variable collection overheads among the child regions. Consider the outer Java HashMap in Figure 7: it maps keys of type Arrays$ArrayList to values of type HashSet. The content schematic includes three regions, one for this outer HashMap, one for the keys, and one for the values. The HashMap$Entry objects of the outer HashMap region are variable collection overhead that is shared between the key and value regions. In some instances, the magnitude of the overlap for a region is a fixed constant times the sum of the primary expansion factors of its child regions. For example, in a Java HashMap with distinct and never-null key and values, the overlap factor will always be one half the sum of the child expansion factors: in this situation, a HashMap of size 100 will have 100 keys and 100 values. However, it is often important to consider this overlap has an expansion factor distinct from the primary expansion factors — e.g. when key and values overlap, or are sometimes null. We term these *overlap* expansion factors.

Finally, it is often desirable to study the effect of varying the size of primitive arrays; e.g. in an application that uses Java BigDecimals, or Java Strings, one may wish to observe the limiting effect of the primitive integer or primitive

symbol	meaning
n_r	average number of elements in region r
m_r	overlap of e for the children of region r
d_r	per-contained structure actual data in r
j_r	per-contained structure overhead in r
c_t	fixed collection overhead of container type t
e_t	amortized variable collection overhead of t

Table 11. The independent variables in a scaling formula In Section 4.4, we show how treating these factors as either unknowns or constants allows exploration of the asymptotic behavior of the scaling formula $\mathcal{S}$.

character array sizes. We term these *primitive* expansion factors.[11]

DEFINITION 6. *Consider a content schematic T with k regions. We denote the* primary expansion factors of T *by the symbols $n_1, \ldots, n_k$, and the* overlap expansion factors of T *by the symbols $m_1, \ldots, m_k$. An expansion factor may be left as an* unknown*, or it may be* hard-wired *to have a positive constant expansion factor.*

For example, the data structure shown in Figure 7 has six primary expansion factors. Three of these correspond to regions whose region heads are classified `head` (i.e. heads of collections): n_1 for the outer HashMap, n_2 for the HashSet, and n_3 for the Arrays$ArrayList. The scaling properties of the entire structure's health, and of each enclosed structures, depends in part upon these expansion factors. Consider the case of n_2: as it increases, the variable overhead of HashSet collections and the fixed overhead of the Edge data structures are magnified, but the fixed costs of the outer HashMap are amortized.

It is often helpful to use the properties of a particular snapshot to hard-wire the values of certain expansion factors. For example, given a heap snapshot, it is possible to hard-wire a primary expansion factor n_r to the average number of elements that belong to r in that heap snapshot. One can calculate the average or maximum size of primitive arrays in a certain region, and hard-wire that value for the primitive expansion factors within that region. Section 4.4 explores various scenarios of selectively hard-wiring expansion factors.

It is also possible to estimate a constant value for the overlap expansion factor of a region from a heap snapshot: find all `entry` objects in that region, and compute the number of graph edges v in the snapshot that point from one entry object to objects in more than one child region. In many cases, the maximum value of v is a reasonable choice for hard-wiring that region's overlap expansion factor.

[11] Our current implementation does not make primitive expansion factors explicit, but instead models primitive array growth by treating the d cost (described below) as an unknown.

type t	e_t	c_t
ArrayList	4	16
LinkedList	24	20
HashSet	28	52
TreeSet	36	52

(a) container costs

region r	j_r	d_r
Integer	12	4
Edge	24	8

(b) contained costs for regions head by the given type

Table 12. For data types inferred to be heads of collections or heads of contained structures, we can optionally hard-wire c, e, j, and d to the numeric values given by the per-element base scaling judgment scheme for those collection types. This table gives some example values, in bytes, from various Java standard collections and for the contained structures in the application from Figure 7.

The Effects of Collection Choices The scaling properties of a data structure also depend on the kinds of collections used, and how those collections scale. Each collection implementation incurs a fixed infrastructure cost that can be amortized across all elements in the collection, and an infrastructure cost for every contained data structure.

DEFINITION 7. *Given a region r with a head of container type t, the* fixed infrastructure cost *of r is denoted by c_t; when we write c_r, we mean this as shorthand for the numeric value 0, if the heads of r are not heads of containers, or c_t otherwise. Likewise, we define e_t and the shorthand e_r to be the* variable infrastructure cost*. We say that c_t or e_t is* hard-wired *if it is assigned a non-negative constant value.*

This definition assigns a single pair (c_t, e_t) to each collection type t, rather than one pair for each instance of a collection type. This accurately reflects the scaling behavior of most collections. However, in some cases there will be e costs that are amortized over a number of contained elements. Consider the case of a Java HashMap, which uses explicit chaining to handle hash collisions. For this collection implementation, the pointer bytes for the base array are amortized over all contained elements with the same hash code. We have chosen to simplify the presentation for this paper, and so do not account for these amortized e costs.

To compute c_t and e_t, we can leverage the scaling judgment scheme introduced in Section 3.2. First, pick an instance of collection type t, and compute its per-element base scaling judgment. Then set c_t to be the size of the fixed collection overhead category of the judgment; set e_t to be the size of the variable collection overhead category.

For example, we can account for the differences in the way a HashMap and a LinkedList scale. Table 12(a) provides values of c and e for four collection implementations from the standard Java library. These values were automatically generated using our implementation of the above algorithm. In generating a scaling formula, one may choose to hard-wire c and e to numeric values in this way, or may treat them as unknowns. The former analysis enables one to study the

$$D_r = d_r + \sum_{r' \in kids(r)} n_{r'} D_{r'}$$

$$J_r = c_r + j_r - m_r e_r + \sum_{r' \in kids(r)} n_{r'} \left(e_r + J_{r'}\right)$$

Figure 8. For a region r, the formulas for the cumulative actual data D_r and infrastructure overhead J_r.

scaling effects of a given set of collection choices, while the latter allows one to study the effect of swapping one collection implementation for another. In this paper, we focus on the former.

The Effects of Contained Structure Design Finally, the scaling formula of a region depends upon the health of the contained data structures.

DEFINITION 8. *Given a region r, we denote the per-element actual data to be d_r, and the per-element infrastructure cost to be j_r.*

Observe that the values of d_r and j_r can be computed just as we did for c_t and e_t. First, compute a per-element base scaling judgment for r. Then d_r and j_r are the values of the judgment from the data category and the data overhead categories, respectively. For example, Table 12(b) shows the values of d and j derived automatically for the application S. Each of the two rows corresponds to a region from Figure 7 whose region heads are classified as `contained`. The Integer object has the expected four bytes of data per element, plus twelve bytes of data overhead (since this analysis was performed on a snapshot from a JVM with twelve-byte object headers). The Edge object has proportionally higher costs of each: 8 and 24 bytes, respectively.

Scaling Formulas For a region r, the scaling formula $\mathcal{S}_r$ depends on the expansion factors, the c and e overheads, and on the d and j costs. We model the data and overhead D_r and J_r with the recursive formulas shown in Figure 8. The cumulative amount of actual data D_r in region r is its per-element actual data plus the sum of its children's actual data, weighted by their primary expansion factors. The cumulative amount of infrastructure in r, J_r, is given by its e overhead and j costs, plus the sum of its children's infrastructure costs, weighted by their primary expansion factors. The infrastructure of r also includes at most one e cost for every element in the child regions. This may over count, in the case of overlap. Therefore, the formula for J_r must subtract off $m_r e$, i.e. one e cost for every degree of variable-cost overlap among the child regions.

Table 13 shows four simple examples of scaling formulas, generated automatically by our implementation. The rows of the table show formulas for data structures that contain Strings in various ways. On its own, a String with d characters has a scaling formula of $1 + \frac{28}{d}$, while an n-element ArrayList of 2-element HashSets of Strings has a formula $1 + \frac{84}{d} + \frac{8}{nd}$.

4.4 Using Scaling Formulas for Limit Studies

Simple data structures have simple scaling formulas. In general, however, the formulas grow quite complicated. Even with simple cases such as shown in Table 13, the formulas alone are not the final story. Rather, they enable a user to perform a variety of asymptotic studies that illuminate the limits of health for a given design, or a given use case.

For example, a scaling formula of $\mathcal{S}_1 = 1 + \frac{28}{d}$, corresponding to the String data structure in Table 13, has the property $1 \leq \mathcal{S}_1 \leq 29$. This structure, on its own, is asymptotically perfectly healthy: longer and longer Strings become increasingly healthy until the ratio of total to actual data is 1. Compare this with an ArrayList of 2-element HashSets of 10-character Strings (note that in Java, each character consumes two bytes). In this case, $\mathcal{S}_2 = 5.2 + \frac{2}{5n}$, whose bounds are $5.2 \leq \mathcal{S}_2 \leq 5.6$. No matter how many entries in the ArrayList, the actual data will consume no more than $1/5.2 \approx 19\%$ of the heap. The health of this second case is bounded because of poor design choices made for enclosed structures: short Strings, and small HashSets. Hard-wiring the expansion factors of the inner regions to be small lets us explore the limits to which infrastructure costs can be amortized.

In any limit study, one must fix certain unknowns and let others vary. We experiment with the effect of scaling the big regions, and also of scaling the average d (the actual data in contained objects). In the former, we fix what's inside large collections, to explore how well health scales given those lower-level decisions. In the latter, we see how healthy the contained structures need to be to achieve good health overall, given a fixed nesting of collections.

To perform a limit study, first choose a region to analyze. Then, choose which factors to hard-wire, and which to leave as unknowns. In the case of a big-regions limit study, hard-wire the expansion factors of not-big regions contained under the chosen region to the average values observed in the heap snapshot used for analysis. Furthermore, hard-wire the d_r and j_r to the average values for region r, as described above. Use the recursive formulas shown in Figure 8 to generate $\mathcal{S}$. In our implementation, we then use a symbolic algebra system, Maxima [7], to simplify the formulas. We then use Maxima to compute limits. In some cases, the limit of $\mathcal{S}$ approaches 1 as the unknowns increase; this is usually the case when treating d as an unknown (with increasing actual data in one's data structures, the health increases monotonically). In this situation, we can use Maxima to solve for the value of d necessary to achieve good health, e.g. $\mathcal{S} < 1.2$ (which corresponds to a fraction of at least 80% actual data).

We first present results for some simple structures. The third and fourth columns of Table 13 give the results of these

data structure	scaling formula ($\mathcal{S}$)	vary n with $d = 20$	d necessary for $\mathcal{S} < 1.2$
String with d bytes of characters	$1 + \frac{28}{d}$	$\mathcal{S} = 2.4$	$d > 140$
n-elt. ArrayList of Strings	$1 + \frac{32}{d} + \frac{16}{nd}$	$2.6 \leq \mathcal{S} \leq 3.4$	$d > 160$
n-elt. HashSet of Strings	$1 + \frac{52}{d} + \frac{16}{nd}$	$3.7 \leq \mathcal{S} \leq 4.4$	$d > 270$
n-elt. ArrayList of 2-elt. HashSets of Strings	$1 + \frac{84}{d} + \frac{8}{nd}$	$5.2 \leq \mathcal{S} \leq 5.6$	$d > 420$

Table 13. Four example scaling formulas and two limit studies: for Strings with 20 bytes of characters the bounds of health that each structure can achieve, and the size of the Strings necessary in order to achieve good health ($\mathcal{S} < 1.2$).

two studies for our simple examples. Observe that a String must have at least 140 characters in order to achieve an $\mathcal{S} < 1.2$ (i.e. no more than 80% actual data). When Strings are placed in a standard Java HashSet, they must have at least 270 characters to achieve this level of health. On the flip side, placing 10-character Strings into a HashSet will result in an $\mathcal{S}$ of no less than 3.7 (i.e. no more than 27% actual data), no matter how many Strings are placed into the HashSet.

We now perform the two limit studies on the subset of application S analyzed in Section 3.3 (and visualized in Figure 7). We have omitted the complete formula; it has a large number of unknowns, and offers an unnecessary level of detail. Each limit study hard-wires many of the unknowns, and results in formulas that can be more easily studied. Table 14 shows the scaling formulas for the two limit studies. The first row shows the limit study which varies the number of keys and values. The formula has two unknowns: m and n, denoting the expansion factors of the two regions that were large in the given snapshot. These correspond to the HashSet and ArrayList nodes in Figure 7. This study shows that this subset of the largest data structure of application S is doomed to have a fraction of actual data between 13% and 17%. Observe from Figure 4 that application S has a fraction of actual data lower than either figure. The largest data structure in the snapshot we analyzed was 170MB, of which 40MB came from the subset studied here. The remaining space was largely preallocated, but as yet unfilled, collections! The superposition of these two structures resulted in a overall fraction of actual data of around 8%. The second row poses the hypothetical study of adding more primitive data to the leaf-most regions. It shows that, in order to achieve $\mathcal{S} < 1.2$ (at least 80% actual data), the Edge and Integer structures must each have at least 242 bytes of data. This study highlights the severe inefficiencies of the application's data model.

5. Related Work

This work has been inspired by a variety of research.[12] Work on dynamic metrics [8] provides insight by quantifying program bloat. It does not do so in application-neutral terms, nor does it provide predictive models. There are many powerful memory profiling tools [23, 25] that identify suspicious data types, or memory leaks [11, 21, 16, 4, 13]. None of them study the underlying design causes of memory bloat or its asymptotic behavior. Others have worked on graph summarization [1, 19, 20, 15]. Each of these works has its strengths, such as for execution time profiling [1] or for more close coupling with source code [20]. We are aware of no summarization techniques that place the focus on containers and contained data structures that the content schematic offers. Automatic heap sizing [27] assumes the program's health as written. Conventional asymptotic analysis of algorithms [6] says nothing about the underlying causes of poor asymptotic behavior.

Finally, there are a number of works that characterize various aspects of the program, whether its configuration complexity [5], its defects [14, 26] or general behavior [10], or its performance bloat [18]. Many of these use application-neutral categories, but none is targeted to memory consumption, nor provides an asymptotic formulation.

6. Future Directions

Analyzing heap snapshots biases our analysis towards the longer-lived objects; we are extending this work to cover shorter-lived data. We can also leverage static information, and richer type information such as given by generic type systems, to possibly infer an approximation of health signatures from source code. We also see interesting possibilities in using health signatures and asymptotic analysis in development tools, to assist in better design, earlier in the development lifecycle.

One limitation of the current work is its inability to infer certain kinds of design intent. For example, to populate a structure may require random insertion and deletion. It is possible that, by incorporating knowledge about runtime access patterns, we can aid tool users in teasing out these cases. For example, it would be helpful to point out cases where a bloated structure is used well beyond the point where its construction has been completed. The user may then consider having two distinct forms: one optimized for construction, and a second optimized for read-only access.

7. Conclusion

A development team should assess the runtime consequences of their design and implementation choices. Ideally,

[12] A preliminary version of the work presented in this paper appeared in [17].

limit study	scaling formula ($\mathcal{S}$)	limit
vary big structures	$\frac{20m+119n+67}{2m+16n+6}$	$5.8 \leq \mathcal{S} \leq 7.4$
d necessary for $\mathcal{S} < 1.2$	$1 + \frac{48}{d}$	$d > 242$

Table 14. Two limit studies for Figure 7 (subset of application S).

this assessment should happen early and often, especially since ameliorating memory bloat may not be possible later on. In reality, teams often feel a sense of fatalism with respect to memory consumption, that this is an inexorable consequence of the object oriented paradigm. Fortunately, this is not the case, but it requires knowing whether an application's memory consumption is out of line with its needs.

As a step in that direction, we presented application-neutral characterizations of the health of a design. They offer the ability to compare one implementation to another and to known ideas of goodness, and to study the limits of the health of a design. By describing the *nature* of bloat, they can also serve to educate developers and managers of the trade-offs inherent in data structure design.

Acknowledgments

We would like to thank Tim Klinger, Palani Kumanan, and Edith Schonberg for their help with this work.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Programming Language Design and Implementation*, pages 85–96, 1997.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. Technical Report MSR-TR-2007-13, Microsoft Research, 2007.

[3] S. M. Blackburn and et. al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

[4] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *Architectural Support for Programming Languages and Operating Systems*, pages 61–72, 2006.

[5] A. B. Brown, A. Keller, and J. L. Hellerstein. A model of configuration complexity and its application to a change management system. In *Integrated Management*, 2005.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001.

[7] P. N. de Souza, R. J. Fateman, J. Moses, and C. Yapp. *The Maxima Book*, 2004. `http://maxima.sourceforge.net`.

[8] B. Dufour, K. Driesen, L. J. Hendren, and C. Verbrugge. Dynamic metrics for java. In *Object-oriented Programming, Systems, Languages, and Applications*, pages 149–168, 2003.

[9] GNU Trove: High performance collections for Java. `http://trove4j.sourceforge.net`.

[10] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 83–90, Montreal, Canada, June 1998.

[11] R. Hastings and B. Joynce. Purify — fast detection of memory leaks and access errors. In *USENIX Proceedings*, pages 125–136, 1992.

[12] S. Holzner. *Eclipse*. O'Reilly Media, Inc., first edition, Apr. 2004.

[13] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *Symposium on Principles of Programming Languages*, pages 31–38, 2007.

[14] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*, chapter 9 (Orthogonal Defect Classification). IEEE Computer Society Press, 1995.

[15] N. Mitchell. The runtime structure of object ownership. In *The European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer-Verlag, July 2006.

[16] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *The European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 351–377. Springer-Verlag, July 2003.

[17] N. Mitchell, G. Sevitsky, P. Kumanan, and E. Schonberg. Data structure health. In *International Workshop on Dynamic Analysis*, Minneapolis, Minnesota, United States, May 2007.

[18] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *The European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 429–451. Springer-Verlag, July 2006.

[19] S. Pheng and C. Verbrugge. Dynamic data structure analysis for java programs. In *International Conference on Program Comprehension*, June 2006.

[20] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic Analysis*, 2006.

[21] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Computational Complexity*, pages 50–66, 2000.

[22] Java 2 Platform, Enterprise Edition. `http://java.sun.`

com/j2ee.

[23] Quest Software. JProbe® Memory Debugger. http://www.quest.com/jprobe, 2005.

[24] SPEC Corporation. The SPEC JVM Client98 benchmark suite. http://www.spec.org/osg/jvm98, 1998.

[25] Yourkit LLC. Yourkit profiler. http://www.yourkit.com.

[26] T. Xie and D. Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *International Conference on Software Maintenance*, pages 28–37, Chicago, Illinois, Sept. 2004.

[27] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: Taking real memory into account. In *International Symposium on Memory Management*, 2004.