Smaller Footprint for Java Collections

Joseph Gil* and Yuval Shimron

Department of Computer Science,
The Technion—Israel Institute of Technology,
Technion City, Haifa 32000, Israel
yogi@cs.technion.ac.il

Abstract. In dealing with the container bloat problem, we identify five memory compaction techniques, which can be used to reduce the footprint of the large number of small objects that make these containers. Using these techniques, we describe two alternative methods for more efficient encoding of the JRE's ubiquitous `HashMap` data structure, and present a mathematical model in which the footprint of this can be analyzed. The *fused hashing* encoding method reduces memory overhead by 20%–45% on a 32-bit environment and 45%–65% on a 64-bit environment. This encoding guarantees these figures as lower bound regardless of the distribution of keys in hash buckets. The more opportunistic *squashed hashing*, achieves expected savings of 25%–70% on a 32-bit environment and 30%–75% on a 64-bit environments, but these savings can degrade and are not guaranteed against bad (and unlikely) distribution of keys to buckets. Both techniques are applicable and give merit to an implementation of `HashSet` which is independent of that of `HashMap`. Benchmarking using the SPECjvm2008, SPECjbb2005 and DaCapo suites does not demonstrate significant major slowdown or speedup. For `TreeMap` we show two encodings which reduce the overhead of tree nodes by 43% & 46% on a 32-bit environment and 55% & 73% on a 64-bit environment. These also give to separating the implementation of `TreeSet` from that of `TreeMap`, which gives rise to footprint reduction of 59% & 54% on a 32-bit environment and 61% & 77% on a 64-bit environment.

1 Introduction

Java and the underlying virtual machine provide software engineers with a programming environment that abstracts over many hardware specific technicalities. The *runtime* cost of this abstraction is offset by modern compiler technologies, including just in time compilations [1, 5, 7, 13, 15, 19, 22]. Indeed, there are indications [5, 13, 19] that Java's time performance is approaching that of languages such as C++ and Fortran which execute directly on the hardware and are free of potential performance penalties incurred by automatic memory management.

In contrast, memory consumption is not an easy target for automatic optimization. The reason is that there is a direct mapping of programmer-defined

* Corresponding author.

J. Noble (Ed.): ECOOP 2012, LNCS 7313, pp. 356–382, 2012.

data structures to runtime layout. An optimizing compiler cannot easily alter programmer defined data structures without disturbing the program semantics. On the other hand, with the ever increasing use of JAVA for implementing servers and other large scale applications, evidence accumulates that the openhanded manner of using memory, which is so easy to resort to in JAVA, leads to memory bloat, with negative impacts on time performance, scalability and usability [17].

The research community response includes methods for diagnosing the overall "memory health" of a program and acting accordingly [18], leak detection algorithms [20], and methods for analyzing [16] profiling [27, 29] and visualizing [23] the heap. (See also a recent survey on the issue of both time and space bloat [28].)

This work is concerned primarily with what might be called "container bloat" (to be distinguished e.g., from temporaries bloat [9]), which is believed to be one of the primary contributors to memory bloat. (Consider, e.g., Mitchell & Sevitsky's example of a large on-line store application consuming $2.87 \cdot 10^9$ bytes, 42% of which are dedicated to class `HashMap` (OOPSLA'07 presentation).)

Previous work on the container bloat problem included methods for detecting suboptimal use of containers [30] or recommendations on better choice of containers based on dynamic profiling [24]. Our line of work is different in that we propose a more compact implementation of containers. Much in the line of work of Kawachiya, Kazunori and Onodera [14] which directly attacks bloat due to the `String` class, our work focuses on space optimization of collection classes, concentrating on `HashMap` and `HashSet`, and to a lesser extent on the `TreeMap` and `TreeSet` classes.

Our work *does not* propose a different and supposedly better algorithmic method for organizing these collections, e.g., by using open-addressing for hashing, prime-sized table, or AVL trees in place of the current implementations. Research taking this direction is rich, but our focus here is on the question of whether *given* data and data structures can be *encoded* more efficiently. After all, whatever method an efficient data structure is organized in, its compaction should lead to an even more frugal use of memory, just as subjecting a super fast algorithm to automatic optimization could improve it further.

To this end, we insist on full compatibility with the existing implementations, including e.g., preserving the order of keys in hash table, and the tree topology of `TreeMap`. Unlike Kawachiya et. al's work, we do not rely on changes to the JVM—the optimization techniques we describe here can be employed by application programmers not only to collections, but to any user defined data structure. Still, the employment of our compaction techniques for aggressive space optimization cannot in general be done without some familiarity with the underlying object model.

We are also motivated by the hope that the techniques we offer could serve optimizing compilers or be employed by other automatic tools for memory optimization (although it is clear that more research is required before all techniques discussed here can thus be exploited).

Table 1.1. Minimal no. of bytes per entry in a set and a map data structures

	32-bit	**64-bit**
Map	8	16
Set	4	8

To appreciate the scale of container overhead, note first that a simple information theoretical consideration sets a minimum of n references for *any* representation of a set of n keys, and $2n$ references for *any* representation of n pairs of key and values.

Table 1.1 summarizes these minimal values for 32-bits and 64-bits memory models, e.g., on a 64-bits memory model no map can be represented in fewer than 16 bytes per entry. Achieving these minimal values is easy if we neglect the time required for retrieval and the necessary provisions for updates to the underlying data structure: A set can be implemented e.g., as a compact sorted array, in which search is logarithmic, while updates consume linear time. The challenge is in an implementation which does not compromise search and update times. Despite recent theoretical results [3] by which one can use, e.g., $n + o(n)$ words for the representation of a dynamic set, while still paying constant time for retrievals and updates, this ideal seems far from being practical: Contemporary implementations of data structures are known to be tolerant of some memory overhead, but, the magnitude of this overhead may be surprising. Consider e.g., `java.util.TreeMap`, an implementation of a red-black balanced binary search tree, which serves as the JRE principal mechanism for the realization of a sorted map datastructure. Memory overheads incurred by this data structure are inferred by examining fields defined for each tree node, realized by the internal class `TreeMap.Entry`.

```
public class TreeMap<K, V> {
  //...
  static final class Entry<K, V>
    implements Map.Entry<K, V> {
      final K key;
      V value;
      Entry<K, V> left = null;
      Entry<K, V> right = null;
      Entry<K, V> parent;
      boolean color = BLACK;
    //...
  }
  //...
}
```

```
public class HashMap<K, V> {
  //...
  static class Entry<K, V>
    implements Map.Entry<K, V> {
      final K key;
      V value;
      Entry<K, V> next;
      final int hash;
    //...
  }
  //...
}
```

Fig. 1.1. Fields defined in `TreeMap.Entry` (a) and in `HashMap.Entry` (b)

Fig. 1.1(a) shows that on top of the object header, each tree nodes stores 5 pointers and a `boolean` whose minimal footprint is only 1 bit, but typically requires at least a full byte (and even an eight bytes word on e.g., the jikes virtual machines). On 32-bits implementation of the JVM which uses 8 bytes per object header and 4 bytes per pointer total memory for a tree node is $8 + 5 \cdot 4 + 1 = 29$ bytes. With the common 4-alignment or 8-alignment requirements (as found in e.g., the HotSpot32 implementation of the JVM), 3 bytes of padding must be added, bringing memory per entry to four times the minimum.

The situation is twice as bad in the implementation of the class `TreeSet` (of package `java.util`), the standard JRE method for realizing a sorted set.

Internally, this class is implemented as proxy to `TreeMap` with a dummy mapped value, bringing memory per entry to eight times the minimum.

A hash-table implementation of the `Map` and `Set` interfaces is provided by the JRE's class `java.util.HashMap` and its proxy class `java.util.HashSet`. Memory per entry of these is determined by two factors. First, as we will explain in greater detail below, each hash table entry consumes $4/p$ bytes (on a 32-bits architecture), where p is the table density parameter ranging typically between 3/8 and 3/4. Secondly, an object of the internal class `HashMap.Entry` (depicted in Fig. 1.1(b)) is associated with each such entry.

On HotSpot32, instances of `HashMap.Entry` occupy 24 bytes, bringing the memory requirements of each `HashMap` entry to the range of 29.33–34.67 bytes in typical ranges of p, i.e., within 10% of memory use with class `TreeMap`.

Table 1.2. Memory overhead per entry of common data structure in central JRE collections

	HotSpot32	HotSpot64
$\mathcal{L}$-`TreeMap`	24	48
$\mathcal{L}$-`TreeSet`	28	56
$\mathcal{L}$-`HashMap`	$16 + 4/p$	$32 + 8/p$
$\mathcal{L}$-`HashSet`	$20 + 4/p$	$40 + 8/p$

Table 1.2 summarizes the overhead, in bytes, for representing the four fundamental JRE collections we discussed. Note that the numbers in the table are of the excess, i.e., bytes beyond what is required to address the key (and value, if it exists) of the data structure.

We describe a tool chest consisting of five memory compaction techniques, which can be used to reduce the footprint of the small `Entry` objects that make the `HashMap` and `TreeMap` containers. These are: null pointer elimination, boolean elimination, object fusion, field pull-up and field consolidation. Techniques can be applied independently, but they become more effective if used wisely together, with attention to the memory model and to issues such as alignments.

Using these techniques, we describe *fused hashing* ($\mathcal{F}$-hash henceforth) and *squashed hashing* ($\mathcal{S}$-hash henceforth): two alternative methods for more efficient encoding of the JRE's implementation of `HashMap` data structure. Fusion and squashing are extended to `TreeMap` as well. Our compaction gives also reason to separating the implementation of `HashSet` from `HashMap` and `TreeSet` from that of `TreeMap`.

We present a mathematical model in which the footprint of these implementations can be analyzed. In this model, we deduce that $\mathcal{F}$-`HashMap` reduces memory overhead by 20%–34% on a 32-bit environment and 48%–54% on a 64-bit environment.

Timing results indicate that no significant improvement or degradation in runtime is noticeable for in three common JVM benchmarks: SPECjvm2008, SPECjbb2005 and DaCapo. Naturally, some specific map operations are slowed down in compare to the simple base implementation due to a more complex and less straightforward implementation.

Table 1.3 summarizes the savings of $\mathcal{F}$-hash, $\mathcal{S}$-hash, $\mathcal{F}$-tree and $\mathcal{S}$-tree. A fully compatible `Map` implementation of the 32-bit and 64-bit versions of $\mathcal{F}$-`HashMap`,

Table 1.3. Memory overhead reduction for a common `Map` instance; implementation of the marked entries is publicly available

		Hash		Tree	
		Fused	*Squashed*	*Fused*	*Squashed*
Map	32-bit	20%–34%†	26%–53%†	43%†	46%†
	64-bit	48%–54%†	32%–56%†	55%†	73%†
Set	32-bit	22%–38%	58%–67%†	59%	54%
	64-bit	50%–62%	64%–73%†	61%	77%

$\mathcal{S}$-`HashMap`, $\mathcal{S}$-`HashSet`, $\mathcal{F}$-`TreeMap` and $\mathcal{S}$-`TreeMap` can be found on the first author's website.These implementations were thoroughly checked correct by the JRE test suite and our own additional testing.

The programming labor in producing the implementations inspired us to present an instrument *virtual entries* that enables transparent inspection of compacted data structures.

A mathematical model was developed for computing the expected savings as a function of the hash table density. This model provides a lower bound on the savings of fused hashing; should the distribution of keys into hash buckets be not as even as the distribution of balls thrown at random into urns, then the actual savings may be greater. For squashed hashing, the model yields an expected value. A non-fully random distribution of keys into buckets may improve, but sometimes worsen the reported savings.

For trees, the balancing algorithms make the reported savings valid for tree with a handful of keys created in any order of key insertions and removals.

A similar analytical approach would not be informative for evaluating time performance. The reason is that the alternative implementations were designed to have the same underlying structure as $\mathcal{L}$-hash: the number of comparisons required to find a key κ is the same in all implementations. Therefore, to evaluate time performance we conducted benchmarking; these were carried out using HotSpot32 (exact settings are described below).

We conjecture that searches in fused and squashed hash tables should be *faster* than the baseline (due to fewer memory dereferencing operations), but that insertions and removals are slower (as they involve repacking of small objects). Initial experiments, not reported here, confirm this conjecture. However, a detailed benchmark timing operations as a function of table density and table size is left for further work, given the intriguing results we, in cooperation with Lenz found [12]. In particular, it was demonstrated that the so called "steady state" which is supposedly reached after sufficient warm-up is not as steady as one might think, with results fluctuating between multiple steady states.

Worse, it was found that the timing of an operation depends on code executed prior to the benchmark. For this reason, the reported benchmarking here focuses on the performance of the compacted data structures as part of a client benchmark application.

Applications using sorted maps and sets are not abundant. Our initial timing of squashed tree data structures indicate that it is as fast, and sometimes

faster than the baseline. However, fused trees are about two times slower; their use should probably be limited to applications operating under strict memory constraints in which either time performance is not a factor, or tree operations are rare.

Outline. The remainder of this article is organized as follows. The five memory compaction techniques we identified are described in Section 2. Sec. 3 then reviews the JRE's implementation class `HashMap`, highlighting the locations in which the optimization can take place based on the statistical properties of distribution of keys into hash table cells. $\mathcal{F}$-hash and $\mathcal{S}$-hash are then described (respectively) in sects. 4 and 5. In Sect. 6 we explain how virtual entries are implemented. Time performance of the compacted hash tables is the subject of Sect. 7. Sect. 8 describes our fused and squashed versions of `TreeMap` and `TreeSet`. Sect. 9 concludes.

2 Compaction Techniques

The space compaction techniques that we identify include the following three:

- *Null pointer elimination.* Say a class C defines an immutable pointer field `p` which happens to be `null` in many of C's instances. Then, this pointer can be eliminated from C by replacing the data member `p` with a non-`final` method `p()` which returns `null`. This method is overridden in a class C_p inheriting from C, to return the value of data member `p` defined in C_p. Objects with `null` values of `p` are instantiated from C; all other objects instantiate C_p.
- *Boolean elimination.* A similar rewriting process can be used to eliminate immutable boolean fields from classes. A boolean field in a class C can be emulated by classes C_t (corresponding to `true` value of the field) and C_f (corresponding to `false`), both inheriting from C.

  In a sense, both null-pointer elimination and boolean elimination move data from an object into its header, which encodes its runtime type. Both however are applicable mostly if class C does not have other subclasses, and even though they might be used more than once in the same class to eliminate several immutable pointers and booleans, repeated application will lead to an exponential blowup in the number of subclasses.

  Mutable fields may also benefit from these techniques if it makes sense to recreate the instances of C should the eliminated field change its value.
- *Object fusion.* Say that a class C defines an ownership [6] pointer in field of type C', then all fields of type C' can be inlined into class C, eliminating the $C \to C'$ pointer. Fusion also eliminates header of the C' object, and the back pointer $C' \to C$ if it exists. It is often useful to combine fusion with null-pointer elimination, moving the fields of C' into C, only if the pointer to the owned object is not `null`.

Before describing the two additional techniques, a brief reminder of JAVA's object model is in place. Unlike C++, all objects in JAVA contain an *object header*, which encodes a pointer to a dynamic dispatch table together with synchronization, garbage collection, and other bits of information. In the HotSpot implementation of the JVM, this header spans 8 bytes on HotSpot32, and 16 bytes on HotSpot64 (but other sizes are possible [4], including an implementation of the JVM in 64-bit environment in which there is no header at all [25]).

The mandatory object header makes fusion very effective. In C++, small objects would have no header (`vptr` in the C++ lingo [10]), and fusion in C++ would merely save the inter-object pointer.

Following the header, we find data fields: `long` and `double` types span 8 bytes each, 4 bytes are used for `int`, 2 bytes for `char` and `short` and 1 byte for types `byte` and `boolean`. References, i.e., non-primitive types take 4 bytes on HotSpot32, and 8 bytes on HotSpot64. Arrays of length m occupy ms bytes, up-aligned to the nearest 8-byte boundary, where s is the size of an array entry. Array headers consume 12 bytes in a 32-bit JVM , 8 for header and 4 for the array `length` field (20 bytes in a 64-bit JVM). Finally, all objects and sub-objects are aligned on an 8-bytes boundary.[1]

Both the header and alignment issues may lead to significant bloat, attributed to what the literature calls *small objects*. Class `Boolean` for example, occupies 16 bytes on Hotspot32 (8 for header, one for the `value` field, and 7 for alignment), even though only one bit is required for representing its content. Applying boolean elimination to `Boolean`, i.e., by making class `Boolean` abstract, while introducing singleton classes `True` and `False` which extend it, would halve the footprint of all `Boolean` objects.

Alignment issues give good reasons for applying the space compaction techniques together. Applying null pointer elimination to class `HashMap.Entry` would not decrease its size (on HotSpot32); one must remove yet another field to reach the minimal saving quantum of 8 bytes per entry.

We propose two additional techniques for dealing with waste due to alignment:

– *Field Pull-up*. Say that a class C' inherits from a class C, and that class C is not fully occupied due to alignment. Then, fields of class C' could be predefined in class C, avoiding alignment waste in C‘, in which the C' subobject is aligned, just as the entire object C. We employ field pull up mostly for smaller fields, typically `byte` sized.

  In a scan of some $20,000$ classes of the JRE, we found that the footprint of 13.6% of these could be reduced by 8 bytes by applying greedy field pullup, while 1.1% of the classes would lose 16 bytes. (Take note that field pullup could be done by the JVM as well, in which case, fields of different subclasses could share the same alignment hole of a superclass, and that the problem of optimizing pullup scheme is NP-complete.)

[1] See more detailed description in
`http://kohlerm.blogspot.com/2008/12/`
`how-much-memory-is-used-by-my-java.html` or
`http://www.javamex.com/tutorials/memory/object_memory_usage.shtml`

– *Field Consolidation.* Yet another technique for avoiding waste due to alignment is by consolidation: instead of defining the same field in a large number of objects, one could define an array containing the field. If this is done, the minimal alignment cost of the array is divided among all small objects, and can be neglected.

  Of course, consolidation is only effective if there is a method for finding the array index back from the object whose field was consolidated.

Object fusion was also called *object inlining* in the literature and used for *automatic* optimization of JAVA programs (see Wimmer's Ph.D. thesis [26] for a survey). We suspect that the other techniques enumerated above were employed by programmers without identifying their universality.

3 Hash Tables of the JRE

Other than the implementations designed for concurrent access, we find three principal implementations of hash tables in the JRE: class `IdentityHashMap` uses open-addressing combined with linear probing, i.e., all keys and values are stored in an array of size m, and a new key κ is stored in position $(H(\kappa) + j \bmod m)$ where $H(\kappa)$ is the hash value of κ and j is the smallest integer for which this array position is empty. Class `HashMap` (henceforth called $\mathcal{L}$-`HashMap`) uses *chained hashing* method, by which the i^{th} table position contains a *bucket* of all keys κ for which $H(\kappa) \bmod m = i$. This bucket is modeled as a singly-linked list of nodes of type `HashMap.Entry` (depicted in Fig. 1.1(b)). The hash table itself is then simply an array `table` of type `HashMap.Entry[]`. Finally, class `HashSet` is a wrapper of `HashMap`, delegating all set operations to `map`, an internal `private` field of type `HashMap` that maps all keys in the set to some fixed dummy object value.

It is estimated[2] that class `IdentityHashMap` is 15% to 60% faster than `HashMap`, and occupies around 40% smaller footprint. Yet class `IdentityHashMap` is rarely used[3] since it breaks the `Map` contract in comparing keys by identity rather than the semantic `equals` method. One may conjecture that open addressing would benefit `HashMap` as well. However, Lea's judgment of an experiment he carried in employing the same open addressing for the general purpose `HashMap` was that it is not sufficiently better to commit.

Function H is realized in `HashMap` as `hash(key.hashCode())` where function `hash` is as in Fig. 3.1. This function's purpose is to improve those overridden versions of the `hashCode()` method in which some of the bits returned are less random than others. This correction is necessary since m, the hash table's size, is selected as a power of two, and the computation of $H(\kappa) \bmod m$

```
static int hash(int h) {
 h ^= h >>> 20 ^ h >>> 12;
 return h ^ h >>> 7 ^ h >>> 4;
}
```

Fig. 3.1. Bit spreading function implementation from `HashMap` class

[2] `http://www.mail-archive.com/core-libs-dev@openjdk.java.net/msg02147.html` `msg02147.html`

[3] `http://khangaonkar.blogspot.com/2010/06/` `what-java-map-class-should-i-use.html`

is carried out as a bit-mask operation. With the absence of this "bit-spreading" function, implementation of `hashCode` in which the lower-bits are not as random as they should be would lead to a greater number of collisions.

Class `HashMap` caches, for each table entry, the value of H on the key stored in that entry. This *Cached Hash Value* (CHV) makes it possible to detect (in most cases) that a searched for key is not equal to the key stored in an entry, *without* calling the potentially expensive `equals` method in function `hash`.

```
public V get(Object κ) {
  if (κ == null)
    return getForNullKey();
  int h = hash(κ.hashCode());
  for (
    Entry<K, V> e = table[h & table.length-1];
      e != null; e = e.next) {
        Object k;
        if (e.h == h
          && ((k = e.K) == κ || κ.equals(k)))
           return e.V;
  }
  return null;
}
```

Fig. 3.2. JAVA code for searching in $\mathcal{L}$-`HashMap`

Function `get` (Fig. 3.2) demonstrates how this CHV field accelerates searching: Before examining the key stored in an entry, the function compares the CHV of the entry with the hash value computed for the searched key. (The listing introduces the abbreviated notation K, V, and h for fields `key`, `value` and `hash`, to be used henceforth.)

A `float` typed parameter known by the name `loadFactor` governs the behavior of the hash table as it becomes more and more occupied. Let n denote the number of entries in the table, and let $p = n/m$. That is what we shall henceforth call *table density*. Then, if p exceeds the `loadFactor`, the table size is doubled, and all elements are rehashed using the CHV. It follows that (after first resize, with the absence of removals), $\texttt{loadFactor}/2 < p \leq \texttt{loadFactor}$. The default value of `loadFactor` is `0.75`, and it is safe to believe [8] that users rarely change this value, in which case, $3/8 < p \leq 3/4$, is an equality we shall call the *typical range of* p, or just the "typical range". The *center of the typical range*, $p = (3/8 + 3/4)/2 = 9/16$ is often used in benchmarking as a point characterizing the entire range.

The memory consumed by the `HashMap` data structure (sans content), can be classified into four kinds:

1. *class-memory.* This includes memory consumed when the class is loaded, but before any instances are created, including `static` data fields, memory used for representing methods' bytecodes, and the reflective `Class` data structure.
2. *instance-memory.* This includes memory consumed regardless of the hash table's size and the number of keys in it, e.g., scalars defined in the class, references to arrays, etc.
3. *arrays-memory.* This includes memory whose size depends solely on the table size.
4. *buckets-memory.* This includes memory whose size depends on the number of keys in the table, and the way these are organized.

Our analysis ignores the first two categories, taking note, for the first category that some of its overheads are subject of other lines of research [21], and for the second, that applications using many tiny maps probably require a conscious optimization effort, which is beyond the scope of this work.

We now compute the *memory use per entry*, i.e., the total memory divided by the number of entries in the table. On HotSpot32 array `table` consumes $4m$ bytes for the array content along with 12 bytes for the array header, which falls in the "instance-memory" category and thus ignored. These $4m$ bytes are divided among the n entries, contributing $4/p$ bytes per entry.

Examining Fig. 1.1(b), we see that each instance of class `Entry` has 8 bytes per header, 3 words for the K, V, and `next` pointers and another word for the CHV, totaling in $8 + 4 \cdot 4 = 24$ bytes per object. The number of bytes per table entry is therefore $24 + 4/p$. For HotSpot64, the header is 8 bytes, the 3 pointers are 8 bytes each and the integer CHV is 4 bytes, which total, thanks to alignment, is $48 + 8/p$ bytes per object. (Comparing these values with Table 1.1 gave the memory overheads of $\mathcal{L}$-`HashMap` and $\mathcal{L}$-`HashSet`, as tabulated in Table 1.2.) Observe that the decision to implement `HashSet` as `HashMap` does not incur any memory toll: eliminating the `value` field gives the same number of bytes per `Entry` object (at least on HotSpot32).

Hashing can be modeled by the famous "balls into urns" statistical model [11], which gives rise to the Poisson distribution: The fraction of hash table buckets with precisely k keys is $\frac{p^k}{k!}e^{-p}$ where $p = n/m$, n being the total number of keys and m the total number of buckets. Fig. 3.3 plots these fractions for $k = 0, \ldots, 4$. The endpoints of the typical range as well as its center are shown as vertical blue lines in the figure. We see that in the typical range a significant portion of the buckets are empty, ranging from 69% (maximal value), to 47% (minimal value). Even when $p = 1$, 37% of the buckets are empty.

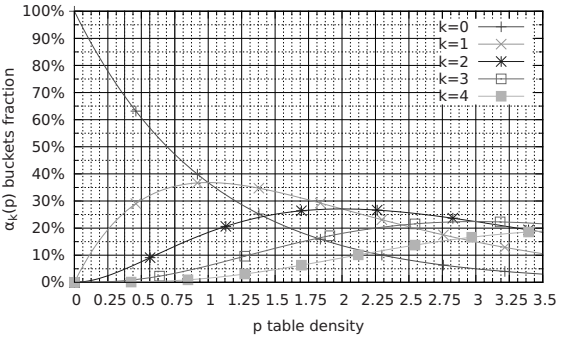

Fig. 3.3. Expected fraction of buckets of size k, $k = 0, \ldots, 4$ vs. table density (and expected fraction of keys falling in buckets of size $k + 1$)

As it turns out, the expected fraction of *keys* which fall into buckets of size $k > 1$, is nothing but the expected fraction of *buckets* of size $k - 1$. We can therefore read the fraction of keys falling into buckets of size k by inspecting the $(k-1)^{th}$ curve in Fig. 3.3. In the range of $p = 3/8$ through $p = 3/4$, we have that the fraction of keys in buckets of size 1 is the greatest, ranging between 69% and 47%. The fraction of keys in buckets of size 2 is between 26% and 35%. At $p = 3/4$, fewer than 15% of the keys fall in buckets of size 3, and, fewer than 4% of the keys are in buckets of size 4.

We identify several specific space optimization opportunities in the $\mathcal{L}$-`HashMap` implementation: First, cells of array `table` which correspond to empty buckets are always `null`. Second, in the list representation of buckets, there is a `null` pointer at the end. A bucket of size k divides this cost among the k keys in it. The greatest cost for key is for singleton buckets, which constitute 47%–69% of all keys in the typical range.

These two opportunities were called "pointer overhead" by Mitchell and Sevitsky [18]. Our fusion and squashing hashing deal with the second overhead (*pointer overhead/entry* in the Mitchell and Sevitsky terminology) but not the first (*pointer overhead/array*): The number of empty buckets is determined solely by p and we see no way of changing this.

Non-null pointers (*collection glue*) are those `next` pointers which are not `null`. These occur in buckets with two or more keys and are dealt with using fusion and squashing. The *small objects overhead* in `HashMap` refers to the fact that *each* `Map.Entry` object has a header, whose size (on HotSpot32) is the same as the essential $\langle K, V\rangle$ pair stored in an entry.

The CHV, the fourth (and last) field of class `HashMap.Entry` (Fig. 1.1(b)), is classified as *primitive-overhead* in the GM taxonomy [18, footnote 4], and can be optimized as well: If $m = 2^\ell$, then the ℓ least significant bits of all keys that are hashed into a bucket i, are precisely the number i. The remaining $32 - \ell$ most significant bits of the CHV are the *only* meaningful bits in the comparison of keys that fall into the same bucket.

We found that storing a `byte` instead of an `int` for the CHV has minimal effect on runtime performance, eliminating 255/256 of failing comparisons. Best results were found for a CHV defined by the coercion `(byte) hash1(key.hashCode())`. where `hash1` is the first stage in computing `hash` (see Fig. 3.4).

```
static int hash1(int h) {
   return h ^ h >>> 20;
}
static int hash2(int h) {
   h ^= h >>> 12;
   return h ^ h >>> 7 ^ h >>> 4;
}
```

Fig. 3.4. Two steps hash code modification function implementation

4 Fused Buckets Hashing

Employing list fusion and pointer-elimination for the representation of a bucket calls for a specialized version of `Entry` for buckets of size k, $k = 1, \ldots, \ell$, for some small integer constant ℓ. The naïve way of doing so is not too effective since in singleton buckets (which form the majority of buckets in the typical range), the specialized entry should include two references (to the key and value) as well as the CHV. With 8-byte alignment, the size of a specialized `Entry` for a singleton bucket is the same as that of the unmodified `Entry`.

Instead, our fused-hashing implementation *consolidates* the CHV of the first key of *all* non-empty buckets into a common array `byte[] chv` of length m, which parallels the main `table` array. If the $\mathtt{i}^{th}$ bucket is empty, then `table[i]` is `null` and `chv[i]` is undefined. Otherwise, `chv[i]` is the CHV of the first key in the $\mathtt{i}^{th}$ bucket, and `table[i]` points to a `Bucket` object that stores the bucket's

contents: for $\mathcal{F}$-`HashMap` this includes all triples $\langle K_j, V_j, h_j \rangle$ that belong in this bucket, with the exception of h_1; for $\mathcal{F}$-`HashSet`, the bucket contents includes all pairs $\langle K_j, h_j \rangle$, with the exception of h_1.

We define four successive specializations of the `abstract` class `Bucket`: `Bucket1` represents singleton buckets and `extends` class `Bucket`; `Bucket2` that `extends` `Bucket1` represents buckets of size 2; `Bucket3` that `extends` `Bucket2`, is dedicated for buckets of size 3; finally, buckets of size 4 or more are represented by class `Bucket4` which `extends` class `Bucket3`. We thus *fuse* buckets of up to four entries into a single object, and employ pointer elimination in buckets of size $k = 1, 2, 3$. In larger buckets, every four consecutive entries are packed into a single object: buckets of size $k > 4$ consist of a list of $\lfloor k/4 \rfloor$ objects of type `Bucket4`. If k is divisible by 4, then the `next` pointer of the last `Bucket4` object of this list is `null`. Otherwise, this `next` field points to a `Bucket`k' object which represents the remaining k' entries in the bucket, where $1 \leq k' \leq 3$ is determined by $k' = k \bmod 4$.

As shown in Table 4.1 in the inheritance chain of `Bucket`, `Bucket1`, ..., `Bucket4` each class adds the fields required for representing buckets of the corresponding length; field pull-up (which depends on the memory model) is employed to avoid wastes incurred by alignment.

Table 4.1. Layout of fused bucket variants in $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` on HotSpot32 and HotSpot64

	HotSpot32				HotSpot64			
	$\mathcal{F}$-`HashMap`		$\mathcal{F}$-`HashSet`		$\mathcal{F}$-`HashMap`		$\mathcal{F}$-`HashSet`	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8	object header	16	object header	16
`Bucket1`	K_1, V_1	16	$K_1, \uparrow h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	16	K_1, V_1	32	$K_1,$	24
`Bucket2`	$K_2, V_2, \uparrow K_3, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	32	$K_2, \uparrow K_3$	24	$K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	56	$K_2 \; h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	40
`Bucket3`	$V_3, \uparrow K_4$	40		24	K_3, V_3	72	K_3	48
`Bucket4`	V_4, `next`	48	K_4, `next`	32	$K_4, V_4,$ `next`	96	K_4, `next`	64

For each class, the table shows the introduced fields along with fields pulled-up into it (such fields are prefixed by an up-arrow). The set of fields present in a given class is thus obtained by accumulating the fields introduced in it and all of its ancestors, shown as former table rows. Concentrating on HotSpot32 we see that class `Bucket2` in $\mathcal{F}$-`HashMap` introduces three fields: K_2, V_2 and h_2, but also includes fields K_3 and h_3 which were pulled-up from `Bucket3`, and field h_4 which was pulled-up from `Bucket4`. Class `Bucket2` includes also a h_5 field, which is the CHV of the first key in the subsequent `Bucket` object (or undefined if no such object exists.) The layout of buckets in $\mathcal{F}$-`HashSet`, is similar, except that the absence of value fields increases field pull-up opportunities, leading to greater memory savings.

The "total size" columns in the table suggest that $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` are likely to be more memory efficient than $\mathcal{L}$-hash, e.g., a bucket of size 4 that

requires 96 bytes in $\mathcal{L}$-hash is represented by 48 bytes in $\mathcal{F}$-`HashMap` and only 32 bytes in $\mathcal{F}$-`HashSet`. More importantly, the bulk of the buckets, that is singleton buckets, require only 16 bytes (32 bytes on HotSpot64), as opposed to the 24 bytes (respectively 48 bytes) footprint in the $\mathcal{L}$-hash implementation.

Naturally, objects consume more memory in moving from a 32-bits memory model to a 64-bits model. However, examining the righthand side of Table 4.1 shows that this increase is not always as high as two fold, e.g., a `Bucket1` object doubles up from 16 bytes to 32 bytes in $\mathcal{F}$-`HashMap` but only to 24 bytes in $\mathcal{F}$-`HashSet` (50% increase), and a `Bucket2` object increases from 32 bytes to 56 bytes in $\mathcal{F}$-`HashSet` and from 24 bytes to 40 bytes in $\mathcal{F}$-`HashSet` (both increases are by 67%).

An important property of fusion is that of *non-decreasing compression*, i.e., the number of bytes used per table entry decreases as the bucket size increases. On HotSpot32, overhead per entry in buckets of size 1,2,3,4 is 16, 16, 13.33, 12 bytes in `FHashMap` and 16, 12, 8, 8 in `FHashSet`. In the HotSpot64 model, the respective numbers are 32, 28, 24, 24 bytes in `FHashMap` and 24, 20, 16 ,16 bytes in `FHashSet`. It is easy to check that this property is preserved even in longer buckets.

A search for a given key κ in a fused bucket is carried out by comparing κ with fields $K_1, K_2, \ldots$ in order, and if $\kappa = K_i$ returning V_i, except that K_i is to be accessed if the current `Bucket` object is of type $\texttt{Bucket}_i$ or a subtype thereof. It is natural to implement this restriction by overriding function `get` in each of the `Bucket` classes. We found however that, in this trivial inheritance structure, dynamic dispatch is slightly inferior to the direct application of JAVA's `instanceof` operator to determine the bucket's dynamic type. Fig. 4.1 shows some of the details.

```
public V get(Object κ) {
 int h = hash1(κ.hashCode());
 int i = hash2(h) &
  table.length-1;
 Bucket1<K, V> b1 = table[i];
 if (b1==null)
  return null; // Empty bucket
 h = (byte) h;
 Object k;
 if (chv[i]==h && ((k = b1.K1)==κ
  || κ.equals(k)))
   return b1.V1;
 if (!(b1 instanceof Bucket2))
   return null;
 Bucket2<K, V> b2 = (Bucket2)b1;
 //...
  if (b4.h4==h && ((k = b3.K4)==κ
   || κ.equals(k)))
    return b3.V4;
  return b4.next==null
         ? null
        : b4.next.get(h, κ, b4.h5);
}
```

Fig. 4.1. JAVA code for searching a key in a fused bucket

In comparing with of $\mathcal{L}$-hash in Fig. 3.2, we see that the first iterations of the loop are unrolled: the search begins with an object b_1 of type `Bucket1`; if κ, the searched key, is different from the K_1 field of b_1, we check whether the current bucket is of type `Bucket2`, in which case, b_1 is down casted into type `Bucket2`, saving the result in b_2, proceeding to examining field K_2, etc. As before the CHV fields $(h_1, h_2, \ldots)$ are used to accelerate the search, and as before an identity comparison precedes the call to the potentially slower `equals` method.

It is generally believed that search operations, and in particular, successful search, are the most frequent operations on collections.[4] This is the reason we did not try to similarly optimize insertions (method `put`) and removals (method `remove`); these were implemented by dynamic dispatch into appropriate methods in the `Bucket` hierarchy.

We turn to space analysis. For that we employ an analytical model to compute the expected number of bytes per table entry. This kind of analysis is justified by the fact that empirical results generally agree with the theoretical model of buckets' distribution: This is true for the initial implementation of function `hashCode` in class `Object`, which on HotSpot is by a pseudo-random number generator. Class designers, particularly of common library classes such as `String`, usually make serious effort to make `hashCode` as randomizing as possible. It is also known [2] that the design of a particularly bad set of distinct hash values is difficult. Finally, the bit-spreading preconditioning of function `hash` (Fig. 3.1), compensates for suboptimal overriding implementations of `hashCode`.

Note that a distribution of keys among buckets which is not random means that buckets tend to be fewer and larger than that predicted by the Poisson distribution. The "non-decreasing compression" property of $\mathcal{F}$-hash guarantees that the analytical model is a lower-bound on the memory savings which can only be larger in practice. For example, in the extreme case in which function `hashCode()` always returns 0, all entries will fall in the first bucket, each map entry will consume only 12 bytes. (The same lower bound could not be stated for $\mathcal{S}$-`HashMap` in which the non-decreasing compression property does not hold.)

A detailed analysis of the space overhead reduction is given in the appendix, for $\mathcal{F}$-`HashMap` on HotSpot32, the expected number of bytes per table entry is

$$12 + \left(9 - 4\cos p \cdot e^{-p}\right)/p.$$

We see that throughout the entire "typical" range, list fusion improves memory use for the hash data structure, reducing it by about a third at $p = 3/4$, and that the improvement increases with p. As it turns out, fusion improves upon the baseline representation throughout the entire typical range as reported above in Table 1.3. Further, this improvement increases monotonically with p; for slightly larger values of p (e.g., in load factor $p \approx 1.5$ in which buckets are still very small) both $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` are close to their asymptotic utility, requiring just a little over 12 bytes of overhead ($\mathcal{F}$-`HashMap`) and just little over 8 bytes of overhead ($\mathcal{F}$-`HashSet`), thus reaching a two (three) fold improvement over the 24 bytes of overhead in $\mathcal{L}$-hash.

The same behavior is exhibited by the 64 bit memory model: significant improvement in the typical range (which surpasses that seen in the 32 bit model), and reaching the same two- or three- fold improvement for larger values of p.

[4] See for example discussion in `http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-June/001807.html`

5 Squashed Buckets Hashing

Squashed buckets hashing further reduces the footprint of fused. The enabling observation is that a singleton bucket does not need to be represented by an object. Consider a cell in array `table` data structure, whose associated bucket is a singleton. Instead of storing in the cell a reference to a singleton bucket object, squashing means that the cell references the *key* residing at this bucket, while the *value* is consolidated into a table-global array. Thus, a hash map consists of three arrays of length m: `keys` and `values` of type `Object`, and, as before, array `chv` of `bytes` storing the CHV of the first key in buckets.

If the $\texttt{i}^{th}$ bucket is empty, then `keys[i]` and `values[i]` are `null`. If it is a singleton, `keys[i]` is the key stored in this bucket, `values[i]` is the associated value, while `chv[i]` is the CHV of this key. Otherwise, bucket `i` has k entries for some $k \geq 2$. In this case, cell `keys[i]` references a `Bucket` object, which must store all triples $\langle K_j, V_j, h_j \rangle$, for $j = 1, \ldots, k$, that fall in this bucket, except that V_1 is stored in `values[i]`, and h_1 is stored in `chv[i]`.[5]

As before, the `Bucket` object is represented using list fusion: class `Bucket2` (which `extends` the `abstract` class `Bucket`), stores the fused triples list when the bucket is of size 2; class `Bucket3`, which `extends` class `Bucket2`, stores the fused triples list when the bucket is of size 3, etc. Class `Bucket6`, designed for the rare case in which a bucket has 6 keys or more, stores the first *five* triples and a reference to a linked list in which the remaining triples reside. For simplicity, we use standard `Entry` objects to represent this list. (A little more memory could be claimed by using, as we did for $\mathcal{F}$-hash, one of `Bucket2`, ..., `Bucket6` for representing the bucket's tail; this extra saving is minute.)

A squashed `HashSet` is similar to a squashed `HashMap`, except for the obvious necessary changes: There is no `values` array, and a `Bucket` object for a k-sized bucket stores pairs $\langle K_i, h_i \rangle$, for $i = 1, \ldots, k$ (h_1 is still stored in `chv[i]`).

Table 5.1 lists the introduced -and pulled-up- fields in classes `Bucket`, `Bucket2`, ..., `Bucket6` in $\mathcal{S}$-`HashSet` and $\mathcal{S}$-`HashMap` on HotSpot32 and HotSpot64.

Of the twenty concrete classes described in the table, only four consume unused space: `Bucket6` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` and `Bucket2` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`. The global waste due to the first two classes is meager since buckets with six keys or more are rare, and the waste is divided among all keys in the bucket. However, the effectiveness of squashed hashing on HotSpot64 in somewhat limited by the waste in buckets of size 2.

Observe that since singleton buckets do not occupy any memory, the non-decreasing compression property of fused hashing is not preserved. In other words, unlike fusion, a key distribution in which all keys fall in distinct buckets is the most memory efficient among all other distributions, and when more keys are added to a bucket it does not necessarily become more efficient in reducing the memory overhead per key.

[5] Squashed hashing does not allow keys whose type inherits from class `Bucket`; this is rarely a limitation as this class is normally defined as an inner `private` class of `HashMap`.

Table 5.1. Layout of squashed bucket variants in $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` on HotSpot32 and HotSpot64

	HotSpot32				HotSpot64			
	$\mathcal{S}$-`HashMap`		$\mathcal{S}$-`HashSet`		$\mathcal{S}$-`HashMap`		$\mathcal{S}$-`HashSet`	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8	—	16	—	16
`Bucket2`	$K_1, K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	24	$K_1, K_2, \uparrow K_3, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	24	$K_1, K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	48	$K_1, K_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	40
`Bucket3`	K_3, V_3	32		24	K_3, V_3	64	K_3	48
`Bucket4`	K_4, V_4	40	$K_4, \uparrow K_5$	32	K_4, V_4	80	K_4	56
`Bucket5`	K_5, V_5	48		32	K_5, V_5	96	K_5	64
`Bucket6`	`next`	56	`next`	40	`next`	104	`next`	72

A search for a given key κ in a squashed bucket is carried out by comparing κ with fields $K_1, K_2, \ldots$ in order, and if $\kappa = K_i$ returning V_i. Unlike fused buckets, this search cannot be implemented solely by dynamic dispatch since no `Bucket` object exists for singleton buckets. Our implementation (Fig. 5.1) deals with singleton buckets by overriding the `equals` method of `Bucket`; the alternative of using `instanceof` and then checking for equality, is possible, but unlikely to be as efficient.

```
public V get(Object κ) {
 int h = hash1(κ.hashCode());
 int i = hash2(h) &
  keys.length - 1;
 h = (byte) h;
 Object k = keys[i];
 if (k==null) return null;
 if (chv[i]==h && (k==κ
   || k.equals(κ)))
    return values[i];
 if (!(k instanceof Bucket2))
  return null;
 Bucket2<K, V> b = (Bucket2) k;
 if (b.h2==h &&
  ((k = b.K2)==κ || κ.equals(k)))
   return b.V2;
 if (!(b instanceof Bucket3))
  return null;
 Bucket3<K, V> b = (Bucket3) b;
 //...
}

class Bucket2<K, V> extends Bucket<K, V> {
  K K1, K2;
  V V2
  byte h2, h3, h4, h5;
  //...
  @Override final boolean equals(Object κ) {
    return K1==κ || K1.equals(κ);
  }
  //...
}
```

Fig. 5.1. JAVA code for searching a given key in a squashed bucket

After computing the index `i` and the CHV value h, the search begins by considering the case of equality with K_1, which is done by comparing `keys[i]` with κ, and if these two are equal, `values[i]` is returned. In case of non-singleton bucket, the call k.`equals`(κ) invokes method `equals` of `Bucket2`. This virtual function call is made only if $h == h_1$. The search continues with a check whether a longer bucket resides in `keys[i]` by checking whether k is an `instanceof` class `Bucket2`, in which case we proceed to comparing κ with field K_2, etc.

The implementation of insertions and removals relied on a special case treatment of singleton buckets and dynamic dispatch of all other buckets.

A detailed analysis of the space overhead reduction is given in the appendix, e.g., it is shown that for $\mathcal{S}$-`HashMap` on HotSpot32, the expected bytes overhead per table entry is

$$24 - \left(55 + e^{-p} \cdot \left(64 + 40p + 20p^2 + 4p^3 + p^4/3 - p^5/15\right)\right)/p.$$

As it turns out, asymptotically, i.e., as p approaches infinity, the memory overheads of both $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` is the same as that of $\mathcal{L}$-hash, i.e., 24 (48) bytes per table entry on HotSpot32 (HotSpot64). The reason is that buckets of size $k > 6$ are not optimized in our implementation.

Nevertheless, in the typical range on HotSpot32, squashed hashing is even more memory efficient than fused hashing. $\mathcal{S}$-`HashSet` is particularly efficient, making an about two fold compaction in this range. It is also evident that both the *absolute* and *relative* savings in using squashed hashing are greater in the 64-bit memory model than in the 32-bits model.

6 Virtual Entries

Virtual entries enable read (and even write) access to the actual data (the "contained" portion in the Mitchell and Sevitsky taxonomy) in a data structure whose representation was compacted. Such access is required e.g., for in-order iteration over tree nodes, and for implementing methods in the `Map` interface (Fig. 6.1), which provide methods for the examination, and even change of ***(i)*** the set of all keys stored in the map, ***(ii)*** the multi-set of all values, and ***(iii)*** the set of all ⟨key, value⟩ pairs, nicknamed `Entry`.

The reference implementation of these methods makes use of a minimal collection data-structure containing objects that implement interface `Map.Entry`. Specifically, class `HashMap.Entry`, which defines entries in $\mathcal{L}$-`HashMap`, implements this interface. Function `entrySet()`, for example, returns an instance of `AbstractSet` wrapped around an iterator over the entire set of hash table entries.

```
public interface Map<K, V> {
  //...
  Set<K> keySet();
  Collection<V> values();
  Set<Map.Entry<K, V>> entrySet();
  //...
  interface Entry<K, V> {
    K getKey();
    V getValue()
    V setValue(V value);
    //...
  }
}
```

Fig. 6.1. Required methods for iteration over `Map` entries and interface `Map.Entry`

```
abstract class VirtualEntry<K, V>
 implements Map.Entry<K, V> {
  abstract
   VirtualEntry<K, V> next();
  protected abstract
   void setV(V v);
  @Override public final
    V setValue(V v) {
     V old = getValue();
     setV(v);
     return old;
  }
}
```

Fig. 6.2. Class `VirtualEntry`

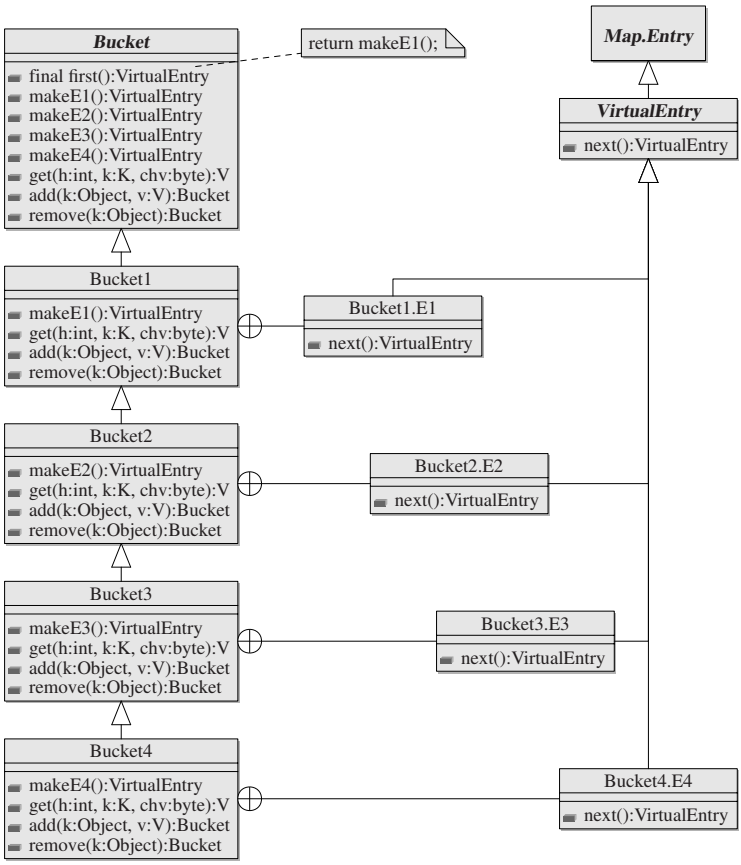

Fig. 6.3. A UML class diagram for virtual entry views on fused buckets

Class `VirtualEntry` (introduced in Fig. 6.2) presents a `Map.Entry` *view* on the fields defined e.g., in a fused bucket; where a fused bucket object typically offers a number of such views. Method `next()` in `VirtualEntry` returns the view of the next key-value pair, which is either in the same object or in a subsequent object.

The implementation of virtual entries for fused hash table includes a hierarchy of `VirtualEntry` subclasses `E1`, `E2`, `E3` and `E4`, which specialize the virtual entry concept for classes `Bucket1`, ..., `Bucket4`. Fig. 6.3 is a UML class diagram portraying the essentials.

Start with class `Bucket`; the class defines `add` and `remove` methods which are used for dynamic dispatch selection of the appropriate insertion and removal method based on the concrete bucket type. Similarly, for each of the virtual views `E1`, `E2`, `E3` and `E4`, this class defines factory methods, with default implementation returning `null`. The `final` method `first` in `Bucket` calls the first of these factory methods to return the first virtual entry stored in a fused bucket.

Then, each of `Bucket1`, ..., `Bucket4` ***(i)*** inherits the views of the class it `extends`, ***(ii)*** adds a view in its turn, and, ***(iii)*** overrides the corresponding

factory method defined in `Bucket` to return its view. For example, `Bucket2` ***(i)*** inherits the view `E1` from `Bucket1` ***(ii)*** defines the view `E2`, and, ***(iii)*** overrides the factory method `makeE2` to return an instance of this view. Class `E2`, an inner class of `Bucket2`, implements the `VirtualEntry` interface by direct access to the K_2, V_2 fields of `Bucket2`; method `next()` in `Bucket2.E2` calls the `makeE3()` factory method to generate the next view. this factory method returns `null` in this class, but overridden in `Bucket3` to return an instance of the view `Bucket3.E3`.

Function `next()` method in `Bucket4.E4` class is a bit special: if the `next` field is not `null` it returns the first view of the bucket object that follows.

The virtual entry views technique carries almost as is to squashed hashing: classes `Bucket`, `Bucket2`, ..., `Bucket6` and their inner virtual entry classes `E2`, ..., `E5` are obtained by almost mechanical application of the pattern by which each class ***(i)*** inherits the views of the class it `extends`, ***(ii)*** adds a view of its own, and, ***(iii)*** overrides the corresponding factory method defined in `Bucket`. Singleton buckets are special, since they are not represented by an object in squashed hashing. A virtual entry view of these is by class `HashMap.E1`, a non-static inner class of `HashMap`, which saves the table index passed to its constructor, as means for accessing later the singleton bucket residing at the $\texttt{i}^{th}$ position.

7 Time Performance of Fused and Squashed Hashing

The impact of fused and squashed hash table on application performance was benchmarked using three widely used standard suites: SPECjvm2008, SPECjbb2005 and DaCapo. To this end, we assembled two JRE versions; the first replacing $\mathcal{L}$-`HashMap` and $\mathcal{L}$-`HashSet` by $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`, tailored for the HotSpot32 memory model; the second JRE version was prepared using $\mathcal{F}$-`HashMap` and $\mathcal{S}$-`HashSet` tailored for the HotSpot64 memory model.

Measurements were carried out on the 32-bit and 64-bit flavors of Linux Mint 11 (Katya) OS, installed on Intel Core i3 processor running at 2.93GHz clock rate and equipped with 2GB RAM. The benchmarked code was compiled with Eclipse Helios's compiler, and linked with JRE version 1.6.0_26-b03 with the respective flavor of HotSpot Server VM 20.1-b02, mixed-mode. To ensure a clean execution environment, the benchmarked machine was placed in a single user mode, with no network connection, and using text mode rather than GUI. Further, all but one core were disabled, and clock rate on that core was set in force mode to maximal value. We also made sure by manual inspection that no background applications were running.

Since certain benchmarks gave rise to great variety in timing results (even on identical, and as "clean" as possible settings), each benchmark was executed ten times with each of the three JRE versions (baseline, fused and squashed). Student's t-test was then employed to evaluate the statistical significance of the difference in running times; our reports include both the throughput change and significance levels, i.e., the α value. However, results in which the significance levels was less than 95%, i.e., $\alpha > 5\%$, are omitted.

SPECjvm2008 benchmark setup comprised `-Xms1500m -Xmx1500m` JVM arguments (i.e., 1.5GB initial and maximal heap) and non-default application arguments `-ict -ikv --peak -bt 1`. The results are given in Table 7.1. Of the eleven benchmarks in this suite, four did not exhibit any statistically significant change to the performance; five benchmarks exhibited a statistically significant change in only one memory model; the remaining two benchmarks exhibited significant throughput change in both memory models. Although some of the values in the table are positive, many show a small performance degradation. In the 32-bit version the average throughput change was -1.22% while the average throughput change in the 64-bit version was -2.10%.

Table 7.1. Throughput change and statistical significance in SPECjvm2008

	HotSpot32		**HotSpot64**	
	throughput change	α	*throughput change*	α
compiler	-2.37%	0.00%	-3.28%	0.00%
derbi	-1.94%	0.61%		
mpegaudio	0.49%	4.74%		
scimark.large			-5.12%	0.00%
scimark.small			1.88%	0.00%
serial	-1.61%	0.08%	-3.01%	0.27%
startup			-0.99%	4.38%
avg.	-1.22%		-2.10%	

Table 7.2. Throughput change and statistical significance in SPECjbb2005 warehouses benchmarks

	HotSpot32		**HotSpot64**	
	throughput change	α	*throughput change*	α
1 warehouse	-0.34%	1.98%	-2.26%	0.22%
2 warehouses	-1.25%	0.33%	-3.14%	0.02%
3 warehouses	-1.94%	0.15%	-4.35%	0.00%
4 warehouses	-1.87%	0.11%	-4.01%	0.00%
5 warehouses	-1.41%	3.01%	1.31%	4.05%
6 warehouses	22.74%	0.00%	1.82%	1.78%
7 warehouses	1.74%	0.00%	5.82%	0.00%
8 warehouses	6.80%	0.00%	24.17%	0.00%
avg.	3.06%		2.42%	

On SPECjbb2005, the JVM arguments were `-Xms256m -Xmx256m`, while the non-default application arguments were `input.deterministic_random_seed=true`. Results are presented in Table 7.2. Evidently, all measured value were statistically significant. Negative impact on throughput typifies the smaller numbers of warehouses, but positive impact is witnessed in the larger number of warehouses. Overall, in the 32-bit memory model the *average* throughput change was 3.06%; in the 64-bit version it was 2.42%.

The JVM arguments for the DaCapo benchmark were `-Xms1500m -Xmx1500m` while the non-default application arguments were `--no-validation -C -t 1`. A few benchmarks could not be applied to our re-implementation of hash tables. The reason is that these benchmarks relied on a stored serialized version of the collection under test. The benchmarks actually used were therefore: avrora, batik, eclipse, h2, jython, luindex, lusearch, pmd, sunflow, tomcat and xalan. All statistically significant results are presented in Table 7.3. In the 32-bit version the average speedup was -0.74% while the average speedup in the 64-bit version was 4.71%.

We conclude that our proposed implementations remain within practical runtimes, imposing in some cases speedup to the JVM compared to the base implementation, while imposing significant memory overhead reduction.

Table 7.3. Throughput change and statistical significance in DaCapo

	HotSpot32		**HotSpot64**	
	speedup	α	*speedup*	α
batik	5.30%	2.18%		
h2	1.65%	3.55%		
jython			1.17%	1.26%
luindex	-17.32%	1.30%	14.55%	0.23%
pmd	8.65%	0.34%		
xalan	-1.92%	0.00%	-1.59%	0.09%
avg.	-0.74%		4.71%	

Slowdowns, when they occur, do not come as a surprise as the common usage of hash tables is as tiny (less than 16 entries) collections[6] . Naturally, our data-structures non-trivial encoding requires a more sophisticated decoding. Some operations are expected to noticeably slow down, i.e.: iterations and removals, compared to the almost trivial baseline implementation of them.

Although guided by some benchmarking, the majority of the code in our implementation was not hand optimized to achieve the ultimate time performance. It is desirable of course to make this possible.

8 Compaction of Balanced Binary Tree Nodes

This section describes the two schemes for compact representation of tree nodes of `TreeMap` (and `TreeSet`), i.e., class `TreeMap.Entry` (Fig. 1.1(a)): *fused binary tree* achieves this compaction with null-pointer and boolean eliminations; *squashed binary tree* consolidates all fields in `TreeMap.Entry`, replacing pointers by integers. The memory saving that these achieve are as reported above in Table 1.3.

Fusion. Field `color` is an obvious candidate for boolean elimination. Also, since half of the children edges in binary trees (fields `left` and `right` in `TreeMap.Entry`) are `null`, null-pointer-elimination is applicable to `left` and `right`. Fig. 8.1 shows how these techniques can be used for the compaction of leaves.

Class `Node` (implementing interface `VNode`) is the base class of all specialized tree node classes; it defines `key`, `value` and `parent` fields, just like `TreeMap.Entry`, except that the parent is necessarily an internal node (class `Internal`). Fields `left`, `right` are represented as `abstract` getter functions; null elimination of these fields is by subclass `Leaf` overriding these functions to return `null`. Field `color` is modeled as abstract getter and setter methods. The contract of the setter `color(c)` is that if `c` is different from the current node's color, it returns a new node which is identical, except for the color. The setters' implementation in class `Leaf`, creates either a `RLeaf` or `BLeaf` object as necessary.

Classes `RLeaf` and `BLeaf` have only three data fields, all of which are pointers. The object size of these classes is thus 24 bytes (with four bytes wasted on

[6] `http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-July/001969.html`

```
interface VNode<K, V>
 extends Map.Entry<K, V> {
  public boolean color();
  public VNode<K, V> parent();
  public VNode<K, V> left();
  public VNode<K, V> right();
  //...
}
abstract static class Node<K, V>
 implements VNode<K, V> {
  final static boolean // colors
   BLACK = true,
   RED = !BLACK;
  K key; V value; // contents
  Internal<K, V> parent; // topology
  VNode(K k, V v, Internal<K, V> p) {
   key = k;
   value = v;
   parent = p;
  }
  public final VNode<K, V> parent() {
   return parent;
  }
  public abstract Node<K, V>
   color(boolean c);
  //...
}
abstract static class Leaf<K, V>
 extends Node<K, V> {
  Leaf(K k, V v, Internal<K, V> p) {
   super(k,v,p);
  }
  final Node<K, V> left() {
   return null;
  }
  final Node<K, V> right() {
   return null;
  }
  final Leaf<K, V> color(boolean c) {
   return c==color() ? this : make(c);
  }
  private Leaf<K, V> make(boolean c) {
   Leaf<K, V> l = c==BLACK
      ? new BLeaf<K, V>(this)
      : new RLeaf<K, V>(this);
   //...
   return l;
  }
  //...
}
static final class RLeaf<K, V>
 extends Leaf<K, V> {
  RLeaf(Node<K, V> l) {
   super(l.key, l.value, l.parent);
  }
  @Override final public boolean color() {
   return RED;
  }
  //...
}
static final class BLeaf<K, V>
 extends Leaf<K, V> {
  BLeaf(Node<K, V> l) {
   super(l.key, l.value, l.parent);
  }
  @Override final public boolean color() {
   return BLACK;
  }
  //...
}
```

Fig. 8.1. Employing null pointer elimination and boolean elimination for the compaction of leaf nodes in red-black binary search tree

alignment) on HotSpot32 and 40 bytes (with no alignments waste) on HotSpot64. The size of the `TreeSet` version of these classes is 16 bytes on HotSpot32, and 32 bytes on HotSpot64.

We empirically found that $\approx 42.8\%$ of nodes in a red-black tree are leaves, and that this ratio is independent of tree size, nor of tree creation order.[7] Employing classes `RLeaf` and `BLeaf`, in an implementation of `TreeMap` reduces overhead from 24 to 20.6 bytes on HotSpot32 (respectively 48 to 37.7 on HotSpot64) which amounts to 14% (21%) savings. With `TreeSet` the respective reductions are 28 to 21.1 and 56 to 42.3 (both 24% saving).

Since each of the tree's nodes is "owned" by its parent, it makes sense to apply fusion in tree nodes, just as we did for lists. The difficulty is in dealing with the very many cases that could occur: Depth-ℓ fusion may entail a specialized class for the $O(2^\ell)$ trees of this depth. Attentions is therefore restricted to fusion of nodes with their leaves (i.e., $\ell = 2$), distinguishing between three different cases: *(i)* internal nodes which are parents to two leaves, *(ii)* internal nodes which are

[7] This high value fraction is not accidental; similar fractions occur in e.g., AVL trees. We can in fact analytically prove that about one quarter of the nodes are leaves in a *random unbalanced* binary tree; balancing leads to increasing the number of leaves.

parents to a single leaf, the other child being `null`, and *(iii)* nodes in which one of the children is a leaf and the other is a non-leaf non-`null` node, which we will ignore.

The characteristics of a red-black tree limit the variety of colors in cases (i) and (ii). In (i), if the parent is RED, then children are both BLACK (if the parent is BLACK then colors of children may be of any color). In (ii), if the parent is BLACK then its leaf child must be RED. Five different concrete types of nodes are thus defined: `ParentLeftLeaf`, `ParentRightLeaf`, `ParentLeavesRBB`, `ParentLeavesBBB`, and, `ParentLeavesBRR`.

As shown in Fig. 8.2, the first two classes have five pointer fields, Fields `parent`, `key` and `value` are inherited from the superclass `Node`, while two additional pointers, `keyChild` and `valueChild` are defined in `ParentLeaf` which is the `abstract` superclass of both `ParentLeftLeaf` and `ParentRightLeaf`.

```
abstract static class
 ParentLeaf<K, V>
  extends Node<K, V> {
   protected K keyChild;
   protected V valueChild;
   //methods and inner classes...
}
static final class
 ParentLeftLeaf<K, V>
  extends ParentLeaf<K, V> {
   //methods and inner classes...
}
static final class
 ParentRightLeaf<K, V>
  extends ParentLeaf<K, V> {
  //methods and inner classes...
}
abstract static class
 ParentLeaves<K, V>
  extends ParentLeaf<K, V> {
    protected final K keyRightLeaf;
    protected V valueRightLeaf;
   //methods and inner classes...
}
static final class
 ParentLeavesRBB<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
static final class
 ParentLeavesBBB<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
static final class
 ParentLeavesBRR<K, V>
  extends ParentLeaves<K, V> {
  //methods and inner classes...
}
```

Fig. 8.2. Classes for fusion of internal tree nodes with their leaf children

The footprint of `ParentLeftLeaf` and `ParentRightLeaf` classes for `TreeMap` is $8 + 5 \cdot 4 = 28$ bytes, which are up-aligned to 32 bytes (on HotSpot32). Class `ParentLeaves` adds two more pointers, making a 40 bytes footprint for each of its three subclasses on HotSpot32.

Empirically we found that 14% of the nodes in a red-black tree have a single leaf child, while 9% of the nodes have two leaves as their children. Then, $2 \cdot 14\% = 28\%$ of the nodes consume $32/2 = 16$ bytes each, while $3 \cdot 9\% = 27\%$ of the nodes consume $40/3 = 13.3$ bytes each. Assuming that no compression is done for the other nodes, we obtain 14.5 bytes of overhead per node, achieving 40% savings in overhead per node, just by using leaf level fusion. If the remaining leaves are represented as in Fig. 8.1, then saving increases to 43% for $\mathcal{F}$-`TreeMap` on HotSpot32.

Our implementation of fused binary trees goes further, employing boolean elimination to class `TreeMap.Entry` (Fig. 1.1(a)) which is used for all other

```
abstract static class Internal<K, V>
 extends Node<K, V> {
  Node<K, V> left, right;
  //...
  @Override final public
   VNode<K, V> left() {
    return left;
  }
  @Override final public
    VNode<K, V> right() {
      return right;
  }
  @Override final Internal<K, V>
    color(boolean c) {
     return c==color()
           ? this
           : make(c);
  }
  private
   Internal<K, V> make(boolean c) {
    Internal<K, V> n = c==BLACK
          ? new BInternal<K, V>(this)
          : new RInternal<K, V>(this);
     //...
     return n;
   }
  //...
}
final static class RInternal<K, V>
  extends Internal<K, V> {
   //...
   @Override final public boolean color() {
    return RED;
   }
}
final static class BInternal<K, V>
 extends Internal<K, V> {
  //...
   @Override final public boolean color() {
    return BLACK;
   }
}
```

Fig. 8.3. Abstract class `Internal` and boolean elimination in specialized internal nodes

kinds of tree nodes (internal nodes). (Described in Fig. 8.3.) Such elimination capitalizes the memory reduction for the other $\mathcal{F}$-tree types besides $\mathcal{F}$-`TreeMap` on HotSpot32, which due to alignment waste does not benefit from it. Combined with the above techniques we achieve the following overhead savings: *(i)* 59% for `TreeSet` on HotSpot32, *(ii)* 55% for `TreeMap` on HotSpot64, and *(iii)* 61% for `TreeSet` on HotSpot64.

Not surprisingly, with nine different concrete classes for tree nodes, coding was not easy. Difficulties include the fact that tree updates may turn internal nodes into leaves and vice versa, and that rotations may change the tree topology. Removals were particularly challenging as they may initiate any number of tree rotations. As with hashing, dynamic dispatch was employed for abstracting over the variety of node types, and virtual entries were used for iterating over the tree nodes. Initial benchmarking results indicate that this abstraction layer lead to 30-50% slowdown.

Full Field Consolidation. Squashing encodes the entire `TreeMap<K,V>` data structure without using *any* small objects. Instead, six tree-global arrays, `K[] key`, `V value[]`, `boolean[] color`, `int[] left`, `int[] right`, and `int[] parent` consolidate the fields of all tree nodes; node i is then the $\mathtt{i}^{th}$ location in all of these arrays, while pointers are replaced by array indices. This consolidation eliminates both headers of all small objects, and alignment waste incurred to the byte sized `color` field. When the arrays are fully occupied, a node overhead is reduced from 32 to 21 bytes on HotSpot32, and from 64 bytes to 20 bytes on HotSpot64. On both memory models overhead is 13 bytes, and we achieve the following overhead savings: *(i)* 46% for `TreeMap` on HotSpot32, *(ii)* 54% for `TreeSet` on HotSpot32, *(iii)* 73% for `TreeMap` on HotSpot64, and *(iv)* 77% for `TreeSet` on HotSpot64. Implementation, as expected, was almost mechanical; initial benchmarking results indicate that the performance of the squashed tree implementation is comparable to the baseline, and sometimes even slightly faster.

Note that squashed trees come at the cost of shifting memory management duties from the JVM back to the programmer. To avoid costly reallocations, array should include sufficient slack. Still, a generous 50% slack places field consolidation behind fusion.

It is feasible to implement a version of squashed trees, in which the arrays' type changes by the current size of the collection: `byte[]` arrays, then `short[]` arrays and finally `int[]` arrays. The expected saving increases for small collections as summarized in Table 8.1. Table ignores *instance-memory* waste (See 3) which may be slightly more significant for smaller objects. Accounting for this waste, the expected addition is of two bytes per key for a collection of size 50, and decreases rapidly as size increases.

Table 8.1. Computed saving in memory overhead per tree entry due to the use of full consolidation using different types of indices in the implementation of `TreeMap` and `TreeSet` for HotSpot32 and HotSpot64

	HotSpot32			**HotSpot64**		
	int	*short*	*byte*	*int*	*short*	*byte*
`TreeMap`	46%	71%	83%	73%	85%	92%
`TreeSet`	54%	75%	86%	77%	88%	93%

9 Further Research

This research raises a number of interesting questions. First, it is important and interesting to understand better the domain of tiny collections, of say up to say 16 entries, as their relative overhead is more significant. Reducing the overhead of these seems more challenging, especially in adhering to the very general `Map` interface. It would be useful to make estimates on abundance of tiny collections in large programs and the manner in which they are used, with the conjecture that a frugal yet less general implementation of these would be worthwhile.

Second, as we have seen in this work the variety of the user-level compaction algorithms are not always easy to employ. A software framework or better yet, automatic tools that abstract over encoding issues would make our findings more accessible. It is crucial for such a framework to be able to produce code for both (say) `TreeMap` and `TreeSet` without code duplication. The virtual entries technique presented in Sect. 6 may serve as a starting point, but other directions may include the use of aspects or more sophisticated generics.

Third, we are intrigued by the fact that despite fewer dereferencing operations, $\mathcal{F}$-hash and $\mathcal{S}$-hash were not significantly faster than $\mathcal{L}$-hash. Micro-benchmark of individual operations should not only clarify this point, but also make room for systematic hand- and later automatic- optimization of these.

Finally, we draw attention to the problems of memory profiling, a meaningful and precise definition of the notion of "footprint" of an application, and its impact on time. These issues are illusive since the "footprint" changes in the course of computation, and the memory consumption curve may depend on garbage collection cycles, which in general are not deterministic, yet may depend on the allocation of physical memory and other factors.

References

1. Adl-Tabatabai, A.-R., Cierniak, M., Lueh, G.-Y., Parikh, V.M., Stichnoth, J.M.: Fast, effective code generation in a just-in-time Java compiler. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2008), Tucson, Arizona, June 7-13. ACM Press (2008)
2. Alon, N., Dietzfelbinger, M., Miltersen, P.B., Petrank, E., Tardos, G.: Linear hash functions. J. ACM 46 (September 1999)
3. Arbitman, Y., Naor, M., Segev, G.: Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. In: Proc. of the 51st IEEE Annual Symp. on Foundation of Comp. Sci. (FOCS 2010), Las Vegas, Navada, October 23-26. IEEE Computer Society Press (2010)
4. Bacon, D.F., Fink, S.J., Grove, D.: Space and time efficient implementation of the Java object model. In: Proc. of the 17th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2002), Seattle, Washington, November 4-8 (2002); ACM SIGPLAN Notices 37(11)
5. Caromel, D., Reynders, J., Philippsen, M.: Benchmarking Java against C and Fortran for scientific applications. In: Thomas, D. (ed.) Proc. of the 20th Euro. Conf. on OO Prog. (ECOOP 2006), Nantes, France, July 3-7. LNCS, vol. 4067. Springer (2006)
6. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. In: Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 1998), Vancouver, British Columbia, Canada, October 18-22 (1998); ACM SIGPLAN Notices 33(10)
7. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling Java just in time. IEEE Micro 17(3) (May/June 1998)
8. Cranor, L.F., Wright, R.N.: Influencing software usage. In: Proc. of the 10th Conference on Computers, Freedom and Privacy (CFP 2000). ACM (2000)
9. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In: Proc. of the 16th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (FSE 2008), Atlanta, Georgia, November 9-14. ACM Press (2008)
10. Eckel, N., Gil, J.: Empirical Study of Object-Layout Strategies and Optimization Techniques. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 394–421. Springer, Heidelberg (2000)
11. Feller, W.: An Introduction to Probability Theory and Its Applications, vol. I. Wiley (1968)
12. Gil, Y., Lenz, K., Shimron, Y.: A microbenchmark case study and lessons learned. In: Proc. of the 5th International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (VMIL 2011). ACM (2011)
13. Hsieh, C.H.A., Conte, M.T., Johnson, T.L., Gyllenhaal, J.C., Hwu, W.M.W.: Compilers for improved Java performance. Computer 30(6) (June 1997)
14. Kawachiya, K., Ogata, K., Onodera, T.: Analysis and reduction of memory inefficiencies in Java strings. In: Harris, G.E. (ed.) Proc. of the 23rd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2008), Nashville, Tennessee, October 19-23. ACM (2008)
15. Kotzmann, T., Wimmer, C., Mossenbock, H., Rodriguez, T., Russell, K., Cox, D.: Design of the Java HotSpot client compiler for Java 6. ACM Trans. Prog. Lang. Syst. 5(1) (May 2008)

16. Maxwell, E.K., Back, G., Ramakrishnan, N.: Diagnosing memory leaks using graph mining on heap dumps. In: Proc. of the 16th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2010), Washington, DC, July 25-28. ACM Press (2010)
17. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to Java runtime bloat. IEEE Software 27(1) (2010)
18. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Gabriel, R.P., Bacon, D. (eds.) Proc. of the 22nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2007), Montreal, Quebec, Canada, October 21-25. ACM Press (2007)
19. Moreira, J.E., Midkiff, S.P., Gupta, M.: A comparison of Java, C/C++, and FORTRAN for numerical computing. IEEE Antennas and Propagation Magazine 40(5), 102–105 (1998)
20. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2009)
21. Ogata, K., Mikurube, D., Kawachiya, K., Onodera, T.: A study of Java's non-Java memory. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proc. of the 25th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA 2010), Reno/Tahoe, Nevada, USA, October 17-21. ACM (2010)
22. Paleczny, M., Vick, C., Click, C.: The Java Hotspot server compiler. In: Proc. of the Java Virtual Machine Research and Technology Symposium (JVM 2001), Monterey, California, April 23-24. USENIX C++ Technical Conf. Proc. (2001)
23. Reiss, S.P.: Visualizing the Java heap. In: Proc. of the 32nd Int. Conf. on Soft. Eng. (ICSE 2010), Cape Town, South Africa, May 2-8. ACM (2010)
24. Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive selection of collections. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2009), Dublin, Ireland, June 15-20. ACM Press (2009)
25. Venstermans, K., Eeckhout, L., Bosschere, K.D.: Java object header elimination for reduced memory consumption in 64-bit virtual machines. TACO 4(3) (2007)
26. Wimmer, C.: Automatic Object Inlining in a Java Virtual Machine. PhD thesis, Institute for System Software, Johannes Kepler University Linz (2008)
27. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: Profiling copies to find runtime bloat. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI (2009)
28. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: Proc. of the 18th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (FSE 2010), Santa Fe, New Mexico, November 7-11. ACM Press (2010)
29. Xu, G., Rountev, A.: Precise memory leak detection for Java software using container profiling. In: Proc. of the 30th Int. Conf. on Soft. Eng. (ICSE 2008), Leipzig, Germany, May 10-18. ACM (2008)
30. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: Proc. of the Conference on Programming Language Design and Implementation (PLDI 2010), Toronto, Canada, June 5-10. ACM Press (2010)