No Bit Left Behind: The Limits of Heap Data Compression *

Jennifer B. Sartor
The University of Texas at Austin
jbsartor@cs.utexas.edu

Martin Hirzel
IBM Watson Research Center
hirzel@us.ibm.com

Kathryn S. McKinley
The University of Texas at Austin
mckinley@cs.utexas.edu

Abstract

On one hand, the high cost of memory continues to drive demand for memory efficiency on embedded and general purpose computers. On the other hand, programmers are increasingly turning to managed languages like Java for their functionality, programmability, and reliability. Managed languages, however, are not known for their memory efficiency, creating a tension between productivity and performance. This paper examines the sources and types of memory inefficiencies in a set of Java benchmarks. Although prior work has proposed specific heap data compression techniques, they are typically restricted to one model of inefficiency. This paper generalizes and quantitatively compares previously proposed memory-saving approaches and idealized heap compaction. It evaluates a variety of models based on strict and deep object equality, field value equality, removing bytes that are zero, and compressing fields and arrays with a limited number and range of values. The results show that substantial memory reductions are possible in the Java heap. For example, removing bytes that are zero from arrays is particularly effective, reducing the application's memory footprint by 41% on average. We are the first to combine multiple savings models on the heap, which very effectively reduces the application by up to 86%, on average 58%. These results demonstrate that future work should be able to combine a high productivity programming language with memory efficiency.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); Optimization
General Terms Experimentation, Languages, Performance, Measurement
Keywords Heap, Compression

1. Introduction

Two consequences of Moore's law are (1) an increasing number of transistors in the same area, which server, desktop, and laptop form factors are now using for multicore processors, and (2) constant processing power on smaller and smaller devices, which is enabling more functionality in the embedded space. Since cache and memory consume a disproportionate amount of area and are expensive [21], the demand for memory efficiency is likely to remain constant or increase. To program all these devices, developers are increasingly turning to managed languages, such as Java [25], due to their *productivity* benefits, which include reduced errors through memory management, reliability due to pointer disciplines, and portability. Java, however, is not known for its memory efficiency and is therefore in conflict with hardware trends.

Researchers have characterized Java memory *usage* patterns [19, 10, 17], but do not study memory *savings* opportunities. A number of researchers propose and measure specific compression approaches [1, 4, 6, 7, 8, 13, 16, 18, 22, 23, 30]. For example, Ananian and Rinard use profiling and static analysis to implement approaches such as constant field elision and bit-width reduction [1]. Others explore approaches for making other programming languages more memory efficient [2, 9, 11, 24, 26, 27, 28, 29]. For example, Appel and Gonçalves share memory between equivalent SML objects using the garbage collector [2]. All the prior approaches consider and compare only a few proposals at a time. This paper compares a wide variety of compression techniques to provide a deeper understanding of memory efficiency and its limits.

This paper includes a comprehensive quantitative and qualitative comparison of heap data compression techniques. Our models of inefficiencies include strict and deep object and array equality, calculating dominant field values and field equivalence, removing zero-bytes, and compressing field values or array elements that use a small number and/or range of values. Our methodology periodically snapshots all heap objects and arrays by performing heap dumps during full heap garbage collections. We post-process these heap dumps to analyze memory inefficiencies and calculate memory savings per model. The contributions of this paper are:

1. Heap data compressibility analysis: A methodology for evaluating the memory savings limits of heap compression techniques.
2. Survey and models of compression techniques: Descriptions of over a dozen techniques with memory savings formulas.
3. Empirical evaluation, including combinations: Apples-to-apples comparison of individual as well as novel hybrid techniques.

Our experiments use the DaCapo Java benchmarks [19] and a variant of SPECjbb2000 called pseudojbb. We find that zero-based array compression saves the most memory (on average 17% of the heap including virtual machine objects, or 41% of the application). Together, deep-equal object and array sharing effectively reduce the total heap on average by 11%, and by 14% for applications. Overall we see that arrays take up the majority of space in the heap and can yield larger compression opportunities, so optimization efforts should be focused here. Experiments with Lempel-Ziv compression indicate that there is a large amount of redundancy in the heap (75 to 90% on average). Performing novel hybrid compression analysis with many savings models, we can get closer to this idealized compression, saving on average 34% of the heap, or 58%

* This work was supported by NSF CCF-0429859, NSF CCR-0311829, NSF EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

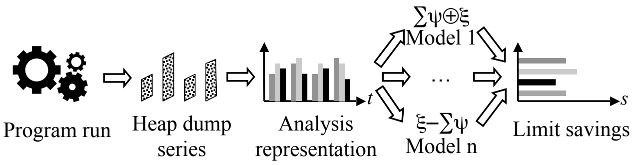

Figure 1. Heap data compressibility analysis.

for the application. We believe that presented compression techniques including new combined hybrids can reduce rampant heap bloat, making memory more efficient. This paper provides a foundation for the research community to make progress in heap data compression.

2. Heap data compressibility analysis

Figure 1 shows our analysis steps for measuring the potential of heap compression. We consider a conventional representation for dynamically allocated objects in a program. The heap contains two kinds of objects: class instances with fields and arrays with elements. Each object occupies a contiguous chunk of memory that consists of its fields or elements plus a header. We assume a conventional two-word object header with type information, garbage collector (GC) bits, and bookkeeping information for locking and hashing. Arrays have a third header word to store the length. Since we perform experiments with a Java-in-Java virtual machine (JVM), the heap contains both application and JVM objects.

Usually, the garbage collector loses some heap memory to fragmentation. We ignore fragmentation for two reasons: (1) the actual amount of fragmentation depends on the particular garbage collector in the runtime system; and (2) saving memory matters most at the peak memory usage, where it makes or breaks the ability to run in a given amount of memory. At peak utilization, the collector will likely apply defragmentation rather than crashing the program with an out-of-memory error. We only consider live objects in our analysis because we assume the garbage collector reclaims dead objects rather than compressing them.

From program run to heap dump series. Since a program's heap changes over time, its memory efficiency is also a function of time. A perfectly accurate heap analysis would compute savings on all live objects after every write and object allocation, but this analysis is prohibitively expensive. Instead, our analysis takes periodic heap snapshots during program execution ("Heap dump series" in Figure 1). It therefore over-approximates heap compression because, for example, a field value may be zero at every heap snapshot, but take on non-zero values between snapshots. We modify the garbage collector to print out a heap dump during live object traversal. In addition to its usual work, during a heap dump the garbage collector also prints object data (excluding bookkeeping information from the header) as it visits each live object on every collection. For each object, the GC prints the class identifier, the size in bytes, the address, whether the object was created by the JVM (`JZZ`) or the application (`AZZ`), the class name, and the list of fields or array elements, including their types and values. Here are two example objects from the heap dump, one class instance and one array:

```
T41 24 0x581e7454 JZZ Ljava/lang/String; \
  f0: object 0x581e746c  f1: int 9  f2: int 0x4c856879  f3: int 0
T26 28 0x581e7444 JZZ [Ljava/lang/Object; \
  reference array [object 0x581e746c,null,null,0x570ab004,]
```

Since heap dumps require a lot of I/O, they take a lot of storage and time to generate. More heap dumps yield more accurate compression measurements, but require more time and space. We empirically selected 25 as our target number of heap dumps. We execute the benchmarks with two times their minimum heap size using a mark-sweep collector, and print around 25 heap dumps at regularly-spaced intervals during normal collections. Our benchmarks perform between four and three hundred garbage collections at this heap size. For those benchmarks with fewer than 25 collections, we force more frequent collections in order to obtain the desired number of heap dumps.

We modified Jikes RVM, a Java-in-Java virtual machine [3] for our experiments. Jikes is unusual because it allocates both JVM and application objects in the heap. We differentiate between JVM and application object allocations by adding a small amount of instrumentation and stealing one unused bit from the object header. At allocation time, the JVM sets the bit to one to indicate that the JVM created the object, or zero for application objects. For most objects, their static class name reveals that they are a JVM object. For example, objects whose class prefix includes "jikesrvm" are JVM objects and objects whose class prefix includes "DaCapo" are application objects. However some cases are ambiguous because the JVM and application share the standard Java libraries, for example, java/lang/String, or the object's class is unspecified because they are primitive arrays. For these special cases, we classify objects as follows: a) We instrument each method call site that calls from a non-library method in to the class libraries. The JVM stores whether the caller is the JVM or the application in a thread-local variable. If the library performs allocation, the JVM queries the thread's local variable to tag the object with its proper status. b) For primitive arrays and other cases where the class is unknown, we walk the stack at allocation time to find the first non-library method descriptor, and tag the object accordingly [15].

From heap dump series to analysis representation. Given a series of heap dumps, a post-processing step applies analytical models that compute potential compression opportunities. The post-processor iterates over the heap dump, entering each object instance's data into a large hash table ("Analysis representation" in Figure 1). We compute memory savings per unique class. We do not further divide objects by their allocation site or data structure, which may be an interesting avenue for future work. The hash table stores every value for every field of each class at each collection. The table also stores the field's class information. The key to the hash table is a combination of the unique class identifier, the field number, a value for this field, and the collection number. The data for a given hash key is the number of object instances of this class during this collection with the same value for the field.

We enter arrays into the hash table as well, but since arrays of a particular type are not all the same length, all array entries are entered in the same "field" and also store the array element class. We compute most compression models after processing an entire heap dump into the hash table. Because we collapse all arrays of a type into one field, we accumulate per-instance savings as we process each array entry in the heap dump.

Helper functions. Many of our savings models require helper functions. Function $sizeof(T)$ returns the size of a primitive type in bytes. Some compression techniques require a hash table at runtime, for example, to find equivalent objects. Their models subtract the size of the hash table from the raw savings. Function $hashTableSize(n, entrySize)$ estimates the size of a hash table with n entries of size $entrySize$ each. We assume a hash table with open addressing, since they have no memory overheads for boxes or pointer chains for overflowing elements. We also assume that $\frac{2}{3}$ of the hash table is occupied. This assumption is conservative; for example, the Java library writers use a load factor of $\frac{3}{4}$ before doubling their size, although they use chaining instead of open-addressing. The helper function works as follows, where $arrayHeaderSize$ is 12 bytes and $keySize$ is 4 bytes:

$$\begin{aligned} &hashTableSize(numberOfEntries, entrySize) = \\ &\quad arrayHeaderSize \\ &\quad + \left\lceil \tfrac{3}{2} \cdot numberOfEntries \cdot (entrySize + keySize) \right\rceil \end{aligned}$$

Compression technique	Cls	Arr	Reference	GC/Run
Lempel-Ziv compression	3.1.2			GC
Strictly-equal object sharing	3.2.1	3.2.2		GC
Deep-equal object sharing	3.2.3	3.2.3	[2, 16]	GC
Zero-based object compression	3.2.4	3.2.4	[8]	GC
Trailing zero array trimming		3.4.1	[8]	GC
Constant field elision	3.3.1		[1, 24]	Run
Bit-width reduction	3.3.2	3.4.2	[1, 24, 30]	GC&Run
Dominant-value field hashing	3.3.3		[1]	GC
Dominant-value field elision	3.3.4		[6]	Run
Value set indirection	3.3.5	3.4.3	[9, 26]	GC
Value set caching	3.3.6	3.4.4		GC
Lazy invariant computation	3.3.7			GC

Table 1. Compression techniques modeled. Columns "Cls" and "Arr" refer to the subsections with the model for class instances or arrays, where applicable. Column "Reference" cites prior work that explored heap data space savings from this compression technique, if any. Column "GC/Run" says whether this model is calculated per collection or over all collections.

From analysis representation to limit savings. We then apply a variety of compression models to compute potential compression opportunities. Each model calculates the memory savings from a particular heap compression technique ("Limit savings" in Figure 1). Section 3 describes and presents formulas for all considered techniques. For many models, we calculate potential memory savings after examining each heap dump, i.e., after each collection. However, some models require the analysis to examine the data from the whole run of the benchmark. For example, if a particular field is constant throughout the entire run, then instead of allocating the same value in each instance, the JVM could eliminate the field from each instance and instead store the single value in a static class variable. To capture these diverse optimizations, our analysis applies compression models both per-collection on each heap snapshot, and over all the heap snapshots for a benchmark. For each snapshot, we count the number of object and application instances and bytes seen in order to calculate savings percentages.

3. Memory Compression Models

A compression model is a formula that computes how many bytes of heap data that technique can save at an instance in time. Table 1 overviews all the models considered in this paper. Most models are formulated for one class at a time. Some are formulated for one field at a time, or one instance for arrays. To obtain the total savings of a model, we compute the savings for each of the classes (or fields/instances) and then sum them up over all classes (or fields/instances).

3.1 Holistic heap data size and information content

Models in this section quantify the size of all the data in the heap. Because the heap contains redundancies, the actual information content is smaller than its conventional representation.

3.1.1 Total heap size

We measure the total heap size by summing all objects, fields, object headers, and array elements in the heap, assuming a conventional representation, and excluding fragmentation, static objects, and the stack. The below models compute savings from this baseline.

3.1.2 Lempel-Ziv compression

We first consider the memory savings achieved by simply zipping the contents of all heap objects. The size given by "`bzip2`" is a rough estimate of the true "information content" of the heap. We expect this savings to be larger than for any of the more realistic models below. Like the other models, Lempel-Ziv compression is

		Total			Application		
Bnchmrk	GCs	min	max	avg	min	max	avg
antlr	25	74	75	74	93	96	94
bloat	34	74	76	75	83	88	84
chart	24	74	75	74	82	92	91
eclipse	25	73	79	74	85	94	87
fop	20	74	75	74	89	99	95
fopfreq	283	74	75	74	89	99	96
hsqldb	24	75	83	81	83	96	85
jython	23	74	74	74	89	89	89
luindex	23	74	75	74	82	94	90
lusearch	26	74	81	80	92	96	96
pmd	22	74	79	75	83	94	95
xalan	22	78	79	79	94	95	91
pseudojbb	21	73	74	73	72	99	75
average		74	77	75	86	95	90

Table 2. Lempel-Ziv percent savings using "`bzip2`".

non-lossy, in other words, the original data can be fully recovered by decompression. Unlike the data representations for most of the other models, Lempel-Ziv compressed data does not permit random access, let alone in-place update. To compute this model as accurately as possible, we perform online compression on the actual heap in the JVM at garbage collection time. We perform compression with the same frequency as the heap dumps. As the collector traverses the object graph, it appends to a heap object stream an exact copy of all the bytes of each object and array, including their headers. To measure their differences, we put application objects in one stream, and all objects (both JVM and application) in another. We use native code to process the object streams so they do not pollute the Java heap or affect the frequency of garbage collections. At the end of the collection, we print the size of the full stream, i.e., all live data in the heap. We then apply Lempel-Ziv compression to the stream and report the compressed size as a percentage of the uncompressed size.

We show the Lempel-Ziv compression in Table 2 to illustrate the potential for heap reductions. The table shows each benchmark, the number of garbage collections (GC), and the minimum, maximum, and average over all snapshots for the total heap and application only savings. One line of the table, "fopfreq", is for a run with frequently forced heap dumps - over 280. When comparing this with the regular run of only 20 heap dumps, we see consistent results, showing that the timing of collections is not biased. Total heap compression is fairly consistent, reducing the heap between 73 and 83%. For only application objects we see larger compression opportunity, up to 99% for fop and pseudojbb. However, we do not expect this much compression in practice.

3.2 Object compression

This section presents object compression techniques that operate on entire objects, as compared to later sections, which describe compression techniques for individual fields and array instances.

3.2.1 Strictly-equal object sharing

Two objects are *strictly-equal* if they have the same class and all fields have the same value. Equality is strict because even pointer fields must be identical. Section 3.2.3 describes additional compression opportunities for objects with *deep equality* in which the pointer values are different, but the objects to which they point are equal [2, 16]. When objects are strictly equal, they can share all their memory. The JVM may allocate only one instance and then point all references of strictly-equal objects to the same instance.

In principle, two objects can not be shared if they are used for pointer comparison or as an identity hash code in the future. In addition, the period of time for sharing may be limited if the program modifies a strictly-equal object later. Our analysis ignores these cases for the purpose of this limit study. If class C has N objects, out of which D are distinct, then the memory savings are $(N - D) \cdot \mathit{sizeof}(C)$. With strict equality, finding the number of distinct objects (D) from a heap snapshot is linear in time and space. We simply iterate over all objects in class C and enter them in a hash table and do not store duplicates. We use the value of all fields as the hash key. In the end, the number of entries in the hash table is D. Online implementations of object sharing use a hash table at runtime as well. This model provides the following net savings:

$$(N - D) \cdot \mathit{sizeof}(C) - \mathit{hashTableSize}(D, \mathit{pointerSize})$$

3.2.2 Strictly-equal array sharing

Array sharing is similar to sharing of non-array objects, except that the array length must match [16]. Since different length arrays have different sizes, we iterate over all arrays to add up their sizes before compression, construct the hash table, and then iterate over all D distinct arrays to find the unique size. The model must also subtract the memory used for the hash table itself. The resulting savings model for array type $T[\,]$ is:

$$\sum_{a \in T[\,] \wedge a \notin D} \mathit{sizeof}(a) - \mathit{hashTableSize}(D, \mathit{pointerSize})$$

3.2.3 Deep-equal object and array sharing

Two objects can share memory even if they differ in a pointer field, as long as the targets of the pointers are equivalent [2, 16]. Every strictly-equal object pair is also deep-equal. Because there are more deep-equal object pairs than strictly-equal ones, deep-equal sharing yields additional compression opportunities.

In the absence of cycles, deep equality can use a bottom-up traversal of the object graph, for example, by piggybacking on GC reachability traversal by adding a post-order breadth-first visitor [2]. This traversal computes sharing for all the leaves first, then computes sharing for the next level of the object graph, and so on. It thus guarantees that when visiting an object, it has already calculated the sharing of all the objects to which it points. Therefore, we simply compare objects at the current level and one level deeper using a hash table. Thus, for the entire heap, deep-equal acyclic object sharing takes time $O(e)$, where e is the number of pointers in reachable heap objects.

Cycles complicate deep equality comparison. A naive algorithm would just propagate sharing opportunities from objects to their predecessors until reaching a fixed point. We are using this approach, but the fixed-point iteration is slow for some benchmarks. Marinov and O'Callahan point out that determining deep equality is a special case of the graph partitioning problem [16] and recommend Cardon and Crochemore's graph partitioning algorithm that takes $O(e \log n)$ [5].

The savings model for a class C with N instances out of which D are distinct is the same as in strictly-equal sharing, except that D is smaller. There are fewer distinct objects than in strictly-equal sharing, since deep equality exposes more sharing opportunities:

$$(N - D) \cdot \mathit{sizeof}(C) - \mathit{hashTableSize}(D, \mathit{pointerSize})$$

or, for arrays of type $T[\,]$:

$$\sum_{a \in T[\,] \wedge a \notin D} \mathit{sizeof}(a) - \mathit{hashTableSize}(D, \mathit{pointerSize})$$

3.2.4 Zero-based object and array compression

Zero-based object compression reduces object size by removing bytes that are zero. We assume an implementation that uses a per-object bit-map to indicate which bytes in the original object are entirely zero [8]. The compressed object representation consists of the header, the bit map, and the values of all non-zero bytes. The size of the bit map is the number of non-header bytes in the original object. The bit-map for object o occupies $\lceil \mathit{totalBytes}(o)/8 \rceil$ bytes. The savings for all objects in the heap are therefore:

$$\sum_{o \in \mathit{Objects}} \left(\mathit{zeroBytes}(o) - \left\lceil \frac{\mathit{totalBytes}(o)}{8} \right\rceil \right)$$

Note that this compression scheme can be applied to both array and non-array objects, and to both primitive and reference fields. We compute memory savings per instance, and then add them up for each class.

3.3 Field compression

Field compression techniques operate on the fields of class instances, which excludes static fields and array elements.

3.3.1 Constant field elision

Constant field elision saves memory by eliding a field if it is constant for all instances of that type. If all instances of class C have the same value for instance field f, that field can be static [1, 24]. For N objects, that saves:

$$(N - 1) \cdot \mathit{sizeof}(f)$$

We compute constant field elision savings by accumulating information over all heap dumps of an experimental run.

3.3.2 Field bit-width reduction

Field bit-width reduction saves memory by allotting fewer bytes for the field than the type takes. If all objects of class C have small values in instance field f, then they can all be represented with a smaller bit-width [1, 24]. For N objects, that saves:

$$N \cdot \big(\mathit{originalBitWidth}(f) - \mathit{reducedBitWidth}(f)\big)$$

We measure field bit-width reduction both per heap dump and over all heap dumps in the run. Field bit-width reduction over all heap dumps takes all seen field values into account and therefore more accurately reflects true memory savings potential.

3.3.3 Dominant-value field hashing

Dominant-value field hashing compresses objects by eliding fields with one dominant value, storing this value as a static class member [1]. The JVM can store aberrant values of instance field f of class C in a hash table using the object ID as the hash key. We only store aberrant values in the hash table; therefore, if the object ID does not exist in the hash table, the program will access the field statically.

We assume that class C has N instances, and that D instances have the dominant value in the field. For example, consider $N = 1000$ instances and $D = 990$ of them have the dominant value. The other $N - D = 10$ instances have hash table entries. The savings are:

$$N \cdot \mathit{sizeof}(f) - \mathit{hashTableSize}\big(N - D, \mathit{sizeof}(f)\big)$$

For the example, dominant-value field hashing would save 31,448 bytes = $1000 * 32 - (3/2 * 10 * (32 + 4) + 12)$. Clearly, there is a cross-over point where the savings become negative. If the computed savings is negative, we assume zero savings as we would not apply the optimization. Because this technique relies on an object ID, actual savings may be lower if the JVM has to use additional memory on ID tracking, for example, if it uses a moving collector.

For the special case of a boolean field, the presence or absence in the hash table is enough to indicate the value without having to store an entry, so the savings are:

$$N \cdot \mathit{sizeof}(f) - \mathit{hashTableSize}\big(N - D, 0\big)$$

3.3.4 Dominant-value field elision

Chen, Kandemir, and Irwin introduced dominant-value field elision [6]. Their approach targets the same inefficiencies as dominant-value field hashing, but their implementation uses offline profiling and then makes dynamic per-instance decisions to deal with mistakes. In addition, they make per-class decisions, rather than per-field decisions, that consider all the fields in each class together.

Chen et al. identify the frequent value for each field per class after a particular benchmark run. In an offline pass, they use the frequent field value count to choose particular fields as good candidates for optimization in a separate benchmark run. If most object instances of a class hold the same dominant value for a particular field, the dominant value fields can be shared by many instances, thereby saving memory. They separate dominant values into two kinds: zero and non-zero. Fields with a dominant value of zero (including null, in the case of pointers) can be elided entirely and no storage need be used for them, whereas non-zero (strictly-equal, in the case of pointers) dominant fields must be stored and pointed to by instances that use them.

For a class C, Z fields are zero-dominant. Similarly, NZ is the subset of fields that are dominant with non-zero values. Because of their object layout, Chen et al. only achieve savings for an object instance if *all* fields in Z remain zero. Similarly, if all fields in NZ retain their dominant values, they achieve compression. If any field in Z or NZ does not retain its dominant value, those subsets of the object cannot be elided, and their implementation will allocate memory for all (dominant and non-dominant) fields in the object.

Determining dominant field sets: To select groups of fields in class C to place in Z and NZ, we define $\mathit{domInstances}(C, f)$ as the number of instances of class C which have the dominant value for field f. We define $\mathit{domSortedFields}$ as a list of fields in class C in descending order of $\mathit{domInstances}(C, f)$. Then, we define

$$\mathit{quality}(i) = i \cdot \mathit{domInstances}(C, \mathit{domSortedFields}[i])$$

This product multiplies the number of fields i by the number of instances $\mathit{domInstances}(C, \mathit{domSortedFields}[i])$ for which those fields could be saved. Chen et al. determine the m that maximizes $\mathit{quality}(m)$. The first m elements of the list $\mathit{domSortedFields}$ go into Z for zero fields, or NZ for non-zero fields[1].

For example, say class C has 3 fields, f_1, f_2, and f_3. Given 20 instances of class C, say $\mathit{domInstances}(C, f_1) = 16$ which means that 16 of the 20 instances share a dominant value for f_1. Let's say $\mathit{domInstances}(C, f_2) = 18$ and $\mathit{domInstances}(C, f_3) = 10$. Since $\mathit{domSortedFields}$ is sorted by descending $\mathit{domInstances}$, we have $\mathit{domSortedFields} = [f_2, f_1, f_3]$. Then we take the max of the following $\mathit{quality}$ values:

$$\begin{aligned}
\mathit{quality}(1) &= 1 \cdot \mathit{domInstances}(C, f_2) = 1 \cdot 18 = 18 \\
\mathit{quality}(2) &= 2 \cdot \mathit{domInstances}(C, f_1) = 2 \cdot 16 = 32 \\
\mathit{quality}(3) &= 3 \cdot \mathit{domInstances}(C, f_3) = 3 \cdot 10 = 30
\end{aligned}$$

Since $\mathit{quality}(2)$ is largest, we know that we should optimize fields 1 and 2 in this class, not field 3. We thus consider two fields for zero or non-zero field elision based on their dominant value.

Following the methodology of Chen et al., we compute the candidate fields for dominant-value field elision using data from all snapshots of a run of the benchmark. We then compute the memory savings per instance by processing all heap dumps a second time.

Dominant zero field elision: Given an object of class C, if all the fields in Z are zero or null, the implementation sets a bit in the object header to record that information and does not store the fields in the object. If at least one of the fields in Z is non-zero or non-null, we clear the bit and hijack the class pointer in the header of the object to point to a secondary object. The secondary object stores the values of the fields in Z.

Assume class C has N instances, and M instances require a secondary object because at least one of the fields in Z is non-zero or non-null. For $N - M$ objects, we save $\mathit{sizeof}(Z)$ each. We assume the extra bit per object, whether compressed or not, is stolen from the object header. For each of the remaining M objects, we have the overhead of a secondary object header, and of course, we don't save $\mathit{sizeof}(Z)$. So the total memory savings for class C are:

$$(N - M) \cdot \mathit{sizeof}(Z) - M \cdot \mathit{headerSize}$$

Dominant non-zero field elision: For the first instance of class C that has all fields in NZ that match dominant values, we assume a secondary object holding those dominant values is allocated. We assume part of the original object header points to the secondary object. The secondary object adds the cost of its header. However, for subsequent instances that have all fields in NZ with dominant values, we simply use a bit in the object header to indicate this instance shares a secondary object and point it to the previously created secondary object. For this instance, we save sizeof(NZ) memory. If an instance of class C has even one field of NZ that holds a non-dominant value, we cannot save memory and this instance has to allocate its own secondary object.

We assume class C has N instances, and M instances require a secondary object because at least one of the fields in NZ does not hold its dominant non-zero value. Memory savings are similar to dominant zero field elision; however, we have to reserve memory for one secondary object, including a header, to hold NZ's dominant values. So the total memory savings for class C are:

$$(N - M - 1) \cdot \mathit{sizeof}(NZ) - (M + 1) \cdot \mathit{headerSize}$$

3.3.5 Field value set indirection

Field value set indirection saves memory by holding a "dictionary" of values for a field separately from object instances, enabling instance fields to hold a smaller index into the dictionary. If the field f of class C has only a few distinct values over all instances of C, then instead of storing those values directly, it stores the dictionary index, and the dictionary stores the actual values [9, 26]. Specifically, if field f stores at most $K < 256$ different values, then instead of storing the values directly, store an 8-bit index into a K-entry dictionary. If class C has N instances, the savings are:

$$N \cdot \big(\mathit{sizeof}(f) - 1\big) - \big(\mathit{arrayHeaderSize} + K \cdot \mathit{sizeof}(f)\big)$$

Field value set indirection makes no assumptions about the type of field f: it applies equally well to char, int, float, pointer, etc. Where bit-width reduction requires all field values to be small, value set indirection only makes requirements on the number of field values. Set indirection applies more generally than bit-width reduction. It also reduces field width, but requires extra space for the dictionary.

3.3.6 Field value set caching

Field value set caching is similar to field value set indirection, but is performed only on fields with $K >= 256$ values, and thus requires some extra separate storage. In the object instance, the field is just

[1] Chen et al. sum up the $\mathit{domInstances}(C, f)$ over the set of all classes that are C or a subclass of C. We do not for simplicity.

an index into a dictionary as in field value set indirection. The most frequent 255 values are "cached" in the dictionary (to allow an 8-bit index). For the other $K - 255$ values, the 256th entry in the dictionary is reserved to indicate that the value is not cached. In that case, the field is stored in a hash table indexed by the object ID. To compute the savings, assume the class has N objects, and M objects have a value in field f that is not among the 255 most frequent values for that field. In practice, if the field values are skewed, M is small. The memory savings are:

$$\begin{aligned}&N \cdot \big(\mathit{sizeof}(f) - 1\big)\\&- \mathit{arrayHeaderSize} - 255 \cdot \mathit{sizeof}(f)\\&- \mathit{hashTableSize}(M, \mathit{sizeof}(f))\end{aligned}$$

Field value set caching also makes no assumptions about the type of field f: it applies equally well to char, int, float, pointer, etc.

3.3.7 Lazy invariant computation

Assume class C has two fields, f_1 and f_2, and they are always identical. Then we only need to store one of them, and save the memory for the other one. As another example, assume class C has three fields, f_1 f_2 f_3, and it is always the case that $f_1 = f_2 + f_3$. Then, we do not need to store f_1, since we can always compute it from f_2 and f_3. In the most general case, if there is a way to compute a field f from other fields of the same object, we can elide the field.

We cannot possibly check for all possible field invariants that translate into memory savings. In our experiments, we only explore the case where two fields are always identical. However, a tool like Daikon or DIDUCE tool [12, 14] could provide invariants which is a possibility for future work. Assume it takes I bytes to encode the invariant. If we eliminate field f in N instances, the savings are

$$N \cdot \mathit{sizeof}(f) - I$$

For our experiments, we can elide the duplicate field entirely, and expect to statically store information of size $I = \mathit{sizeof}(f)$ that says which field to look up instead. For this special case, savings come out the same as constant field elision even though the field is not constant over all instances.

3.4 Array object compression

Array compression techniques operate on array instances. We compute overall compression by accumulating savings for all instances of all classes.

3.4.1 Trailing zero array trimming

Programs often over-provision the capacity of arrays used as buffers, leading to unused trailing zeros [8]. These can be trimmed, provided that the trimmed array remembers the nominal and true length. Assuming it takes an additional 4 bytes to store both lengths, the savings for array type $T[\,]$ are:

$$\sum_{a \in T[\,]} \big(\mathit{trailingZeros}(a) \cdot \mathit{sizeof}(T) - 4\big)$$

3.4.2 Array bit-width reduction

Array bit-width reduction computes savings per instance, compressing array elements similarly to field bit-width reduction.

Boolean arrays: Per default, a Java virtual machine uses a byte to represent a boolean, and hence, an array of L booleans occupies 3 words for the header plus L bytes for the elements. The trivial optimization of representing an array of boolean by a bit vector saves:

$$\sum_{a \in \mathbf{boolean}[\,]} \left\lfloor \frac{7}{8} \cdot a.\mathit{length} \right\rfloor$$

Character arrays: Java represents characters using a 16-bit encoding for unicode. But English-language applications tend to use mostly characters that require only the lower 8 bits. The accordion arrays bit-width compression optimization represents each array that consists entirely of 8-bit characters using a byte array [30] to save:

$$\sum_{a \in \mathbf{char}[\,] \wedge \mathit{onlyUsesBits}(a,8)} \lfloor a.\mathit{length} \rfloor$$

Other types: The above examples use boolean and char arrays, but the array bit-width reduction also works for arrays of short, int, or long [24]. In general, you can optimistically represent an array of type $T[\,]$ as a B-bit array provided all values need at most B bits to save:

$$\sum_{a \in T[\,] \wedge \mathit{onlyUsesBits}(a,B)} \left\lfloor \frac{8 \cdot \mathit{sizeof}(T) - B}{8} \cdot a.\mathit{length} \right\rfloor$$

3.4.3 Array value set indirection

Array value set indirection is similar to field value set indirection. If all elements of all arrays of a given class have elements drawn from a small set of distinct values, then replace each instance element with a small index into a dictionary that stores the actual value. For example, if all instances of an array type $T[\,]$ contain at most $K < 256$ different values, array elements can store an 8-bit index into a K-entry table of values of type T. The memory savings are:

$$\begin{aligned}&\sum_{a \in T[\,]} a.\mathit{length} \cdot \big(\mathit{sizeof}(T) - 1\big)\\&- \mathit{arrayHeaderSize} - K \cdot \mathit{sizeof}(T)\end{aligned}$$

This optimization makes no assumptions about the element type T. It applies equally well for int, float, pointer, etc. This model does reduce element size, but because it is more generally applicable than array bit-width reduction, it incurs the overhead of storing the dictionary.

3.4.4 Array value set caching

Array value set indirection can be generalized to the case where there are a few aberrant values that do not fit in the primary dictionary. Caching reserves one dictionary index (of 256) to indicate an aberrant value, and stores the aberrant values into a secondary hash table. We use a combination of the original array's object ID and the index of the array element for the secondary hash table's key. An array access `a[i]` in this case is as follows:

```
if a[i] == aberrant_indicator:
  return secondary_hash.get(a, i)
else:
  return dictionary[a[i]]
```

Let A be the total number of aberrant array elements in all arrays of type $T[\,]$. Then the savings are:

$$\begin{aligned}&\sum_{a \in T[\,]} a.\mathit{length} \cdot \big(\mathit{sizeof}(T) - 1\big)\\&- \mathit{arrayHeaderSize} - K \cdot \mathit{sizeof}(T)\\&- \mathit{hashTableSize}'\big(A, \mathit{sizeof}(T)\big)\end{aligned}$$

The $\mathit{hashTableSize}'$ function assumes that keys are 8 bytes, because the key represents both an array and an index.

3.5 Hybrids

Hybrids combine multiple compression techniques to obtain more savings than one technique alone.

3.5.1 Maximal hybrid

The maximal hybrid chooses the compression technique that saves the maximum amount of memory for each piece of data. We first compute the maximum for field techniques. For example, within the same class C, one field may save most from bit-width reduction, another field may save most from dominant-value hashing. The maximal hybrid uses the technique that saves the maximum amount of memory for each particular field f of class C. Savings are:

$$maxFieldSavings(C) = \sum_{f \in Fields(C)} \max_{o \in FieldOpts} savings(C, f, o)$$

In other words, we sum the savings from each field f when using the optimization o with maximum savings for that field. The set $FieldOpts$ contains constant field elision (per snapshot), bit-width reduction, dominant-value hashing, value set indirection or caching, and lazy invariant computation[2].

Besides field optimizations, the maximal hybrid also considers optimizations that apply to entire objects of a class rather than individual fields. Again, the idea is to pick, for each class, the optimization that yields the highest savings:

$$maxClassSavings = \sum_{C \in Classes} \max \left\{ maxFieldSavings(C), \max_{o \in ClassOpts} savings(C, o) \right\}$$

Note that $maxClassSavings$ considers $maxFieldSavings(C)$ as one alternative for each class, along with the class optimization techniques in $ClassOpts$, which are zero-based object compression and strictly-equal object sharing[3].

For arrays, the maximal hybrid starts by choosing the maximal array instance compression techniques:

$$maxArrayISavings(T[]) = \max_{o \in ArrayIOpts} \sum_{i \in T[]} savings(T[], i, o)$$

The set of array instance optimizations $ArrayIOpts$ contains trailing zero trimming, bit-width reduction, and zero-based compression. Next, just like for classes, the maximal hybrid picks the best optimizations for each array type $T[]$:

$$maxArrayTSavings = \sum_{T[] \in Arrays} \max \left\{ maxArrayISavings(T[]), \max_{o \in ArrayTOpts} savings(T[], o) \right\}$$

The set of array type optimizations $ArrayTOpts$ contains strictly-equal array sharing, value set indirection, and value set caching.

In total, the maximal hybrid saves the sum of the maximal savings of class and array type optimizations:

$$maxClassSavings + maxArrayTSavings$$

3.5.2 Combined hybrid

In some cases, after applying an optimization o_1 to a piece of data, it is possible to apply o_2 as well on the same data to obtain additional savings. For example, we can have perform "trailing zero array trimming"=o_1 first and then do "array bit-width reduction"=o_2, achieving more savings with the hybrid $o_1 \circ o_2$ than either o_1 or o_2.

We calculate combined-hybrid heap compression by applying multiple models in sequence. We first find $maxFieldSavings(C)$ for each class C. In other words, each field is optimized with the best optimization in $FieldOpts$ for that field. Next, we check if each instance can benefit further from the optimizations in $ClassOpts$: zero object compression and strictly-equal object sharing. To correctly compute these compression models, the previously-applied field optimizations modify the analysis representation as required, for example, by changing field and object sizes, and by re-populating the hash table for strictly-equal object sharing. We then simply add up each type's savings to achieve a global $combinedClassSavings$, which is the total savings from applying all non-array optimizations in sequence.

We compute hybrid array savings similarly. For the maximum potential savings, per array instance, we apply the optimizations from $ArrayIOpts$ in the following order: (1) trailing zero trimming, (2) bit-width reduction, and (3) zero-based compression. Throughout these calculations, we keep track of changes to the array length, array size, element size, and number of zero entries to feed into later optimizations. Similar to combined object savings, we follow the instance optimizations by type optimizations in $ArrayTOpts$ to explore further compression. Even if instance optimizations have been performed to reduce the array footprint, strictly-equal array sharing, array value set indirection, and caching could realize further savings. However, we do not need to recalculate the array sharing hash table, as instance optimizations only elide zeros and do not change element values. After we calculate combined savings for each array type, we add them to compute the total $combinedArrayTSavings$.

We then sum combined-hybrid class and array type savings to obtain total combined-hybrid savings:

$$combinedClassSavings + combinedArrayTSavings$$

4. Results

This section evaluates and compares the compression models.

4.1 Methodology

We added heap data compressibility analysis to Jikes RVM [3] version 2.9.1. We used the "FastAdaptiveMarkSweep" configuration, which optimizes the boot image ("Fast") and uses a mark-sweep GC. We disabled the optimizing compiler during the application run to reduce compiler objects in the heap. Since Jikes RVM itself is written in Java, it allocates JVM objects in the Java heap alongside application objects; we show both total and application-only results. Our benchmark suite consists of the DaCapo benchmarks [19] version "dacapo-2006-10-MR1", and of pseudojbb, a variant of SPECjbb2000 (see `www.spec.org/osg/jbb2000/`). We used Ubuntu Linux 2.6.20.3.

Bnchmrk	GC	# Types		Size [KB]			Size [Instances]		
		Cls	Arr	Total	Arr	App	Total	Arr	App
antlr	14	529	69	49,811	70%	5%	785,561	33%	0.5%
bloat	122	593	79	56,724	67%	16%	967,360	32%	17%
chart	23	675	85	56,616	68%	8%	970,070	33%	3%
eclipse	44	1,136	175	87,799	68%	25%	1,534,580	35%	18%
fop	22	784	73	52,184	70%	4%	830,111	34%	0.6%
hsqldb	13	538	78	201,375	43%	76%	7,231,418	21%	89%
jython	218	892	78	63,706	67%	9%	1,067,359	33%	6%
luindex	11	533	70	50,185	70%	7%	794,633	33%	2%
lusearch	26	536	73	70,994	78%	34%	850,828	33%	8%
pmd	71	644	72	59,220	67%	9%	992,510	34%	6%
xalan	125	711	88	71,148	76%	27%	940,201	36%	9%
pseudojbb	18	495	72	74,180	73%	35%	1062,901	36%	25%

Table 3. Benchmark and heap dump characterization.

Due to space constraints, we pick one representative heap dump mid-way through the approximately 20 we gather. For more comprehensive results, see the companion technical report [20]. Section 4.5 validates the generalizability of one heap dump for an example benchmark, comparing savings over many dumps. Even though savings are for one dump, some models consider constraints

[2] Due to the offline analysis required in dominant value elision, we exclude it from this hybrid calculation.

[3] Due to implementation limitations, we exclude deep-equal object sharing from this hybrid calculation.

from all dumps, see Table 1. Table 2 shows the number of heap dumps. Table 3 characterizes our benchmark suite, including the number of GCs, the number of class and array types represented in the measured heap dump, and the size of the measured heap dump. For all benchmarks, arrays occupy more bytes but have fewer instances than classes. The application occupies between 4% and 76% of the amount of bytes occupied by Jikes RVM. Subsequent sections and tables represent total memory savings as a percentage of total KB, and represent application memory savings as a percentage of application KB. We were not able to compute application-specific savings for models that require analysis over all dumps.

4.2 Object compression

Bnchmrk	Total						Application					
	Equal sharing				Zero-based		Equal sharing				Zero-based	
	Strictly		Deep				Strictly		Deep			
	Cls	Arr	Cls	Arr	Cls	Arr	Cls	Arr	Cls	Arr	Cls	Arr
antlr	2	4	4	5	6	14	0	12	0.2	12	0.2	48
bloat	1	8	7	8	6	14	0	6	3	7	8	25
chart	2	4	4	5	6	13	2	2	3	2	2	41
eclipse	1	4	2	4	6	14	0.2	2	0.5	2	5	25
fop	2	4	4	5	6	16	0	14	0.1	15	0.3	57
hsqldb	0.5	3	1	3	8	12	0	3	0	3	9	11
jython	1	6	5	7	6	13	0	8	0.6	10	3	28
luindex	2	5	4	5	6	15	0.4	27	0.9	27	2	58
lusearch	2	4	4	4	4	32	2	4	4	5	1	72
pmd	1	7	6	7	7	14	0.1	9	2	9	8	33
xalan	1	21	3	22	5	29	0.7	58	2	59	1	66
pseudojbb	1	4	4	5	5	20	0	4	0	4	4	33
average	1	6	4	7	6	17	0.5	12	1	13	4	41

Table 4. Percent memory savings from object compression.

Table 4 shows memory savings from the object compression techniques in Section 3.2. As expected, deep-equal sharing results in more savings than strictly-equal sharing. Whereas deep equality is essential for saving class-instance memory, strict equality suffices for arrays, because most arrays are primitive. *Xalan* in particular benefits greatly from deep-equal array sharing, saving 22% of the total heap and 59% of the application heap. Also, zero-based compression gets some good savings ranging from 4 to 8% for classes and 12 to 32% for arrays. Application-specific zero-based compression savings greatly depend on the benchmark. Application numbers vary more widely than total numbers, showing that the JVM is fairly consistent. Overall zero-based compression achieves the highest individual savings (with objects and arrays 36%), but at the cost of having to decompress individual object instances before use.

4.3 Field compression

Table 5 shows memory savings from the field compression techniques in Section 3.3. A field can be constant-elided only if it is constant over the whole run. Our benchmarks have a significant number of elidable constant fields. The "Bitw" columns refer to field bitwidth reduction, either based on whether the field obeyed its bitwidth over all heap dumps ("Run") or just the selected heap dump ("GC"). In both cases, reducing field bit-width can save between 4 and 7% of the heap size. This similarity implies that the range of all field values is fairly accurately represented in one heap dump. As discussed in Section 3.3.4, dominant-value field elision requires two passes over all heap dumps, one to compute candidate fields and another to consider all instances for savings. Zero elision is effective for fields, for example, it reduces *bloat* by 12%. Fewer fields can be elided due to a non-zero dominant value as expected, but it saves up to 6% on *eclipse*. However, both dominant elision techniques require ahead of time profiling to achieve savings. Dominant-value hashing and value set indirection each can save 4 to 9% of the heap. So our benchmarks have many fields with one dominant value, and have many fields with fewer than 256 values that can benefit from a "dictionary". Our benchmarks do not see a lot of benefit from value-set caching, meaning they do not have many fields with more than 256 values. Similarly, few pairs of fields are equal over all instances of a class, so lazy invariants do not compress the heap much. Overall, the field optimizations yield smaller savings than object compression. For field compression, bit-width reduction, dominant-value hashing, and value set indirection yield the greatest savings. If it is easy to have an offline pass of the run, then dominant zero elision affords good compression too.

4.4 Array compression

Bnchmrk	Total						Application				
	Trl	Bitwidth			Value set		Trl	Bitwidth		Value set	
	Zro	Ch	GC	Run	Indr	Cch	Zro	Ch	GC	Indr	Cch
antlr	3	4	8	0.2	0.3	5	3	2	2	2	0
bloat	3	5	8	0.4	0.5	6	3	9	10	10	0
chart	3	4	8	0.2	0.3	5	11	2	4	0	2
eclipse	2	6	9	0.1	0.2	6	0.9	13	13	0.1	13
fop	6	4	7	0.2	0.5	5	0.1	0.1	0.4	0.5	0
hsqldb	1	0.9	2	0	0.1	1	0.4	0	0.3	0	0.3
jython	2	5	8	0.2	0.4	6	2	3	4	0.2	5
luindex	5	4	8	0.2	0.3	5	25	0.9	1	0.9	0.2
lusearch	2	5	7	0.1	0.2	5	0.9	5	6	6	0
pmd	3	5	9	0.2	0.5	6	8	5	6	5	1
xalan	18	4	10	0.2	2	7	49	6	17	13	24
pseudojbb	3	14	18	0.1	0.4	14	4	32	34	32	0
average	4	5	9	0.1	0.4	6	9	7	8	6	4

Table 6. Percent memory savings from array compression.

Table 6 shows memory savings from the array compression techniques in Section 3.4, which are overall greater than for objects. Many benchmarks have a significant amount of arrays allocated with padding that can benefit from trailing zero trimming. *Xalan*, with a large part of its heap being arrays, can save 18% by trimming zeros. For just application arrays, this model helps most for *chart*, *luindex*, and *xalan*. Our benchmarks spend very little space on boolean arrays, hence we do not show a separate column for bit-width compressing them; the numbers were all zero. However, many character arrays benefit from bit-width reduction – up to 14% with *pseudojbb*. *Eclipse* and *pseudojbb* application character arrays are prime candidates for savings. Although application savings vary per benchmark, similar to Zilles' results which were computed with different methodology, we see up to 32% compression possibility [30]. Interestingly, arrays of types other than character can also benefit significantly from bitwidth reduction, with savings between 2 and 18%. Value set indirection helps little for arrays, because it is too strict: at least some arrays exceed the allotted dictionary. However, using the dictionary as a cache and placing aberrant values in a secondary hash increases the opportunities significantly, and so value set caching saves more memory. In particular, *pseudojbb* can compress the heap by 14% with value-set caching. Overall, the array optimizations in this section yield smaller savings than zero-based object compression for arrays, but value caching and bit-width reduction are competitve with deep-equal sharing.

4.5 Compressibility over time

Most of the results in this section are for one mid-run heap dump only. We investigated whether one heap dump can be representative for the entire run by plotting a compressibility time series for one benchmark, *fop* in Figure 2. We forced frequent heap dumps every 512KB of allocation, collecting 148 heap dumps. Each curve is one compression technique, the x-axis is time, and the y-axis is the percent memory savings. The lines are mostly horizontal, validating that compressibility changes little from heap dump to heap dump. More variation is seen at startup and shutdown as expected, but the middle of the run is fairly stable. This shows our per collection savings for classes and arrays at a middle heap

Bnchmrk	Total Per Run				Total					Application				
	Const Elision	Bitw Run	Dom elision Zero	Dom elision !Zero	Bitw GC	Dom Hash	Value set Indir	Value set Cache	Lazy Invar	Bitw GC	Dom Hash	Value set Indir	Value set Cache	Lazy Invar
antlr	4	5	5	1	4	5	5	1	1	0.2	0.1	0.2	0	0
bloat	4	7	12	1	5	6	5	1	1	6	7	8	0.2	0.4
chart	4	5	4	0.8	5	6	5	1	1	1	0.8	2	0.2	0.1
eclipse	5	6	9	6	5	6	6	1	1	5	6	7	0.4	0.8
fop	6	7	6	1	4	5	5	1	1	0.3	0.1	0.1	0	0
hsqldb	4	7	4	3	7	9	7	0.2	0.2	7	10	7	0	0
jython	4	5	5	0.9	5	6	5	1	0.9	3	3	4	0	0.1
luindex	4	5	5	1	5	5	5	1	1	2	1	2	0	0.3
lusearch	3	4	3	0.7	4	4	4	1	0.7	1	0.8	1	0.7	0
pmd	6	8	8	2	5	6	6	0.9	1	10	9	9	0	4
xalan	3	4	5	1	4	4	4	0.7	0.9	1	1	1	0	0.5
pseudojbb	3	4	4	1	4	5	4	1	0.7	3	4	5	0	0.2
average	4	6	6	2	5	6	5	0.9	0.8	3	4	4	0.1	0.5

Table 5. Percent memory savings from field compression.

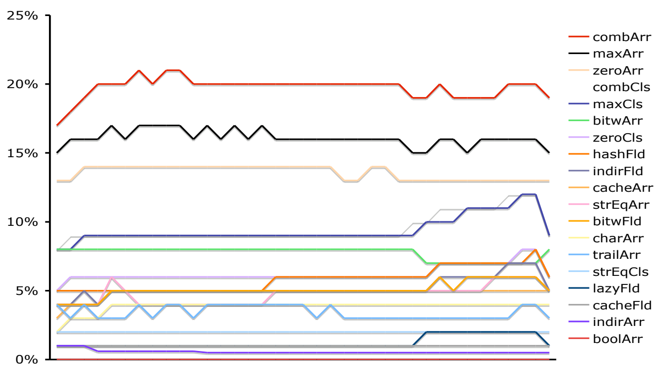

Figure 2. Compressibility of *fop* over time.

dump should be representative. We also gathered savings for the per run models with frequent heap dumps which were very similar to savings for 20 heap dumps, showing that there is little collection bias.

4.6 Hybrid object compression

Benchmark	Total				Application			
	Maximal Cls	Maximal Arr	Combined Cls	Combined Arr	Maximal Cls	Maximal Arr	Combined Cls	Combined Arr
antlr	9	16	9	20	0.3	48	0.3	50
bloat	10	17	10	21	12	28	13	38
chart	9	16	9	19	4	43	3	46
eclipse	9	17	10	22	10	28	10	41
fop	9	18	9	22	0.3	58	0.3	60
hsqldb	12	12	12	13	12	12	12	12
jython	9	16	10	21	5	30	5	35
luindex	9	17	9	21	3	61	3	66
lusearch	7	34	7	39	3	73	3	80
pmd	10	16	10	22	14	34	14	37
xalan	7	33	7	38	2	78	2	84
pseudojbb	8	24	9	37	7	42	7	74
average	9	20	9	25	6	45	6	52

Table 7. Percent memory savings from hybrid compression.

Table 7 shows memory savings from the hybrid compression techniques in Section 3.5. The savings from the maximal hybrid exceed the savings of any individual optimization, because it picks the best compression technique for each individual piece of data. The savings from the combined hybrid exceed those from the maximal hybrid, because each piece of data may be optimized by multiple techniques. But the additional savings of combined over maximal are low for non-arrays. Arrays afford much greater savings in general. These results suggest that JVM developers should focus compression optimizations more on arrays. Using models presented in this paper, we see that the combined hybrid for arrays is able to compress the heap by up to 39%. Adding the savings for both objects and arrays, we see we savings up to 46% of all memory, and 86% for application data. These results show that there is a lot of bloat and redundancy currently in the Java heap that could be exploited. Although we cannot reach the 83% (or 99% for applications) compression that bzip can achieve, we can achieve over half of the savings while still being able to access and update individual objects.

Overall, we see the most potential for space optimization with arrays. For our benchmarks, the majority of the heap size is taken up by arrays (43 to 78%), and our models show the greatest potential compression with arrays. Removing bytes that are zero is particularly effective, saving on average 45% of the heap and up to 73% for applications. Although previous researchers have analyzed many compression techniques, we are the first to apply many models successively to each piece of data in the heap. For application classes and arrays, on average we can compress the heap by 58% with our combined hybrid model. Our hybrid analysis shows great potential to reduce the heap bloat causing Java memory inefficiency.

5. Related work

Previous work either characterizes heap data without specifically studying compression, or focuses on specific compression techniques without attempting to be comprehensive.

Modeling and characterization. Mitchell and Sevitsky categorize fields by the role they play in an object (header, pointer, null, primitive), and categorize objects by the role they play in a data structure (head, array, entry, contained) [17]. These measures together with scaling formulas predict the heap data reductions of manual program changes. Whereas Mitchell and Sevitsky focus on providing human heap understanding, we focus on heap compression that can be performed in the JVM.

Dieckmann and Hölzle study object lifetimes, size, type, and reference density for the SPECjvm98 benchmarks [10]. In addition to these measures, Blackburn et al. study time varying heap, allocation, and lifetime behaviors of the SPECjvm98, SPECjbb2000, and DaCapo benchmarks [19]. They show that DaCapo is significantly richer in code and data resource utilization than SPEC, which is why we use DaCapo here. While these studies provide general insights on heap memory composition, we measure specific limits of heap data compression.

Compression techniques. This section offers an incomplete survey of compression techniques for executable images, object headers, code, and the virtual execution engine itself. Stephenson et al. use static analysis to perform bit-width compression on C programs before compiling those C programs to FPGAs [24]. Cooprider and

Regehr [9], Titzer [27], and Titzer and Palsberg [28] compress executable images for embedded chips. They apply bit-width compression and field value set indirection on pre-allocated static data, but not on dynamically allocated heap data. Bacon et al. [4] compress header "fields" (type, hash, lock, GC bits) with some of the techniques that our paper explores for non-header data. Ernst et al. [11], Evans and Fraser [13], and Pugh [18] compress executable code and class files. Titzer et al. reduce the footprint of a Java virtual machine [26].

Heap data compression. The following research implements specific heap data compression techniques. Ananian and Rinard use static analysis and an offline profiling run for constant field elision, field bit-width reduction, and dominant-value field hashing [1]. Appel and Gonçalves use generational garbage collection for deep-equal acyclic object sharing [2]. Chen et al. use compacting garbage collection for zero-based object compression and speculative trailing zero array trimming [8]. Shankar et al. use online program analysis to create a specializer that exploits heap constants in interpreters [23]. Zhang and Gupta use static analysis and an offline profiling run for field bit-width reduction [29]. Zilles uses speculative narrow allocation for character array bit-width reduction [30].

These approaches expose some of the real-world implementation challenges for compression. For example, relying on offline profiling and static analysis reduces applicability to languages like Java with dynamic class loading, reflection, and native code. Some techniques target only specialized domains, such as interpreters, English-language characters, or acyclic data. Speculative optimizations require a back-out mechanism when compressed data properties are violated and this mechanism must be thread-safe. These challenges and runtime overheads expose space-time tradeoffs that are particular to the application setting. We leave to future work the space-time tradeoff of particular compression implementations. We address here the limits of memory efficiency by measuring the impact and applicability of compression on a large number of benchmarks, thus enabling apples-to-apples comparisons.

Towards heap data compression. Whereas the above research overcomes some real-world challenges of heap data compression, the others propose optimizations without evaluating full implementations. Chen et al. simulate dominant-value field elision [6]. Shaham et al. hand-optimize benchmarks with object-level lifetime optimizations [22], and Chen et al. hand-optimize benchmarks with field-level lifetime optimizations [7]. Marinov and O'Callahan hand-optimize benchmarks with deep-equal object sharing [16]. We also explore the limits of compression techniques, but we go a step further by empirically comparing a wider variety and combinations of techniques.

6. Conclusion

Memory is expensive, yet Java applications often squander it. Based on Lempel-Ziv compression, we estimate that at least 73% of heap data is redundant and compressible. Previous work has suggested a variety of compression techniques to harness some of this redundancy in the form of space savings. We developed a methodology for evaluating the limits of such compression techniques. It consists of a heap data compressibility analysis along with over a dozen models for the savings potential of individual optimizations. Thus, we are the first to offer an apples-to-apples comparison of a large number of different heap data compression techniques. We show that significant space savings are possible, especially with array compression and combined hybrid techniques. We hope that optimizing this discovered heap bloat can make Java a space efficient, high productivity programming language.

Acknowledgments We would like to acknowledge the help of Steve Blackburn with experimental implementation, Maria Jump for assistance with heap dump generation, and Mike Bond and reviewers for helpful feedback on the writing of this paper. We would also like to thank Nick Mitchell and Gary Sevitsky for fruitful discussions on the problem of heap bloat.

References

[1] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Languages, Compiler, and Tool Support for Embedded Systems*, 2003.

[2] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, 1993.

[3] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] D. Bacon, S. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *European Conference for Object-Oriented Programming*, 2002.

[5] A. Cardon and M. Crochemore. Partitioning a graph in $o(|a| \log_2 |v|)$. *Theoretical Computer Science*, 1982.

[6] G. Chen, M. Kandemir, and M. J. Irwin. Exploiting frequent field values in Java objects for reducing heap memory reqirements. In *Virtual Execution Environments (VEE)*, 2005.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Field level analysis for heap space optimization in embedded Java environments. In *International Symposium on Memory Management*, 2004.

[8] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained Java environments. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[9] N. D. Cooprider and J. D. Regehr. Offline compression for on-chip RAM. In *Programming Language Design and Implementation*, 2007.

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

[11] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code compression. In *Programming Language Design and Implementation*, 1997.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE)*, 1999.

[13] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *Programming Language Design and Implementation*, 2001.

[14] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *International Conference on Software Engineering*, 2002.

[15] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *International Symposium on Memory Management*, 2002.

[16] D. Marinov and R. O'Callahan. Object equality profiling. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[17] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[18] W. Pugh. Compressing Java class files. In *Programming Language Design and Implementation*, 1999.

[19] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[20] J. B. Sartor, M. Hirzel, and K. McKinley. No bit left behind: The limits of heap data compression (extended version). Technical Report TR-08-17, The University of Texas at Austin, 2008.

[21] Semiconductor Industry Association. SIA world semiconductor forcast 2007–2010, Nov. 2007. http://www.sia-online.org/pre_release.cfm?ID=455.

[22] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Programming Language Design and Implementation*, 2001.

[23] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith. Runtime specialization with optimistic heap analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[24] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Programming Language Design and Implementation*, 2000.

[25] TIOBE Software. TIOBE programming community index, 2007. http://tiobe.-com.tpci.html.

[26] B. Titzer, J. S. Auerbach, D. F. Bacon, and J. Palsberg. The ExoVM system for automatic VM application reduction. In *Programming Language Design and Implementation*, 2007.

[27] B. L. Titzer. Virgil: Objects on the head of a pin. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2006.

[28] B. L. Titzer and J. Palsberg. Vertical object layout and compression for fixed heaps. In *Compilers, Architectures, and Synthesis for Embedded Systems*, 2007.

[29] Y. Zhang and R. Gupta. Compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction*, 2002.

[30] C. Zilles. Accordion arrays: Selective compression of unicode arrays in Java. In *International Symposium on Memory Management*, 2007.