Smaller Footprint for Java Collections

Yuval Shimron

Technion - Computer Science Department - M.Sc. Thesis MSC-2012-05 - 2012

Smaller Footprint for Java Collections

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Yuval Shimron

Submitted to the Senate of
the Technion — Israel Institute of Technology
Cheshvan, 5772 Haifa November, 2011

The research thesis was done under the supervision of Prof. Joseph (Yossi) Gil in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged.

Contents

List of Figures

List of Tables

Abstract

In dealing with the container bloat problem, we identify five memory compaction techniques, which can be used to reduce the footprint of the 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.

First of this is our *fused hashing* encoding method, which 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. Second, our 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.

Timing results indicate that no significant improvement or degregation in runtime is noticable for several common JVM benckmarks: 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.

We note that with the use of our compaction techniques, there is merit in an implementation of `HashSet` which is distinct from that of `HashMap`.

For `TreeMap` we show two encodings which reduces the overhead of tree nodes by 43% , 46% on a 32-bit environment and 55% , 73% on a 64-bit environment. These compactions give also a reason to separating the implementation of `TreeSet` from that of `TreeMap`. A separete implementation

is expected to increase the footprint reduction to 59% , 54% on a 32-bit environment and 61% , 77% on a 64-bit environment.

Abbreviations and Notations

n	—	Collection size (num. of keys)
m	—	Array length
κ	—	Retrieved/stored key
$p = n/m$	—	Table density
$\alpha(k)$	—	Expected fraction of buckets of size k
$\beta(k)$	—	Expected fraction of keys that fall into buckets of size k
μ_k	—	Total number of bytes required for representing a buckets of size k
CHV	—	Cahced hash value of a key in a hash map or set
K	—	`key` field of some `Bucket` or `Entry` object
K_j	—	j^{th} `key` field of some `Bucket` or `Entry` object
V	—	`value` field of some `Bucket` or `Entry` object
V_j	—	j^{th} `value` field of some `Bucket` or `Entry` object
h	—	CHV field of some `Bucket` or `Entry` object
h_j	—	j^{th} `chv` field of some `Bucket` or `Entry` object
$\mathcal{L}$-hash	—	Legacy JRE version of hash map implementation (class `HashMap` or class `HashSet`)
$\mathcal{L}$-`HashMap`	—	Legacy JRE version of class `HashMap`
$\mathcal{L}$-`HashSet`	—	Legacy JRE version of class `HashSet`
$\mathcal{F}$-`HashMap`	—	"Fused" version of class `HashMap`
$\mathcal{F}$-`HashSet`	—	"Fused" version of class `HashSet`
$\mathcal{S}$-`HashMap`	—	"Squashed" version of class `HashMap`
$\mathcal{S}$-`HashSet`	—	"Squashed" version of class `HashSet`

Chapter 1

Introduction

1.1 Background

JAVA [4] and the underlying virtual machine provide software engineers with a programming environment which abstracts over many hardware specific technicalities. The *runtime* cost of this abstraction is offset by modern compiler technologies, including *just in time* compilations [1, 7, 9, 17, 19, 23, 26]. Indeed, we see more and more indications [7, 17, 23] that JAVA's time performance is getting closer and closer to that of languages such as C++ [29] and FORTRAN that execute directly on the hardware and do not suffer from the performance penalties of memory management.

In contrast, the JAVA memory cost is not an easy target for automatic optimization. With the ever increasing use of JAVA for implementing servers and other large scale applications, indications are increasing that the open-handed manner of using memory, which is so easy to resort in JAVA, leads to memory bloat, with negative impacts on time performance, scalability and usability [21].

The research community responses include methods for diagnosing the overall "memory health" of a program and acting accordingly [22], leak detection algorithms [24], and methods for analyzing [20] profiling [31, 33] and visualizing [27] the heap. (See also a recent survey on the issue of both time and space bloat [32].)

This work is concerned primarily with what might be called "container bloat" (to be distinguished e.g., from temporaries bloat [11]), which is be-

lieved to be one of the primary contributors to memory bloat.[1] Previous work on the container bloat problem included methods for detecting suboptimal use of containers [34] or recommendations on better choice of containers based on dynamic profiling [28]. Our line of work is different in that we propose more compact containers. Much in the line of work of Kawachiya, Kazunori and Onodera [18] 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.

We emphasize that 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 on this has its place, but our focus here is on the question of whether given data and data structures can be *encoded* more efficiently. After all, whatever method a data structure is organized in, there could be room for encoding it more efficiently, just as subjecting a super fast algorithm to automatic optimization could improve it further.

To this end, we insist on maximal 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 believe that the techniques we identify and the detailed case studies in employing these for current Java collections could pave the way to a future in which automatic tools would aid and even take control of exercising these.

1.2 Memory Overhead per Entry

To appreciate the scale of container overhead, note first that a simple information theoretical consideration sets a minimum of n references for *any*

[1] Consider, e.g., Mitchell & Sevitsky's example of a large online store application consuming $2.87 \cdot 10^9$ bytes, 42% of which are dedicated to class `HashMap` (OOPSLA'07 presentation).

representation of a set of n keys, and $2n$ references for *any* representation of a n pairs of key and values. Table 1.2.1 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.

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

Table 1.2.1: Minimal memory (in bytes) per entry in a set and a map data structures.

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 as a compact sorted array, in which search is logarithmic, while updates are 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, one is willing to tolerate, for practical purposes some memory overhead. Still, 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 implementation of a sorted map. Memory overheads incurred by using this data structure can be appreciated by examining fields defined for each tree node, realized by the internal class `TreeMap.Entry`.

Listing 1.2.1 Fields defined in `TreeMap.Entry`.

```
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;
    //...
}
```

Listing 1.2.1 shows that on top of the object header, each tree node stores 5 pointers and a **`boolean`** whose minimal footprint is only 1 bit, but

typically requires at least a full byte. 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. For various reasons this class is realized 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 a parameter which ranges typically between 3/8 and 3/4. Secondly, an object of the internal class `HashMap.Entry` (depicted in Listing 1.2.2) is associated with each such entry.

Listing 1.2.2 Fields defined in class `HashMap.Entry`.

```
public class HashMap<K, V> {
  // ...
  static class Entry<K, V> implements Map.Entry<K, V> {
    final K key; // The key stored in this entry
    V value; // The value associated with this key
    Entry<K, V> next; // The next HashMap.Entry object in the bucket
    final int hash; // A cached hash value.
    // ...
  }
  // ...
}
```

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

Table 1.2.2 summarizes the overhead, in bytes, for representing the four fundamental JRE collections we discussed.

	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 1.2.2: Memory overhead per entry of common data structure in central JRE collections.

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.

1.3 This Work

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: null pointer elimination, boolean elimination, object fusion, field pull-up and field consolidation. The techniques can be applied independently, but they become more effective if used wisely together, often 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. The description includes a software method for treating the encoded information transparently.

We present a mathematical model in which the footprint of these implementations can be analyzed. In this model, we deduct that $\mathcal{F}$-hash reduces memory overhead by 20%–40% on a 32-bit environment and 45%–65% on a 64-bit environment. These figures constitute a lower bound, i.e., the encoding guarantees at least these savings regardless of the distribution of keys in the buckets. The more opportunistic and more aggressive $\mathcal{S}$-hash achieves expected savings of 30%–50% on a 32-bit environment and 60%–70% on a 64-bit environments, but these savings can degrade and are not guaranteed against bad (and unlikely) distribution of keys to buckets.

Timing results indicate that no significant improvement or degregation in runtime is noticable for several common JVM benckmarks: 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.

Further, we note that with the use of the compaction tool chest, memory savings which were not effective for `HashSet` become feasible. These additional savings give merit to an implementation of `HashSet` which is distinct from that of `HashMap`.

For `TreeMap` we show encodings which reduce the overhead of tree nodes by 40% and more. These compactions give similar reason to a separate implementation of `TreeSet` which is more memory efficient.

Outline The remainder of this article is organized as follows. The five memory compaction techniques we identified are described in Chapter 2. Chapter 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 chapters 4 and 5, while their space and time performance are the subject of Chapter 6. Chapter 7 discusses the compaction of `TreeMap` and `TreeSet`. Chapter 8 concludes with problems for further research.

Chapter 2

Compaction Techniques

The space compaction techniques that we identify include:

- *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 [8] 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 which we use, 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 [5], including an implementation of the JVM in 64-bit environment in which there is no header at all [30]).

The mandatory object header makes fusion very effective. In C++, small objects would have no header (`vptr` in the C++ lingo [12]), 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, 8 for `object` header and 4 for the array `length` field. 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` could have halved its footprint.

[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`.

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:

1. *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 pre-defined 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.)

2. *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.

Chapter 3

Hash Tables of the JRE

This section reviews the present implementation of hash tables in the JRE due to J. Bloch and D. Lea (Section 3.1), computes the memory requirements of the implementation (Section 3.2), reminds the reader of the Poisson distribution and its application for hashing (Section 3.3), and discusses memory inefficiency in the JRE implementation (Section 3.4).

3.1 Implementation

Other than the implementations designed for concurrent access, we find three principal classes: First is class `IdentityHashMap` which 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.

Secondly, class `HashMap` (nicknamed $\mathcal{L}$-`HashMap`) uses *chained hashing* method, by which the i^{th} table position contains a *bucket* of all keys κ for which

$$H(\kappa) \bmod m = i. \tag{3.1.1}$$

This bucket is modeled as a singly-linked list of nodes of type `HashMap.Entry` (depicted in Listing 1.2.2). The hash table itself is then simply an array `table` of type `HashMap.Entry[]`.

Finally, class `HashSet` is nothing but a `HashMap` in disguise, delegating all set operations to `map`, an internal `private` field of type `HashMap` which maps all keys in the set to some fixed dummy object.

It is estimated[1] that class `IdentityHashMap` is between 15% and 60% faster than `HashMap`, and occupies around 40% smaller footprint. Yet class `IdentityHashMap` is rarely used [2] 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.

The hash function H mentioned in (3.1.1) above is defined in the implementation of `HashMap` as `hash(key.hashCode())` where function `hash` is as in Listing 3.1.1.

Listing 3.1.1 Bit spreading function implementation from `HashMap` class.

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

The purpose of function `hash` 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 $H(\kappa) \bmod m$ 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 this 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`.

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

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

Listing 3.1.2 JAVA code for searching in $\mathcal{L}$-`HashMap`.

```
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;
}
```

The code of function `get` (Listing 3.1.2) 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 introduced the abbreviated notation K, V, and h for fields `key`, `value` and `hash`, a notation which we will use 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),

$$\texttt{loadFactor}/2 < p \leq \texttt{loadFactor}. \tag{3.1.2}$$

The default value of `loadFactor` is `0.75`, and it is safe to believe [10] that users rarely change this value, in which case,

$$3/8 < p \leq 3/4. \tag{3.1.3}$$

We call (3.1.3), the *typical range of* p, or just the "typical range". The center of the typical range,

$$p = (3/8 + 3/4)/2 = 9/16 \tag{3.1.4}$$

is often used in benchmarking as a point characterizing the entire range. [3]

[3] If p is distributed evenly in the range (3.1.3), and $M = M(p)$ is some cost measure, then the "average" value of M can be approximated by $M(9/16)$, due to the approxima-

3.2 Footprint Analysis

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 [25], 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 carry on with the analysis, computing the memory use per entry, i.e., total memory divided by the number of entries in the table.

On HotSpot32 array `table` consumes $4m$ bytes for the array content along with 16 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 Listing 1.2.2, 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. Number of bytes per table entry is therefore

$$24 + 4/p. \tag{3.2.1}$$

tion $\frac{1}{b-a}\int_a^b M(p)dp \approx M(\frac{a+b}{2})$; a number of issues could be raised against this approximation, but we will stick to it nevertheless.

For HosSpot64, 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

$$48 + 8/p \tag{3.2.2}$$

bytes per object. (Comparing (3.2.1) and (3.2.2) with Table 1.2.1 gave the memory overheads of $\mathcal{L}$-`HashMap` and $\mathcal{L}$-`HashSet`, as tabulated in Table 1.2.2.)

Observe that alignment issues are probably among the reasons behind the decision to implement `HashSet` as `HashMap`: eliminating the `value` field will still give the same number of bytes per `Entry` object (at least on HotSpot32).

3.3 Distribution of Buckets' Size

Hashing can be modeled by the famous urn statistical model [13]: In hashing n keys into m buckets, the value of $m(k)$, the number of buckets with precisely k keys, $k \geq 0$, is governed by the Poisson distribution

$$m(k) = m\frac{p^k e^{-p}}{k!}. \tag{3.3.1}$$

Let

$$\alpha_k(p) = \frac{m(k)}{m} = \frac{p^k}{k!}e^{-p}, \tag{3.3.2}$$

denote the expected fraction of buckets of size k.

Figure 3.3.1 plots $\alpha_k(p)$ vs. p for $k = 0, \ldots, 4$.

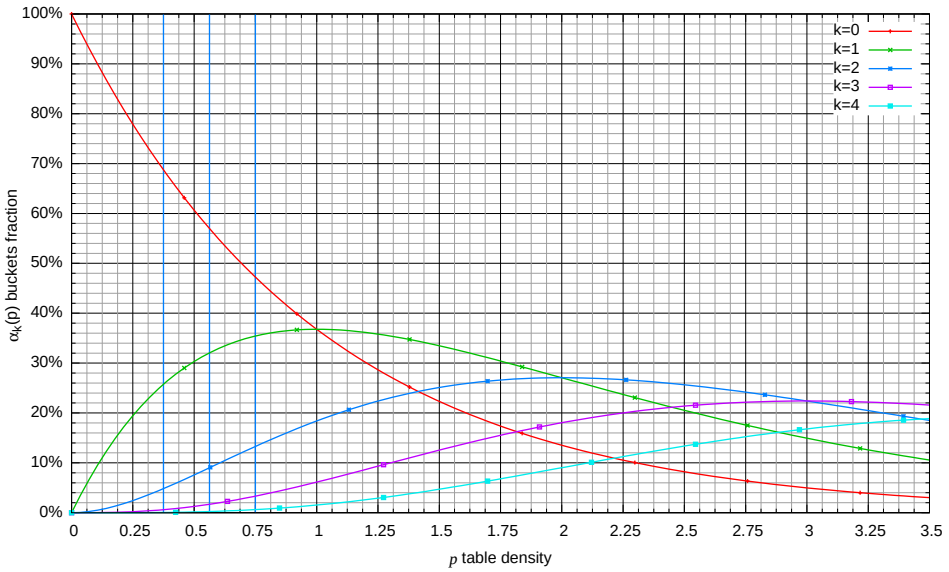

Figure 3.3.1: Expected fraction of buckets of size k, $k = 0, \ldots, 4$ vs. table density.

The figure portrays an exponential decay of $\alpha_0(p)$ as p grows ($\alpha_0(p) = e^{-p}$). But, for values of p typical in implementation of hash data structures, a significant portion of the buckets are empty. Even when $p = 1$, 37% of the buckets are empty.

The endpoints of the typical range (3.1.3) as well as its center (3.1.4) are shown as vertical blue lines in the figure. We see that the fraction of empty buckets ranges between $\alpha_0(3/8) \approx 69\%$ (maximal value), and $\alpha_0(3/4) \approx 47\%$ (minimal value) in the typical range.

For $k > 0$, and for a given density, let $\beta_k(p)$ denote the expected fraction of keys which fall into buckets of size k. We have that for all $k > 0$,

$$\beta_k(p) = \frac{km(k)}{n} = \frac{k}{p}\frac{p^k}{k!}e^{-p} = \frac{p^{k-1}}{(k-1)!}e^{-p} = \alpha_{k-1}(p), \tag{3.3.3}$$

while

$$\beta_0(p) = 0. \tag{3.3.4}$$

We can therefore read the fraction of keys falling into buckets of size k

by inspecting the $(k-1)^{th}$ curve in Figure 3.3.1. 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.

3.4 Discussion

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 [22]. 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.

Having considered the header, and the first three fields of `HashMap.Entry` (Listing 1.2.2) it remains to consider the CHV, i.e., field `hash`. This field should be classified as *primitive-overhead* in the GM taxonomy [22, 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 bits that are meaningful in the comparison of for 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

`(byte) hash1(key.hashCode())`.

where `hash1` is the first stage in computing `hash` (see Listing 3.4.1).

Listing 3.4.1 Two steps hash code modification function implementation.

```
static int hash1(int h) {
    return h ^ h >>> 20;
}

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

Chapter 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. As it turns out, the naïve way of doing so is not very effective. The reason is that in singleton buckets (which form the majority of buckets in typical densities), the specialized entry should include two references (to the key and value) as well as the CHV. It follows that in the common 8-byte alignment memory models, 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 `chv` of length m, which parallels the main `table` array. (As explained above, it is sufficient to let array `chv` to be of type `byte[]` rather than a lavish `int[]`.)

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, which 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`: first we have class `Bucket1`, which represents singleton buckets, `extends` class `Bucket`; then, class `Bucket2` which `extends` class `Bucket1` is used for the representation of buckets of size 2; next comes class `Bucket3` extending

class `Bucket2`, dedicated to 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 $k = 1, 2, 3$. Moreover, every four consecutive entries are packed into a single object: buckets of size $k > 4$ are represented by 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 $k' = k \bmod 4$.

4.1 Fused Objects Layout

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 4.1.1 elaborates the contents and size in the HotSpot32 memory model of each of the five bucket classes, both for the $\mathcal{F}$-`HashMap` and for $\mathcal{F}$-`HashSet` implementations.

	$\mathcal{F}$-`HashMap`		$\mathcal{F}$-`HashSet`	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8
`Bucket1`	K_1, V_1	16	K_1, $\uparrow h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	16
`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
`Bucket3`	$V_3, \uparrow K_4$	40		24
`Bucket4`	V_4, `next`	48	K_4, `next`	32

Table 4.1.1: Content and size of classes `Bucket`, `Bucket1`, ..., `Bucket4` in $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` (HotSpot32).

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.

We have, for example, 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.)

Table 4.1.1 also describes the layout of buckets in $\mathcal{F}$-`HashSet`. This layout is similar, except that the absence of value fields increases field pull-up opportunities, leading to greater memory savings.

Examining 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, as opposed to the 24 bytes footprint in the $\mathcal{L}$-hash implementation. A more careful analysis which uses our statistical model of distributions of bucket sizes, to estimate the exact savings is offered below in Chapter 6.

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. An entry of a bucket of size 1,2,3,4 occupies 16, 16, 13.33, 12 bytes (respectively) in `FHashMap` and 16, 12, 8, 8 (respectively) in `FHashSet`. It is easy to check that this property is preserved even in longer buckets.

Table 4.1.2 repeats the contents of Table 4.1.1 but for the HotSpot64 memory model.

	$\mathcal{F}$-HashMap		$\mathcal{F}$-HashSet	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	16	object header	16
`Bucket1`	K_1, V_1	32	K_1,	24
`Bucket2`	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`	K_3, V_3	72	K_3	48
`Bucket4`	K_4, V_4, `next`	96	K_4, `next`	64

Table 4.1.2: Content and size of classes `Bucket`, `Bucket1`, ..., `Bucket4` in $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` (HotSpot64).

Naturally, objects consume more memory in moving from a 32-bits memory model to a 64-bits model. But, the 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%).

Observe that the non-decreasing compression property holds in Table 4.1.2 as well: In the HotSpot64 model, an entry of a bucket of size 1,2,3,4 occupies 32, 28, 24, 24 bytes (respectively) in `FHashMap` and 24, 20, 16 ,16 bytes (respectively) in `FHashSet`.

4.2 Searching in a Fused Bucket

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. However, we can only access K_i if we know that the `Bucket` object is of type `Bucket`$_i$ or a subtype thereof.

It is possible to implement the search by sending a `get` message to the `Bucket` object, and have different implementations of `get` in each of the `Bucket` classes. We found however that, in this trivial inheritance structure, even efficient implementation of dynamic dispatch [15] is slightly inferior to the direct application of JAVA's **`instanceof`** operator to determine the bucket's dynamic type.

Listing 4.2.1 shows the details of the implementation of method `get` in class $\mathcal{F}$-`HashMap`.

Listing 4.2.1 JAVA code for searching a given key in a fused bucket (buckets with 0, 1 or 2 keys).

```
public V get(Object κ) {
  if (κ == null) κ = nullKey;
  int h = hash1(κ.hashCode());
  final int i = hash2(h) & table.length - 1;
  final 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;
  final Bucket2<K, V> b2 = (Bucket2)b1;
  if ((b2.h2 == h && ((k = b2.K2) == κ || κ.equals(k)))
    return b2.V2;
  if (!(b2 instanceof Bucket3)) return null;
  return get3((Bucket3)b2, h, κ);
}
```

This implementation of `get` essentially follows that of $\mathcal{L}$-hash in Listing 3.1.2. [1] We see that the sequential linear 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 this bucket is of type `Bucket2`, in which case, b_1 is down casted into type `Bucket2`, saving the result in b_2 and we proceed to examining field K_2.

As in the baseline implementation (Listing 3.1.2), the CHV fields (h_1, h_2 in Listing 4.2.1) are used to accelerate the search, and as in the baseline implementation, an identity comparison precedes the call to the potentially slower `equals` method.

The code in Listing 4.2.1 deals with buckets of up to two entries. If the bucket has three entries or more, the code chains forward to method `get3` depicted in Listing 4.2.2.

Method `get3` is similar in structure to `get`: after a failure in comparison of the searched key to field K_3, operator **`instanceof`** is employed to check whether the current bucket is a `Bucket4` object. If the searched key is not equal to field K_4 and the bucket has more than four keys, i.e., if the bucket

[1] The handling of the case that κ is **`null`** in the first line of the implementation is slightly different, but it is safe to disregard the variation for the purposes of this paper.

Listing 4.2.2 JAVA code for searching a given key in a fused bucket (buckets with 3 keys or more).

```
private V get3(Bucket3<K, V> b3, int h, Object κ) {
  Object k;
  if (b3.h3 == h && ((k = b3.K3) == κ || κ.equals(k)))
    return b3.V3;
  if (!(b3 instanceof Bucket4)) return null;
  final Bucket4<K, V> b4 = (Bucket4) b3;
  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);
}
```

is of type `Bucket4` and its `next` field is not **`null`**, then the search proceeds by chaining forward using dynamic dispatch to a `get` method specialized for each bucket size.

4.3 Hash Table Updates

It is generally believed that search operations, and in particular, successful search, are the most frequent operations on collections. [2] For this reason we invested some effort in hand optimizing the `get` function, including partitioning the code to three distinct cases: buckets of size 0, 1 or 2, buckets of size 3 or 4, and larger buckets.

A similar motivation stood behind our decision to use **`instanceof`** operator instead of dynamic binding for method `containsKey`—which is arguably the most frequent operation in using a hash data structure for set representation.

The remaining operations in the `Map` interface can be similarly optimized, but we choose a more straightforward implementation for these. Method `put`, in charge of insertion of keys into the table, is implemented by a `add` method in class `Bucket`, which has a particular specialization in classes `Bucket1` through `Bucket4`. Method `add` inserts a key into the bucket, returning a freshly created `Bucket` object in which the inserted key is added. (With the exception that the same `Bucket` object is returned in case the key is found in the table or it was added to a referenced `Bucket`)

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

A similar approach is used for removing table entries: a dynamic dispatch based on the `Bucket` type removes the entry from the fused bucket and creates a new fused bucket.[3]

4.4 Collection Perspectives

The `Map` interface includes three methods that make it possible to examine, and even change, ***(i)*** the set of all keys stored in the map, ***(ii)*** the multi-set of all values, and ***(iii)*** the set of all $\langle \text{key}, \text{value} \rangle$ pairs, nicknamed `Entry`. The pertinent portions of the JAVA definition are depicted in Listing 4.4.1.

Listing 4.4.1 Required methods for iteration over `Map` entries and interface `Map.Entry`.

```
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);
    // . . .
  }
}
```

Thus, methods `keySet()`, `values()`, and `entrySet()` offer different collection perspectives on the data stored in a `Map`. The implementation of these three methods in $\mathcal{L}$-`HashMap` relies on the fact that class `HashMap.Entry` (the type of entries in the hash table data structure) **`implements`** interface `Map.Entry`. Specifically, each of these implementations returns an instance of class `AbstractCollection`—a minimal and not-too-efficient implementation of a collection, whose underlying representation is a meager iterator over the entries in the collection. Function `entrySet()`, for example, returns an instance of `AbstractSet` wrapped around an iterator over the entire set of hash table entries.

[3] We tacitly ignore in this discussion the slight complication raised by the necessity to return the value associated with removed key.

The difficulty in redoing these methods for $\mathcal{F}$-`HashMap` is that we cannot iterate directly over the entries in a bucket of a hash table, since several entries are fused together into a single object. Instead, we define an **`abstract`** class named `VirtualEntry` as depicted in Listing 4.4.2.

Listing 4.4.2 Class `VirtualEntry`.

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

Class `VirtualEntry` will offer a `Map.Entry` *view* on the fields defined within a fused bucket, where a fused bucket object would typically have a number of such views. Most importantly, method `next()` in this class makes it possible to proceed to the next view, which can be either within the same object or in a object subsequent to it.

We realize these views as non-**`static`** inner classes of classes `Bucket1`, ..., `Bucket4`. (As we shall see, the implicit inclusion of members of the enclosing class in the inner class contributes makes the code a bit more succinct.)

Figure 4.4.1 is a UML [6] class diagram portraying the essentials of the implementation. We see in the diagram the four different views: `E1`, `E2`, `E3` and `E4`, and the classes in which these are defined.

To understand how these views are created, start with class `Bucket` which other than the `get`, `add` and `remove` methods discussed above, defines factory methods for the each of these views. More details are offered by Listing 4.4.3.

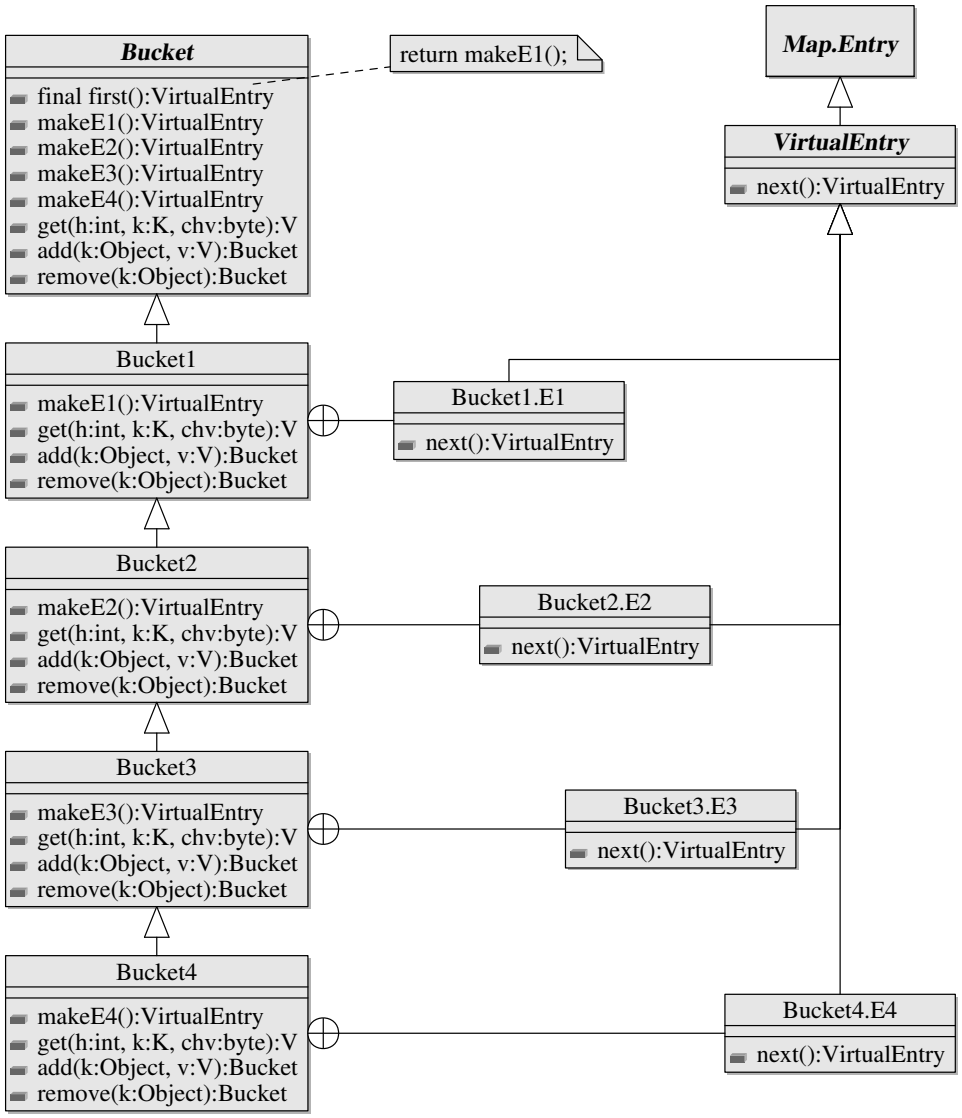

Figure 4.4.1: A UML class diagram for virtual entry views on fused buckets.

Listing 4.4.3 Obtaining the first virtual entry perspective of a fused bucket, and factory methods in class `Bucket`.

```
abstract class Bucket<K, V> {
   final VirtualEntry<K, V> first() { return makeE1(); }
   VirtualEntry<K, V> makeE1() { return null; }
   VirtualEntry<K, V> makeE2() { return null; }
   VirtualEntry<K, V> makeE3() { return null; }
   VirtualEntry<K, V> makeE4() { return null; }
  //...
}
```

The **final** method `first` in class `Bucket` calls the first of these factory methods to return the first virtual entry stored in a fused bucket.

Each of the classes `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, class `Bucket2` (Listing 4.4.4) ***(i)*** inherits the view `E1` from class `Bucket1` ***(ii)*** defines the view `E2`, and, ***(iii)*** overrides the factory method `makeE2` to return an instance of this view.

Listing 4.4.4 Virtual entry class `E2` and its enclosing class `Bucket2`.

```
class Bucket2<K, V> extends Bucket1<K, V> {
  // ...
  @Override final VirtualEntry<K, V> makeE2() {
    return new E2();
  }
  final class E2 extends VirtualEntry<K, V> {
    public K getKey() { return K2; }
    public final V getValue() { return V2; }
    void setV(V v) { V2 = v;}
    VirtualEntry<K, V> next() { return makeE3(); }
  }
  // ...
}
```

The methods in the inner class `Bucket2.E2` are truly simple accessors: all they do is return set or get interface to the appropriate fields.

Take note that method `next()` in class `Bucket2.E2` calls the `makeE3()` factory method to generate the next view. Even though this factory method returns **null** in this class, it is overridden in class `Bucket3` to return an instance of the view `Bucket3.E3`.

Similarly, class `Bucket4.E4` is depicted in Listing 4.4.5.

Listing 4.4.5 Virtual entry class `VE4` of class `Bucket4`.

```
class Bucket4<K, V> extends Bucket3<K, V> {
  //...
  final VirtualEntry<K, V> makeE4() {
    return new E4();
  }
  final class E4 extends VirtualEntry<K, V> {
    public K getKey() { return K4; }
    public V getValue() { return V4; }
    void setV(V v) {V4 = v; }
    VirtualEntry<K, V> next() {
      return next == null ? null : next.first(); }
  }
  //...
}
```

Observe that the `next()` method in this class is a bit special: if the `next` field is not `null` it returns the first view of the bucket object that follows. (Recall method `first()` of class `Bucket`, Listing 4.4.3.)

Chapter 5

Squashed Buckets Hashing

This section discusses $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`, a re-implementation of `HashMap` and `HashSet` designed to further improve the memory requirements of fused hashing, as described in the previous section.

This additional saving is achieved by a more thorough consolidation and the observation that a singleton bucket does not need to be represented by an object. Consider a certain cell in the main array used for representing the hash table data structure, and suppose that the bucket that belongs in this cell is a singleton. Then, instead of storing in the cell a reference to a singleton bucket object, this cell may directly reference the single key residing at this bucket. The value associated with this key is consolidated into an additional array.

Thus, a hash map consists of three arrays: `keys` and `values`, both of length m and both with entries of type `Object`, and, as before, a **`byte`** array of the same length of array named `chv` for storing the CHV of the first key in buckets.

If the $\texttt{i}^{th}$ bucket is empty, then both `keys[i]` and `values[i]` are **`null`**. If this bucket is a singleton, then `keys[i]` is the key stored in this bucket, `chv[i]` is the CHV of this key, and `values[i]` is the value associated with 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 still stored in

`values[i]`, and h_1 is still stored in `chv[i]`.[1]

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, but this extra saving seems to be infinitesimal.)

5.1 Squashed Buckets Layout

Table 5.1.1 lists the introduced -and pulled-up- fields in classes `Bucket`, `Bucket2`, ..., `Bucket6` in the HotSpot32 memory model.

	Hash Map		**Hash Set**	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	object header	8	object header	8
`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
`Bucket3`	K_3, V_3	32		24
`Bucket4`	K_4, V_4	40	$K_4, \uparrow K_5$	32
`Bucket5`	K_5, V_5	48		32
`Bucket6`	`next`	56	`next`	40

Table 5.1.1: Content and size of `Bucket` ... `Bucket6` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` (HotSpot32).

The implementation of `HashMap` in this model includes a pull-up into `Bucket2` of the **`byte`**-sized h_3, h_4 and h_5 (which belong, respectively, in `Bucket3`, `Bucket4` and `Bucket5`).

[1] Observe that 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`.

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]`). Fields pull-up may be more extensive than in `HashMap`, as demonstrated in the last two columns of Table 5.1.1: the pull-up into `Bucket2` includes not only h_3, h_4 and h_5, but also of K_3. In addition, K_5 is pulled up from `Bucket5` into `Bucket4`.

Table 5.1.2 repeats the contents of Table 5.1.1 for HotSpot64. As before, pull-up opportunities are more limited with HotSpot64 since the width of a reference is the same as the alignment width.

	Hash Map		**Hash Set**	
	introduced fields	*total size (bytes)*	*introduced fields*	*total size (bytes)*
`Bucket`	—	16	—	16
`Bucket2`	$K_1, K_2, V_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	48	$K_1, K_2, h_2, \uparrow h_3, \uparrow h_4, \uparrow h_5$	40
`Bucket3`	K_3, V_3	64	K_3	48
`Bucket4`	K_4, V_4	80	K_4	56
`Bucket5`	K_5, V_5	96	K_5	64
`Bucket6`	`next`	104	`next`	72

Table 5.1.2: Content and size of `Bucket` ... `Bucket6` of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` (HotSpot64).

Comparing Table 5.1.1 with Table 4.1.1 and Table 5.1.2 with Table 4.1.2 reveals that the consolidation of values into an array does not only lead to the elimination of a `Bucket` object for the singleton bucket, i.e., no class `Bucket1` in $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet`; other fused `Bucket` objects may benefit in size. For example, on the HotSpot32 memory model, a `Bucket2` of $\mathcal{F}$-`HashMap` requires 32 bytes, but only 24 bytes on $\mathcal{S}$-`HashMap`. On HotSpot64, a `Bucket3` of $\mathcal{F}$-`HashMap` requires 72 bytes, and 64 bytes on $\mathcal{S}$-`HashMap`. This saving is traded off by additional memory for the three arrays constituting the hash table.

Observe that since singleton buckets do not occupy any memory, the non-decreasing compression property is violated: In other words, a key distribution in which all keys fall in distinct buckets is the most memory efficient among all other distributions.

5.2 Searching in a Squashed Bucket

As before, 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. However, unlike $\mathcal{F}$-`HashMap`, we cannot implement a search in a squashed bucket by simply sending a `get` message to the appropriate `Bucket` object, and relying on dynamic dispatch to carry out the correct sequence of comparisons. The difficulty is that the squashed implementation does not have `Bucket1` objects which can carry out this dispatch. A singleton bucket is represented by a pointer to the key in the `keys` array.

A possible solution would be to rely on the **`instanceof`** operator, as we did with the hand optimized version of `get` for $\mathcal{F}$-`HashMap`. This solution however may be inefficient due to the addition of runtime checking for K_1 comparison. Adding an **`instanceof`** check may (unnecessarily) slow down the search time in a singleton bucket or in cases where the searched key is not `equals` to K_1. We decided to avoid additional explicit runtime check by overriding the unused `equals` method of class `Bucket`. Listing 5.2.1 and Listing 5.2.2 show the details.

Listing 5.2.1 JAVA code for searching a given key in a squashed bucket (buckets with 0, 1 or 2 keys).

```
public V get(Object κ) {
  if (κ == null) κ = nullKey;
  int h = hash1(κ.hashCode());
  final 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;
  final Bucket2<V> b = (Bucket2) k;
  if (b.h2 == h && ((k = b.K2) == κ || κ.equals(k)))
    return b.V2;
  if (!(b instanceof Bucket3)) return null;
  return get3(κ, h, b);
}
```

Listing 5.2.2 JAVA code for equality test against a squashed `Bucket` object.

```
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);
  }
  //...
}
```

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 the method `equals` described in Listing 5.2.2. Even though this call may be expensive it is done only if $h == h_1$. If necessary 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.

As before, the case of buckets with more than two keys, and comparison to K_3, K_4 is by chaining to method `get3`. The implementation of method `get3` in $\mathcal{S}$-`HashMap` are pretty much the same as `get3` in $\mathcal{F}$-`HashMap` (Listing 4.2.2) except that it takes care of the extra two cases `Bucket5` (comparison with K_5) and `Bucket6` (initiating a search in a list of ordinary entries at the bucket's tail).

Updates to the hash table data structure were implemented in the same fashion as searches, i.e., by hand written code which uses **`instanceof`** to determine the bucket type. An alternative would have been to use hand written code only for the irregular case of singleton buckets, and use dynamic dispatch for the remaining cases, `Bucket2`, ..., `Bucket6`, i.e., defining **`abstract`** methods `add` and `remove` in class `Bucket` with appropriate implementations, selected by dynamic dispatch, in each of `Bucket2`, ..., `Bucket6`. Each of these methods would reconstruct the fused object, and return a pointer to the newly created object.

5.3 Iteration over Squashed Table Entries

The virtual entry views technique carries almost as is from $\mathcal{F}$-`HashMap`, making it possible to conduct a topology preserving iteration over hash table entries, with disregard to the fact that the key of the first entry in each bucket may be stored directly in a cell of array `keys` or in a `Bucket` object whose reference is contained in this cell.

Figure 5.3.1 is a UML class diagram showing the classes and methods pertinent for the creation of the virtual entries views necessary for this iteration.

Much of Figure 5.3.1 is mundane and very similar to Figure 4.4.1. In fact, classes `Bucket`, `Bucket2`, ..., `Bucket6` and their inner virtual entry classes `E2`, ..., `E5` were 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`. (Observe that in forward iteration from class `Bucket6`, no fused objects will be encountered, and hence there is no need for a `E6` class; method `next` of class `E5` may return either a **`null`** or an instance of the vanilla `HashMap.Entry` class.)

More interesting is class `HashMap.E1`, a non-static inner class of the squashed implementation of class `HashMap`, shown in Listing 5.3.1

Listing 5.3.1 Virtual entry class `HashMap.E1`, representing the entry in a singleton bucket at position `i`.

```
class HashMap<K, V> implements Map<K, V> {
  //...
  class E1 extends VirtualEntry<K, V> {
    private int i;
    E1(int i) { this.i = i; }
    public K getKey() { return (K) keys[i]; }
    public V getValue() { return values[i]; }
    void setV(V v) { values[i] = v; }
    VirtualEntry<K, V> next() { return null; }
  }
  //...
}
```

Class `HashMap.E1` saves the integer `i` passed to its constructor, as means for accessing later a singleton bucket residing at the $\texttt{i}^{th}$ position. Both the key and the value are retrieved by simple array access.

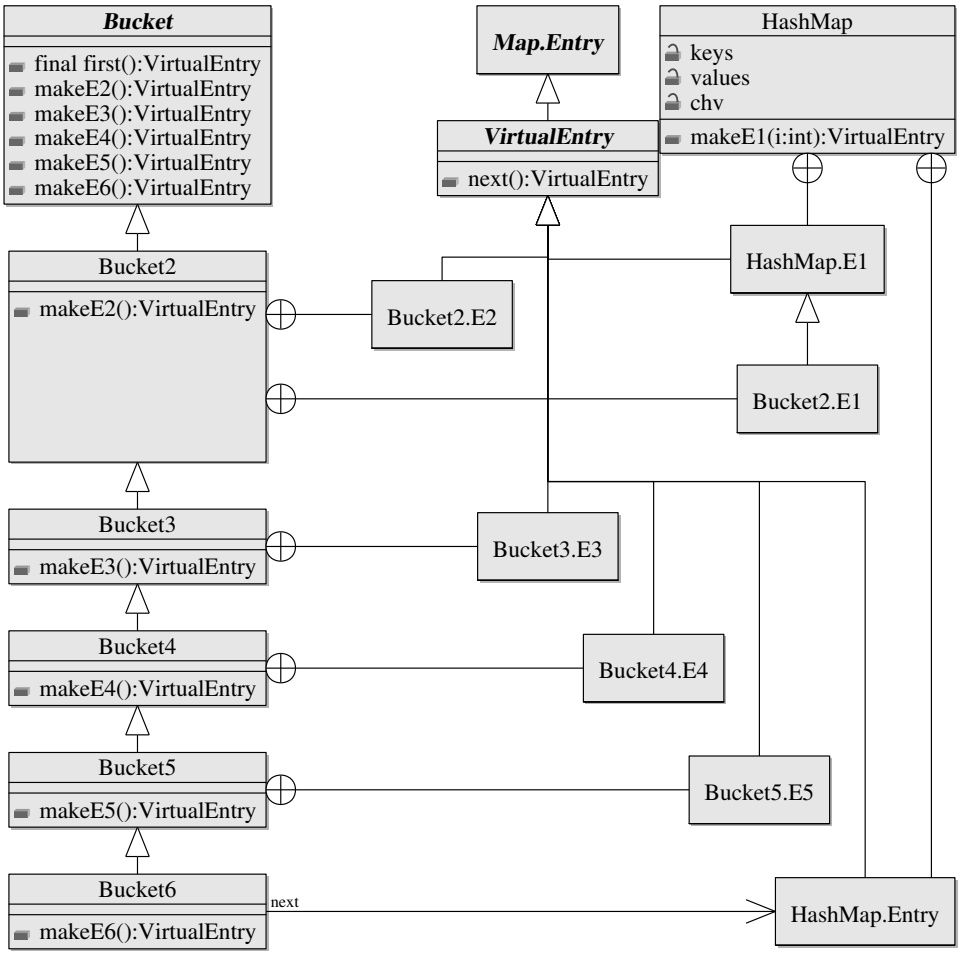

Figure 5.3.1: A UML class diagram for virtual entry views on squashed buckets.

A virtual entry view on the first key of a non-singleton bucket is represented by class `Bucket2.E1`, depicted in Listing 5.3.2.

The implementation of class `Bucket2.E1` is by extension of `HashMap.E1`, i.e., it is an inner class of *both* class `Bucket2` and class `HashMap`. The super class saves a reference to the entire hash table (the inner class pointer), and index `i`. Class `Bucket2.E1` inherits the setter and the getter for the value, from `HashMap.E1`, and overrides the methods for retrieving the key and for proceeding to the next virtual entry.

Listing 5.3.2 Virtual entry class `Bucket2.E1`, representing the first entry of a non-singleton bucket as position `i`.

```
class Bucket2<K, V> extends Bucket<K, V> {
  K K1, K2;
  V V2
  byte h2, h3, h4, h5;
  //...
  final class E1 extends HashMap<K, V>.E1 {
    E1(int i, HashMapV4<K, V> table) { table.super(i); }
    public K getKey() { return K1; }
    VirtualEntry<K, V> next() { return makeE2(); }
  }
  //...
}
```

The choice between `Bucket2.E1` and `HashMap.E1` is carried out by method `makeE1` of `HashMap` (Listing 5.3.3).

Listing 5.3.3 Method `makeE` of class `HashMap` distinguishing between a singleton bucket and the first entry of a non-singleton bucket.

```
VirtualEntry<K, V> makeE1(int i) {
  // assumes keys[i] != null
    return keys[i] instanceof Bucket2 // a non−singleton bucket?
      ? ((Bucket2<K, V>)keys[i]).new E1(i, this) // Yes
      : new E1(i); // No
  }
```

Given `i`, an index of a table entry, the single statement in this method examines the runtime type of `keys[i]`. If this is an object of type `Bucket2`, the method creates a virtual entry composed of the value residing in the `values` array and the key residing in the bucket object. Otherwise, a simple virtual entry view of the singleton bucket is created by instantiating class `HashMap.E1`.

Chapter 6

Memory and Time Performance of Fused and Squashed Hashing

Having described the essential details of the implementation of fused and squashed hashing, we can turn to evaluating the space and time performance of these. For space, 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` (Listing 3.1.1), compensates for suboptimal overriding implementations of `hashCode`.

Note that a distribution of keys among the buckets which is not random means that buckets tend to be fewer and larger than what is 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 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 of $\mathcal{L}$-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 and HotSpot64 (exact settings are described below).

6.1 Space Overhead of $\mathcal{F}$-hash

We start with an analysis of $\mathcal{F}$-`HashMap` on the HotSpot32 memory model. In this implementation, we have that arrays `table` and `chv` consume together $(4+1)m$ bytes, regardless of n—the number of entries stored in the table. The contribution of these two arrays to each table entry is hence

$$5m/n = 5/p. \tag{6.1.1}$$

(Note that this overhead is the same for both $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet`.)

For the analysis of memory consumed by the buckets, let us denote by μ_k the *total number of bytes required for the representation of a bucket of size* k. Then, μ_k/k is the number of bytes that each individual entry in a k-sized buckets incurs. Since $\beta_k(p)$ is the expected fraction of entries falling in buckets of size k, we have that the expectation of the buckets-memory contribution is

$$\sum_{k=1}^{\infty} \beta_k(p) \frac{\mu_k}{k} \tag{6.1.2}$$

bytes to each table entry.

The implementation of fused hashing is such that μ_k depends chiefly

on k mod 4. Specifically, for $\mathcal{F}$-`HashMap`, we write $k = 4\ell + k'$, and

$$\mu_k = \begin{cases} 48\ell & \text{if } k = 4\ell \\ 48\ell + 16 & \text{if } k = 4\ell + 1 \\ 48\ell + 32 & \text{if } k = 4\ell + 2 \\ 48\ell + 40 & \text{if } k = 4\ell + 3 \end{cases}. \tag{6.1.3}$$

Substituting (6.1.3) into (6.1.2) and adding (6.1.1) yields that the expected number of bytes per table entry in $\mathcal{F}$-`HashMap` is

$$\frac{5}{p} + \sum_{\ell=0}^{\infty} \Big(\beta_{4\ell} \frac{48\ell}{4\ell} + \beta_{4\ell+1} \frac{48\ell + 16}{4\ell + 1} + \beta_{4\ell+2} \frac{48\ell + 32}{4\ell + 2} + \beta_{4\ell+3} \frac{48\ell + 40}{4\ell + 3} \Big),$$

which after substituting (3.3.3) and (3.3.4) and some tidying up converges to

$$12 + \frac{9}{p} - 4\frac{\cos p}{p} e^{-p}. \tag{6.1.4}$$

(The occurrence of a trigonometric function in the above is explained by the fact that the structure of long buckets is periodic, with period 4.)

Similarly, reading the final column of Table 4.1.1 we obtain for $\mathcal{F}$-`HashSet`

$$\mu_k = \begin{cases} 32\ell & \text{if } k = 4\ell \\ 32\ell + 16 & \text{if } k = 4\ell + 1 \\ 32\ell + 24 & \text{if } k = 4\ell + 2 \\ 32\ell + 24 & \text{if } k = 4\ell + 3 \end{cases}. \tag{6.1.5}$$

Substituting the above into (6.1.2) and adding (6.1.1) gives, after some tidying up, that the expected number of bytes per table entry in the implementation of $\mathcal{F}$-`HashSet` on HotSpot32 is

$$8 + \frac{9}{p} - 4\frac{\cos p - \sin p}{p} e^{-p}. \tag{6.1.6}$$

Asymptotically, i.e., as p tends to infinity, both (6.1.4) and (6.1.6) are superior to the memory requirements of $\mathcal{L}$-hash (3.2.1): for large values of p,

$\mathcal{F}$-`HashMap` requires about 12 bytes per entry, while $\mathcal{F}$-`HashSet` requires about 8 bytes per entry, compared to the 24 bytes required by $\mathcal{L}$-hash for large values of p.

The more interesting domain of smaller values of p is depicted in Figure 6.1.1 which plots (6.1.4) and (6.1.6) vs. p, comparing these to the $\mathcal{L}$-hash baseline (3.2.1).

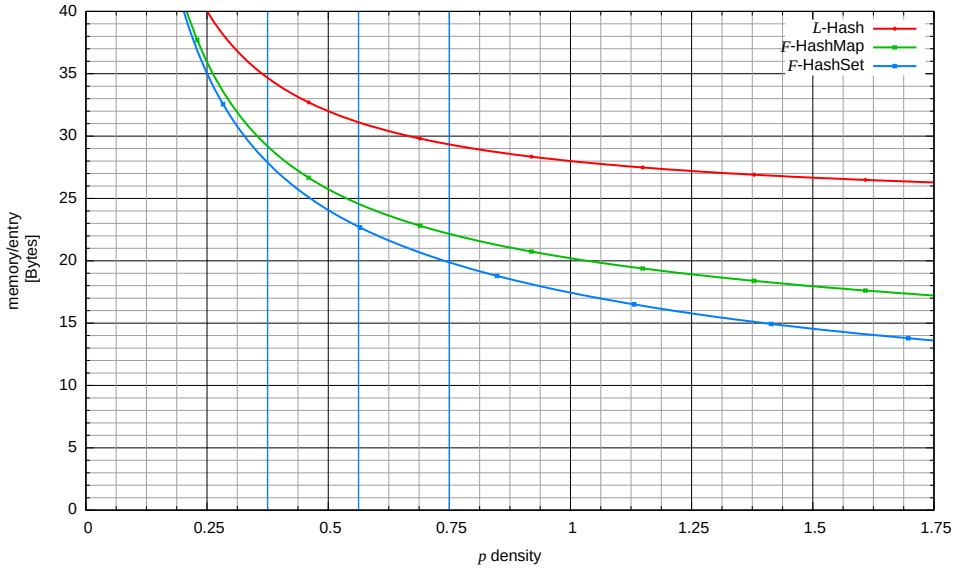

Figure 6.1.1: Expected memory per table entry in $\mathcal{L}$-hash, $\mathcal{F}$-`HashMap`, and $\mathcal{F}$-`HashSet` vs. table density (HotSpot32).

We see that throughout the entire "typical" range, list fusion improves memory used for the hash data structure, reducing it by about a third at $p = 3/4$, and that the improvement increases with p.

We now repeat the analysis of $\mathcal{F}$-hash for the HotSpot64 memory model. Observe first that arrays `table` and `chv` contribute

$$(8 + 1)/p = 9/p \tag{6.1.7}$$

bytes to each entry. For the values μ_k, we read the size of classes `Bucket1`, ...`Bucket4` in this model in the "total size" column of Table 4.1.2. Em-

ploying the same substitutions as above we obtain that in the HotSpot64 memory model, the expected number of bytes per table entry is

$$24 + \frac{9}{p} - 4\frac{\cos p - \sin p}{p}e^{-p} \tag{6.1.8}$$

for $\mathcal{F}$-`HashMap` and

$$16 + \frac{9}{p} - 4\frac{\cos p - \sin p}{p}e^{-p} \tag{6.1.9}$$

for $\mathcal{F}$-`HashSet`. For very dense tables we see again a reduction from 48 bytes down to 24 bytes for `HashMap`, and down to 16 bytes for `HashSet`. More interestingly, the typical range is explored in Figure 6.1.2 which compares (6.1.8) and (6.1.9) with (3.2.2).

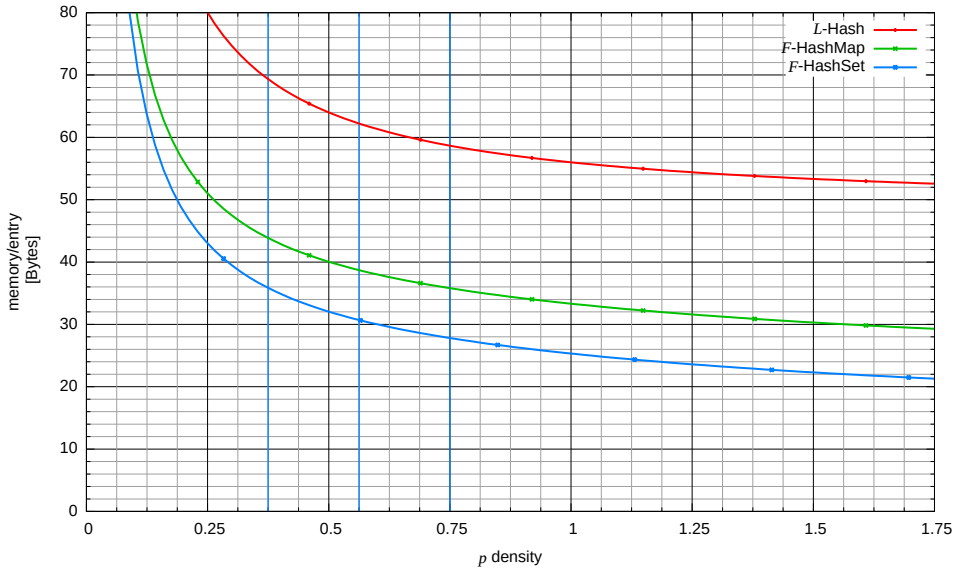

Figure 6.1.2: Expected memory per table entry in $\mathcal{L}$-hash, $\mathcal{F}$-`HashMap`, and $\mathcal{F}$-`HashSet` vs. table density (HotSpot64).

The relative memory savings witnessed in Figure 6.1.1 are seen again in Figure 6.1.2. Moreover, it is visually apparent that these improvements are more drastic for the 64-bit model. To emphasize this point, Figure 6.1.3 plots the savings in memory overhead of $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` compared to

$\mathcal{L}$-hash for both models.

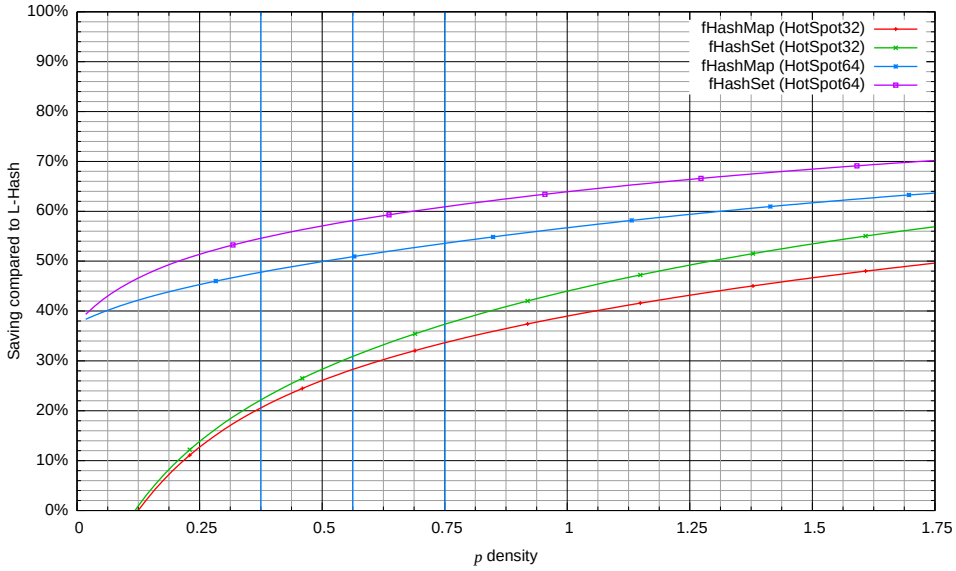

Figure 6.1.3: Expected saving in overhead memory of $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet` compared to $\mathcal{L}$-hash in HotSpot32 and HotSpot64.

Evidently, in the typical range, savings in overhead are 20%–40% for HotSpot32 and 45%–60% in HotSpot64.

6.2 Space Overhead of $\mathcal{S}$-hash

We observed that the more compact representation of buckets in $\mathcal{S}$-hash is made possible by consuming more memory in the fixed `keys` and `values` tables. Let us start (again) with the implementation of `HashMap` on the HotSpot32 memory model. In these settings, arrays `keys`, `values` and `chv` contribute together

$$(4 + 4 + 1)/p = 9/p \tag{6.2.1}$$

bytes for each table entry (compared to $5/p$ for $\mathcal{F}$-`HashMap`).

Three cases are to be distinguished in computing μ_k in $\mathcal{S}$-hash. First, singleton buckets are represented solely by the appropriate cells in arrays

`keys`, `values` and `chv`—no additional bytes are required for the bucket itself, i.e.,

$$\mu_1 = 0. \tag{6.2.2}$$

Secondly, the memory used by a bucket of size k, for $k = 2, 3, 4, 5$ is given by the sequence of values in the third column of Table 5.1.1, i.e.,

$$\mu_k = 8(k+1) \qquad \text{for } k = 2, 3, 4, 5. \tag{6.2.3}$$

Finally, a bucket of size k, $k > 6$, requires 56 bytes for the fixed `Bucket6` object, and additional 24 bytes for each entry beyond the fifth, i.e.,

$$\mu_k = 56 + 24(k-5) \qquad \text{for } k = 6, 7, \ldots. \tag{6.2.4}$$

We now substitute (6.2.4), (6.2.3), and (6.2.2) into (6.1.2), and add (6.2.1) to obtain the expected number of bytes per entry,

$$\frac{9}{p} + \frac{0 \cdot \beta_1(p)}{1} + \sum_{k=2}^{5} 8\frac{k+1}{k} \cdot \beta_k(p) + \sum_{k=6}^{\infty} \frac{56 + 24(k-5)}{k} \cdot \beta_k(p),$$

which, after a bit of tidying converges to,

$$24 - \frac{55}{p} + \left(\frac{64}{p} + 40 + 20p + 4p^2 + \frac{p^3}{3} - \frac{p^4}{15}\right) e^{-p}. \tag{6.2.5}$$

Redoing this analysis for $\mathcal{F}$-`HashSet` we find (again) that $\mu_1 = 0$ and that $\mu_2 = 24$, $\mu_3 = 24$, $\mu_4 = 32$, $\mu_5 = 32$, while

$$\mu_k = 40 + 24(k-5)$$

for $k > 5$. We obtain that the expected number of bytes per entry is

$$24 - \frac{75}{p} + \left(\frac{80}{p} + 56 + 28p + \frac{16p^2}{3} + \frac{2p^3}{3} - \frac{p^4}{15}\right) e^{-p}. \tag{6.2.6}$$

Due to our inefficient implementation of the tail of non-fused buckets $k > 6$, asymptotically, as p approaches infinity, the memory overheads of both F-Map (6.2.5) and F-Set is the same as that of L-Hash, i.e., 24 bytes per table entry.

The range of smaller, and more interesting values of p is depicted in Figure 6.2.1 which plots (6.2.5) and (6.2.6) vs. p, while comparing these to the memory requirements of $\mathcal{F}$-`HashMap`, $\mathcal{F}$-`HashSet`, and $\mathcal{L}$-hash.

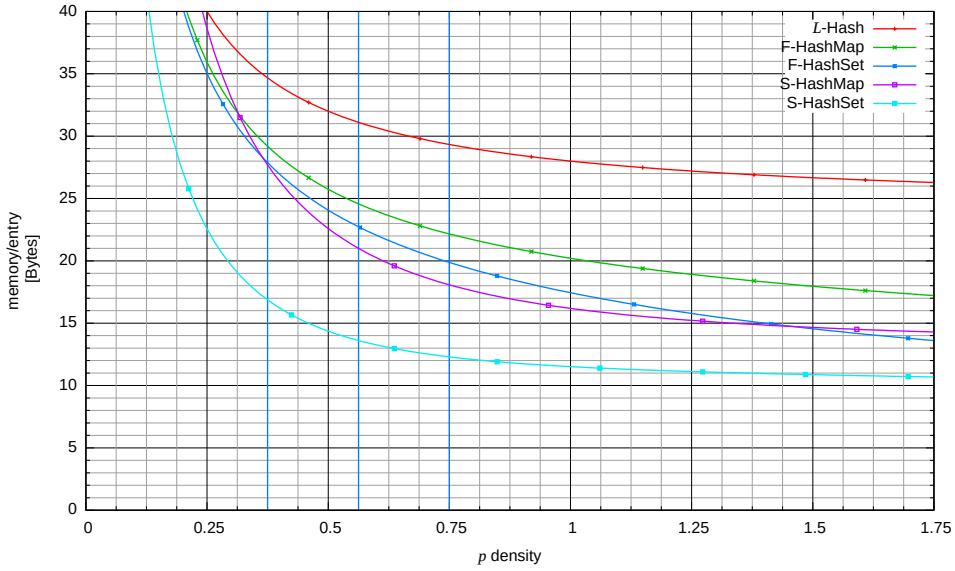

Figure 6.2.1: Expected memory per hash table entry of $\mathcal{F}$-`HashMap`, $\mathcal{F}$-`HashSet`, $\mathcal{S}$-`HashMap`, $\mathcal{S}$-`HashSet`, and $\mathcal{L}$-hash vs. table density (HotSpot32).

Observe that the squashed implementation is superior, memory-wise, to L-Hash, throughout the entire depicted range: for higher values of p, where $1 \leq p \leq 1.75$, memory per entry is reduced from about 27 bytes to about 15 bytes for `HashMap`, and to about 11 bytes for `HashSet`. At the typical range, the number of bytes per entry is reduced from 34.7 to 27.7 ($\mathcal{S}$-`HashMap`) and 16.9 ($\mathcal{S}$-`HashSet`) at $p = 3/8$, and from 29.3 to 18.1 ($\mathcal{S}$-`HashMap`) and 12.3 ($\mathcal{S}$-`HashSet`) at $p = 3/4$, the latter representing almost 60% savings.

We also see that $\mathcal{S}$-`HashSet` is the most memory efficient of all implementations, and that in general, $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` are more memory efficient than $\mathcal{F}$-`HashMap` and $\mathcal{F}$-`HashSet`. Curiously, in the typical domain, $\mathcal{S}$-`HashMap` is even more memory efficient than $\mathcal{F}$-`HashSet`, despite the fact a map data structure needs to store additional value information, and despite

the fact that the arrays overhead is greater in squashed hashing.

In analyzing $\mathcal{S}$-hash on the HotSpot64 memory model, we note first that arrays contribute $(8+8+1)/p = 17/p$ bytes to each table entry in $\mathcal{S}$-`HashMap`, and $(8+1)/p = 9/p$ bytes to each entry in $\mathcal{S}$-`HashSet`. To compute μ_k, we read again the "total bytes" columns of Table 5.1.2. Redoing the above steps, we obtaining that the expected number of bytes per table entry is

$$48 - \frac{119}{p} + \left(\frac{136}{p} + 88 + 44p + \frac{28p^2}{3} + p^3 - \frac{p^4}{15}\right) e^{-p} \tag{6.2.7}$$

for $\mathcal{S}$-`HashMap` and

$$32 - \frac{79}{p} + \left(\frac{88}{p} + 56 + 32p + \frac{20p^2}{3} + \frac{2p^3}{3} - \frac{p^4}{15}\right) e^{-p} \tag{6.2.8}$$

for $\mathcal{S}$-`HashSet`.

Note that even for dense tables (i.e., p approaching infinity) $\mathcal{S}$-`HashSet` is memory-wise superior to $\mathcal{L}$-hash, achieving 24- instead of 48- bytes per table entry.

Figure 6.2.2 repeats Figure 6.2.1 but for the HotSpot64 memory model, i.e., plotting (6.2.7) and (6.2.8).

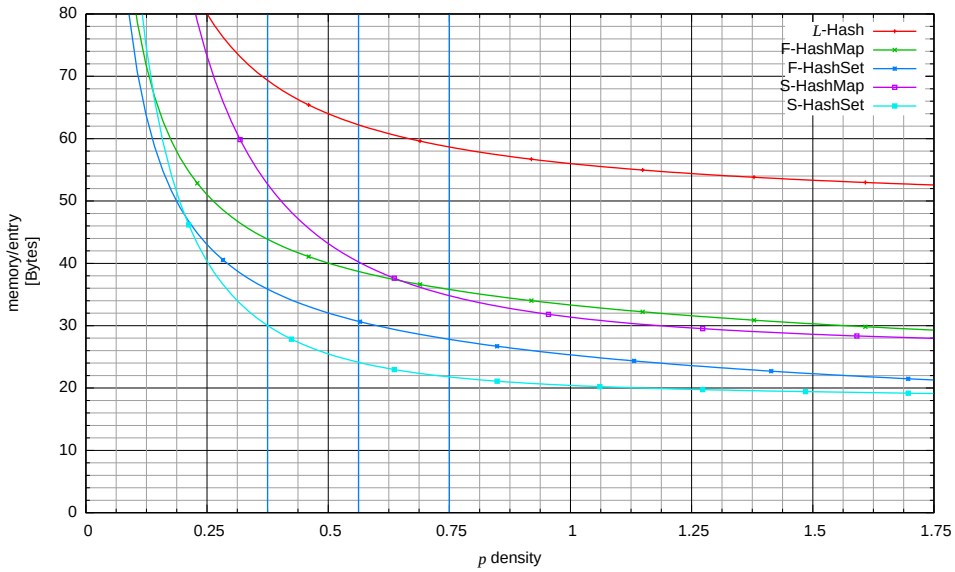

Figure 6.2.2: Expected memory per hash table entry of $\mathcal{F}$-`HashMap`, $\mathcal{F}$-`HashSet`, $\mathcal{S}$-`HashMap`, $\mathcal{S}$-`HashSet`, and $\mathcal{L}$-hash vs. table density (HotSpot64).

Examining the figure, we can make the same qualitative judgment as in Figure 6.2.1, i.e, that squashed hashing is more memory efficient than fused hashing, which in its turn, beats $\mathcal{L}$-hashing. One important exception to this claim is that $\mathcal{F}$-`HashMap` is more memory efficient than $\mathcal{S}$-`HashMap` in the lower and more common values of p, due to the high fraction of empty `values` array entries.

Moreover, comparing figures 6.2.2 and 6.2.1 we see that both the *absolute* and *relative* savings in using memory aware implementations are greater in the 64-bit memory model than in the 32-bits model.

Relative savings of $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` are presented in Figure 6.2.3.

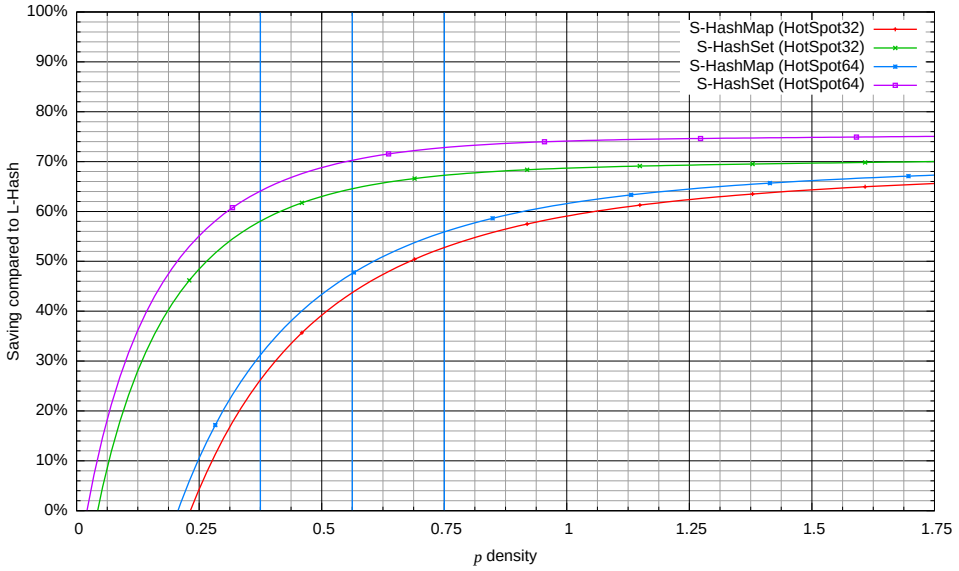

Figure 6.2.3: Expected saving in overhead memory of squashed hashing compared to L-hashing) in HotSpot32 and HotSpot64.

Comparing these plots to those of Figure 6.1.3 we can appreciate the further memory advantage offered by $\mathcal{S}$-hash. Savings in overhead in the typical range are 30%–50% in the HotSpot32 memory model and on the 60%-70% range in HotSpot64.

6.3 Time

Benchmarking in JAVA, in particular micro-benchmarking, is a delicate task. The extra abstraction layer posed by the virtual machine, the JIT compilation cycles and the asynchronous garbage collection are the main reasons that make it perhaps impossible to introduce deterministic reproducible results. Our endless attempts to micro-benchmark $\mathcal{F}$-hash and $\mathcal{S}$-hash did not yield conclusive conclusions. A case study and findings we have encountered are discussed in details in [16]. These findings were: (*i*) instability of the Virtual Machine. (*ii*) multiple and even multiple simultaneous steady states, and (*iii*) prologue effects (executing irrelevant code prior to the benchmarked code).

The latter is one of our most important conclusions: The timing results obtained in a clean benchmarking environment may be meaningless when the benchmarked code is used in an application. For this reason and since we were unable to introduce conclusive results we've diverged our effort in benchmarking our implementations using well known large-scale JVM benchmarking tools. These are SPECjvm2008, SPECjbb2005 and daCapo 2009.

Measurements were done using two versions of Linux Mint 11 (Katya) OS: a 32-bit version and a 64-bit version. Hardware used: Intel Core i3 processor with 2.93 GHz and 2GB RAM. Benchmarked code was compiled using Eclipse Helios. The JRE version was 1.6.0_26-b03 with HotSpot Server VM 20.1-b02, mixed-mode.

In the 32-bit environment we've used $\mathcal{S}$-`HashMap` and $\mathcal{S}$-`HashSet` implementations as they both introduced the best memory reduction opportunity. In the 64-bit environment we've used $\mathcal{F}$-`HashMap` instead of $\mathcal{S}$-`HashMap` as it offers better memory reduction in the common range of p. All benchmarks were done in a clean environment (single user mode with no GUI and no background applications running), where only one CPU was enabled.

First we present the SPECjvm2008 benchmarking results. Benchmark setup was as follows: JVM arguments -Xms1500m -Xmx1500m, non-default application arguments: -ict -ikv --peak -bt 1. The SPECjvm2008 benchmark contains 11 sub benchmarks which were executed 10 times each, both for the baseline implementation and the benched implementation. In the 32-bit version the average baseline result was 30.48 ops/m ($\pm$0.31%) while the average benched result was 30.318 ops/m ($\pm$0.31%), a degradation of 0.53%. In the 64-bit version the average baseline result was 35.38 ops/m ($\pm$0.25%) while the average benched result was 35.04 ops/m ($\pm$0.17%), a degradation of 0.96%.

Second benchmark was SPECjbb2005. Benchmark setup was as follows: JVM arguments -Xms256m -Xmx256m, non-default application argument: `input.deterministic_random_seed=true`. The SPECjbb2005 was executed 10 times each, both for the baseline implementation and the benched implementation. In the 32-bit version the average baseline result was 31,536 ops/m ($\pm$0.42%) while the average benched result was 31,293 bops ($\pm$0.34%), a degradation of 0.77%. In the 64-bit version the average

baseline result was 32.779 ops/m (±1.60%) while the average benched result was 31,900 bops (±1.26%), a degradation of 2.68%.

Finally we've tested the daCapo benchmark. Benchmark setup was as follows: JVM arguments -Xms1500m -Xmx1500m, non-default application arguments: --no-validation -C -t 1. We've used 9 sub benchmarks in the 32-bit version and 10 sub benchmarks in the 64-bit version. Each benchmark was executed 10 times, both for the baseline implementation and the benched implementation. These benchmarks were: avrora, batik, eclipse, h2, jython, luindex, lusearch, pmd, sunflow and xalan (in 64-bit). In the 32-bit version the average baseline result was 5,870 msec (±5.18%) while the average benched result was 5,906 msec (±3.11%), a negligible improvement of 0.60%. In the 64-bit version the average baseline result was 5,557 msec (±4.24%) while the average benched result was 5,528 msec (±5.38%), a degradation of 0.51%.

We conclude that our proposed implementations remain withing practical runtimes, imposing minimal and negligible degradation to the JVM in compare to the base implementation, while imposing significant memory overhead reduction. These slowdowns do not come as a surprise as the common usage of hash tables is as tiny (less than 16 entries) collections[1]. 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, in compare to the almost trivial baseline implementation of them.

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

Chapter 7

Compaction of Balanced Binary Tree Nodes

In this section, we describe two schemes for more compact representation of tree nodes of `TreeMap` (and `TreeSet`), i.e., class `Entry` (Listing 1.2.1). First we describe fusion, combined with null-pointer and boolean eliminations. Second we describe complete fields consolidations, which is no more than a representation of the red-black `TreeMap` (and `TreeSet`) using arrays of `int`s.

7.1 Employing Fusion, Null Pointer Elimination and Boolean Elimination

Examining Listing 1.2.1 we see a number of compaction opportunities: First, it is well known that half of the children edges in binary trees (fields `left` and `right` in `TreeMap.Entry`) are `null`. It therefore makes sense to apply null-pointer-elimination to these. Similarly, field `color` is a candidate for boolean elimination. Applying both eliminations methods, we obtain a more compact representation of tree leaves. These are realized by defining specialized classes for red-leaves and black leaves, as in Listing 7.1.1.

Listing 7.1.1 Abstract class `Node` and null pointer elimination combined with boolean elimination in specialized leaf nodes

```
interface VirtualNode<K, V> extends Map.Entry<K, V> {
  public boolean color();
  public VirtualNode<K, V> parent();
  public VirtualNode<K, V> left();
  public VirtualNode<K, V> right();
  // . . .
}
abstract static class Node<K, V> implements VirtualNode<K, V> { // Base class of all tree nodes.
  final static boolean BLACK = true, RED = !BLACK; // colors
  K key; V value; // contents
  InternalNode<K, V> parent; // topology
  Node(K k, V v, InternalNode<K, V> p) { key = k; value = v; parent = p; } // ctor
  @Override final public VirtualNode<K, V> parent() { return parent; }
  abstract Node<K, V> color(boolean c);
  // . . .
}
abstract static class Leaf<K, V> extends Node<K, V> {
  Leaf(K k, V v, InternalNode<K, V> p) { super(k,v,p); } // ctor
  @Override final Node<K, V> left() { return null; }
  @Override final lNode<K, V> right() { return null; }
  @Override 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 BlackLeaf<K, V>(this)
        : new RedLeaf<K, V>(this);
    // . . .
    return l;
  }
  // . . .
}
static final class RedLeaf<K, V> extends Leaf<K, V> {
  RedLeaf(Node<K, V> l) { super(l.key, l.value, l.parent); }
  @Override final public boolean color() { return RED; }
  // . . .
}
static final class BlackLeaf<K, V> extends Leaf<K, V> {
  BlackLeaf(Node<K, V> l) { super(l.key, l.value, l.parent); }
  @Override final public boolean color() { return BLACK; }
  // . . .
}
```

Class `Node` in this listing, which will serve as the base class of all specialized tree node classes, defines `key`, `value` and `parent` fields, just like `TreeMap.Entry`. (We discuss `InternalNode` later on) Fields `left`, `right` which might be eliminated, are represented only as **`abstract`** getter functions, and consume no object space. The subclass `Leaf` overrides these functions to return **`null`**, again consuming no object space.

The `color` field is modeled as **`abstract`** getter and setter methods. The contract of the setter function `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. This setter is then given body in class `Leaf`, which creates either a `RedLeaf` or `BlackLeaf` objects as necessary.

During updates to the tree, internal nodes may become leaves, and leaves may turn into internal nodes. The code that supports these operations is part of the insertion/removal code, which are not shown here.

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

Our measurements indicate that $\approx 42.8\%$ of the nodes in a red-black tree are leaves, and that this ratio is independent of tree size, and of the manner in which it was created. (This fraction is not a coincident, and similar values 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, and balancing of course contributes to the increasing of the number of leaves.)

This large fraction of leaves contributes to significant savings. Just by using classes `RedLeaf` and `BlackLeaf`, an implementation of `TreeMap` on HotSpot32 would use on average 20.6 bytes of overhead instead of 24 (14% savings), and 37.7 bytes on HotSpot64 (instead of 48, 21% savings). Similarly, the average overhead of `TreeSet` on HotSpot32 would be 21.2 bytes per entry (instead of 28) and 42.3 bytes on HotSpot64 (instead of 56)—both making a 24% savings.

A straightforward generalization of Listing 7.1.1 would extend boolean elimination to internal nodes as well. Such extension would not benefit `TreeMap` on HotSpot32, but would contribute to savings on the other three combinations. The implementation is described in Listing 7.1.2.

Listing 7.1.2 Abstract class `InternalNode` and boolean elimination in specialized internal nodes

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

Table 7.1.1 tabulates savings in overhead per entry due to the use of either *(i)* pointer elimination (in leaf nodes only) or *(ii)* boolean elimination (in all nodes), and *(iii)* applying both techniques, in the implementation of `TreeMap` and `TreeSet` for HotSpot32 and HotSpot64.

	HotSpot32			**HotSpot64**		
	pointer elimination	*boolean elimination*	*both*	*pointer elimination*	*boolean elimination*	*both*
`TreeMap`	14%	–	14%	21%	17%	31%
`TreeSet`	24%	29%	41%	24%	29%	41%

Table 7.1.1: Computed saving in memory overhead per tree entry due to the use of pointer elimination (in leaf nodes only), boolean elimination (on all nodes), and *both* in the implementation of `TreeMap` and `TreeSet` for HotSpot32 and HotSpot64.

As the table shows, savings are not additive, which means that the different combinations cannot be judged in isolation.

The saving is the smallest on the `HashMap`/`HotSpot32` combination. Luckily, fusion shall improve on that.

The tree topological structure is such that each node is "owned" by its parent. It therefore makes sense to apply fusion in tree nodes, just as we did for lists. The difficulty however is in dealing with the very many cases that could occur: If we apply a depth- ℓ fusion, one would need, in general, to define a specialized class for the $O(2^\ell)$ trees of this depth.

A more sane approach would be the fusion of nodes with their leaves. There are three cases to distinguish: *(i)* there are internal nodes which serve as parents to two leaves, *(ii)* there are internal nodes which serve as parents to a single leave, the other child being **`null`**, and *(iii)* there is the case in which a node is a parent both to a leaf and to a non-leaf non-**`null`** node, which we will ignore for now. Note that due to the characteristics of a red-black tree, there are only few possibilities for the colors of the nodes in cases (i) and (ii). For (i), if parent is RED children are both BLACK leaves. If parent is BLACK both children leaves are either RED or BLACK. For (ii), parent is BLACK while its leaf child is RED. These insights allow us to eliminate the boolean fields of these nodes, leading to five different (concrete) types of nodes.

To apply fusion for the first two cases we thus use five additional classes:

1. `ParentLeftLeaf` (black parent with a red left leaf),
2. `ParentRightLeaf` (black parent with a red right leaf),
3. `ParentLeavesRBB` (red parent with two black leaves),
4. `ParentLeavesBBB`,(black parent with two black leaves) and,
5. `ParentLeavesBRR` (black parent with two red leaves).

As before, we use virtual entry classes for easy traversal over these fused nodes. These are not described here. The first two classes have five pointer fields, as shown in Listing 7.1.3: 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`.

Listing 7.1.3 Classes for fusion of internal tree nodes with their leaf children

```
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. . .
}
```

The footprint of each 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, which gives 40 bytes footprint for each of its three subclasses on HotSpot32.

Our measurements show 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 therefore consume $32/2 = 16$ bytes each, $3 \cdot 9\% = 27\%$ 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 Listing 7.1.1, then saving increases to 43% for `TreeMap`.

We see that fusion is more effective than null-pointer and boolean elimination. The reason is that in fusing, e.g., three nodes into one, we eliminate two object headers, six tree nodes and three boolean fields. A practical comments are in place: If fusion is employed combined with field pullup, the

waste in not eliminating booleans is minimal: one, or two bytes, which are divided among two or three virtual nodes.

Nothing of course stops us from applying again boolean elimination to the internal nodes (Listing 7.1.2). If this is done (and combined with fusion), we achieve the following overhead savings: *(i)* 59% for `TreeSet` in HotSpot32, *(ii)* 55% for `TreeMap` in HotSpot64, and *(iii)* 61% for `TreeSet` in HotSpot64. (as explained, due to alignment no additional saving is introduced for `TreeMap` in HotSpot32.)

Second, we can employ fusion also for leaves into their parent, even if the other child is neither **`null`** nor a leaf. The fused class would then again use a single object header to represent three virtual nodes, and would again eliminate six internal tree pointers. The difficulty is however, that such fusion might interfere with fusion of the other child with its own children.

7.2 Full Field Consolidation

We finally note that an entire `TreeMap<K,V>` data structure can be encoded without using any small objects. This is done by defining six arrays, `K[] key`, `V value[]`, **`boolean`**`[] color`, **`int`**`[] left`, **`int`**`[] right`, and **`int`**`[] parent`. In this representation a node i is merely the `i`th location in all of these arrays, and pointers are replaced by array indices.

With full field consolidation, we save the headers of all small objects, and the alignment issue due to the byte sized `color` field is gone. A node thus consumes, assuming full occupation of the arrays, 21 bytes instead on 32, on HotSpot32. On HotSpot64, total size per node is 29 bytes instead of 64. On both cases the overhead is 13 bytes, and we achieve the following overhead savings: *(i)* 46% for `TreeMap` in HotSpot32, *(ii)* 54% for `TreeSet` in HotSpot32, *(iii)* 73% for `TreeMap` in HotSpot64, and *(iv)* 77% for `TreeSet` in HotSpot64.

We mention that implementation was relatively easy, and required minimal changes to the existing implementaion. Most of the changes involved replacing `p.left` like expressions with `left[p]` expressions. Moreover, this approch encourages a more sophisticated, but obviously more time consuming, arrays allocation scheme, where arrays type changes according to the current size of the collection. First **`byte`**`[]` arrays, then **`short`**`[]` arrays and

finally `int[]` arrays. The expected saving increases rapidly for small collections as summarized in Table 7.2.1. We remind the reader that we ignored the *instance-memory* (See 3.2), which decreases the saving for small collections. Nevertheless, even if we do add the arrays header cost, the expected addition is around two bytes per key for a collection of size 50, and would decrease rapidly as size increases.

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

Table 7.2.1: 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.

We can even squeeze four additional bytes in this representation by replacing the three arrays `int[] left`, `int[] right`, and `int[] parent`, by storing two independent linear combinations of these three integers, and using the fact that one of them is always pre-known (you always reach a tree node from some other node), to compute the two remaining indices. However, this implementation is not fully compatible with the existing one.

The main disadvantage of the full consolidation method is that memory management duties are shifted from the JVM back to the programmer. Damaging is also the fact that the arrays need to be kept longer with sufficient slack for future updates. A generous 50% slack for example, places field consolidation behind fusion.

An interesting mental exercise would be the combination of consolidation, fusion, and even null pointer elimination. This is possible by defining distinct array regions for each of the "node kinds", but it would raise all memory management issues back to the user level.

Chapter 8

Conclusions and Further Research

In this research we have shown that by using five relatively simple to employ memory reduction techniques we were able to dramatically reduce the memory overhead of some of the most common JRE collections. It is reasonable to argue that these collections are being used in every large-scale Java application including the JVM implementation itself. The introduced memory techniques do not require any JVM modifications and may be applied on any user-defined Java data structure by the programmer oneself. We have shown that wisely combining techniques together leads to greater memory savings and that these saving dramatically improve when used in 64-bit Java environments.

Unfortunately runtime performance was hard to predict due to many different possible usage scenarios and due to Java micro-benchmarking difficulties we encountered as summarized and discussed in details in [16].

This research raises two questions. First, it is important and interesting to understand better the domain of tiny collections, of say up to 16 entries, as their relative overhead is more significant. As reducing the overhead of these seems even 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.

Bibliography

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 7-13 2008. ACM Press.

[2] N. Alon, M. Dietzfelbinger, P. B. Miltersen, E. Petrank, and G. Tardos. Linear hash functions. *J. ACM*, 46, September 1999.

[3] Y. Arbitman, M. Naor, and G. Segev. Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. of the 51th IEEE Annual Symp. on Foundation of Comp. Sci. (FOCS'10)*, Las Vegas, Navada, Oct.23–26 2010. IEEE Computer Society Press.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[5] D. F. Bacon, S. J. Fink, and D. Grove. Space and time efficient implementation of the Java object model. In *Proc. of the 17th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'02)*, Seattle, Washington, Nov. 4–8 2002. ACM SIGPLAN Notices 37(11).

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[7] D. Caromel, J. Reynders, and M. Philippsen. Benchmarking Java against C and Fortran for scientific applications. In D. Thomas, editor, *Proc. of the 20th Euro. Conf. on OO Prog. (ECOOP'06)*, volume 4067 of *LNCS*, Nantes, France, July 3–7 2006. Springer.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 48–64, Vancouver, British Columbia, Canada, Oct.18-22 1998. ACM SIGPLAN Notices 33(10).

[9] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3), May/June 1998.

[10] L. F. Cranor and R. N. Wright. Influencing software usage. In *Proc. of the 10th Conference on Computers, Freedom and Privacy (CFP'00)*. ACM, 2000.

[11] B. Dufour, B. G. Ryder, and G. Sevitsky. 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'08)*, Atlanta, Georgia, Nov. 9–14 2008. ACM Press.

[12] N. Eckel and J. Gil. Empirical study of object-layout strategies and optimization techniques. In E. Bertino, editor, *Proc. of the 14th Euro. Conf. on OO Prog. (ECOOP'00)*, volume 1850 of *LNCS*, Sophia Antipolis and Cannes, France, June 12–16 2000. Springer.

[13] W. Feller. *An Introduction to Probability Theory and Its Applications*, volume I. Wiley, 1968.

[14] R. P. Gabriel and D. Bacon, editors. *Proc. of the 22nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'07)*, Montreal, Quebec, Canada, Oct.21-25 2007. ACM Press.

[15] J. Y. Gil and Y. Zibin. Efficient dynamic dispatching with type slicing. *ACM Trans. Prog. Lang. Syst.*, 30(1), 2007.

[16] Y. Gil, K. Lenz, and Y. Shimron. 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'11)*. ACM, 2011.

[17] C. H. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. M. W. Hwu. Compilers for improved Java performance. *Computer*, 30(6), June 1997.

[18] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in Java strings. In G. E. Harris, editor, *Proc. of the 23rd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'08)*, Nashville, Tennessee, Oct.19-23 2008. ACM.

[19] T. Kotzmann, C. Wimmer, H. Mossenbock, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Trans. Prog. Lang. Syst.*, 5(1), May 2008.

[20] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *Proc. of the 16th ACM SIGKDD Comference on Knowledge Discovery and Data Mining (KDD'10)*, Washington, DC, July25–28 2010. ACM Press.

[21] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1), 2010.

[22] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In Gabriel and Bacon [14].

[23] J. E. Moreira, S. P. Midkiff, and M. Gupta. A comparison of Java, C/C++, and FORTRAN for numerical computing. *Antennas and Propagation Magazine, IEEE*, 40(5):102–105, Oct 1998.

[24] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[25] K. Ogata, D. Mikurube, K. Kawachiya, and T. Onodera. A study of Java's non-Java memory. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proc. of the 25th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'10)*, Reno/Tahoe, Nevada, USA, Oct.17-21 2010. ACM.

[26] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *Proc. of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, California, Apr. 23-24 2001. USENIX C++ Technical Conf. Proc.

[27] S. P. Reiss. Visualizing the Java heap. In *Proc. of the 32nd Int. Conf. on Soft. Eng. (ICSE'10)*, Cape Town, South Africa, May2-8 2010. ACM.

[28] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'09)*, Dublin, Ireland, June 15-20 2009. ACM Press.

[29] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3^{rd} edition, 1997.

[30] K. Venstermans, L. Eeckhout, and K. D. Bosschere. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *TACO*, 4(3), 2007.

[31] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[32] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proc. of the 18^{th} ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (FSE'10)*, Santa Fe, New Mexico, Nov. 7–11 2010. ACM Press.

[33] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *Proc. of the 30^{th} Int. Conf. on Soft. Eng. (ICSE'08)*, Leipzig, Germany, May10-18 2008. ACM.

[34] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'10)*, Toronto, Canada, June 5-10 2010. ACM Press.

הקטנת חתימת הזכרון עבור אוספים בשפת ג'אווה

יובל שמרון

הקטנת חתימת הזכרון עבור אוספים בשפת ג'אווה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

יובל שמרון

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
חשוון ה'תשע"ב חיפה נובמבר 2011

המחקר נעשה בהנחיית פרופ' יוסי גיל בפקולטה למדעי המחשב.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תקציר

כחלק מההתמודדות עם בעיית ה"התנפחות`` של תוכניות מחשב (שהפכה לבעיה נפוצה ככל שנפח הזכרון (הראשי/המשני) של מחשבים גדל וגדל) ובפרט של מימושים של אוספים ושל מבני נתונים אחרים, אנו מציגים חמש טכניקות כלליות לצמצום חתימת הזכרון. טכניקות אלו יכולות להיות בשימוש כדי לצמצם את הזכרון של העצמים (אובייקטים) הבונים את מבני הנתונים.

הטכניקות הן: (א') העלמה של מצביעים לעצמים שערכם הנפוץ הוא `null` על-ידי טכניקה של הורשה ויצירת מחלקה יורשת שבה מוגדר לראשונה המצביע. בטכניקה זו אפשר לצמצם עד שמונה בתים בסביבה וירטואלית של 32 סיביות וגם בסביבה של 64 סיביות. (ב') העלמה של שדות בוליאנים שערכם הוא קבוע לרוב, על-ידי טכניקה של הורשה ויצירת שתי מחלקות יורשות המחזירות בעזרת מתודות וירטואליות את הערך המתאים של השדה הבולאני. גם בטכניקה זו אפשר להגיע לתוצאות צמצום כמו ב- (א'). (ג') איחוד ו"התכה`` של אובייקטים לכדי אובייקט יחיד. מצב זה אפשרי למשל ברשימה מקושרת ובה מספר קטן של אלמנטים כאשר כל ההצבעות לאלמנטים פנימיים ברשימה הן מאלמנטים אחרים ברשימה. טכניקה זו חוסכת מצביעים (ואולי גם מצביעים דו-כיווניים) וגם את הרישא (header) של האובייקטים שהעלמנו. (ד') הגדרה מקדימה של שדות בהיררכית ירושה. כלומר, במידה ומחלקה יורשת ממחלקה אחרת מגדירה שדה אשר גורם להגדלת הזיכרון ובמחלקה הנורשת נותר מקום לא מנוצל בגלל תופעת היישור של גדלים של אובייקטים בזיכרון ניתן להגדיר מבעוד מועד את השדה במחלקה הנורשת (ולא ייעשה בו שימוש). במצב זה לא נגדיר את השדה במחלקה היורשת ונוכל לחסוך בזכרון בגלל תופעת היישור של האובייקט היורש. (ה') הוצאה של שדות למערכים נפרדים. טכניקה זו מתאימה למשל למצב בו ישנו מערך של אובייקטים מטיפוס מסויים. יתכנו מצבים שונים המובילים לבזבוז של זכרון, ובהם מצב של בזבוז בגלל תופעת היישור ובזבוז בגלל הרישא של האובייקטים המוצבעים מהמערך ומבמצביעים עצמם. למשל, אם האובייקט מגדיר שדה בודד, ניתן להצביע ישירות לשדה זה ובכך לחסוך בשני מצביעים ורישא של אובייקט ואולי בבזבוז שנוצר מתופעת היישור. לעיתים גם אם מוגדרים מספר שדות במחלקה מסוימת ישתלם להגדיר מערך ישיר עבור כל אחד מן השדות.

עיקר המחקר מציג, תוך שימוש בטכניקות הללו, שתי שיטות אלטרנטיביות לקידוד המידע הנשמר על-ידי המחלקות הנפוצות בשפת ג'אווה והן `HashMap`, `HashSet`. בנוסף מוצג

מודל מתמטי לניתוח חתימת הזכרון של המימושים החדשים לעומת המימוש הקיים.

כיום, מימוש אלגוריתם הערבול במחלקות שלעיל נעשה על-ידי ערבול ב''שיטת השרשראות``. בשיטה זו אלמנטים שאמורים להיות מאוחסנים באותו תא בטבלה מאוכסנים ברשימה מקושרת פשוטה. המימוש הראשון שאנו מציגים מבוסס על שיטת ה''התכה`` של אלמנטים ברשימה מקושרת זו. על-ידי שימוש בטכניקה זו אנו מסוגלים לצמצם את תקורת הזכרון (כלומר את כל הזכרון שאינו מידע הכרחי) בטווחים שבין 20 אחוזים ועד 40 אחוזים במכונה וירטואלית ובה 32 סיביות, ובטווחים שבין 45 אחוזים לבין 65 אחוזים במכונה וירטואלית ובה 64 סיביות. יתרה מזאת מובטח כי טווחים אלו ישמרו ללא תלות בצורה בה המפתחות מפוזרים בטבלה. הטכניקה המבטיחה יותר, מבצעת שיפורים נוספים (תוך האטה מינימאלית בביצועים) ומציגה צמצום של תקורת הזכרון בטווחים שבין 30 אחוזים ועד 50 אחוזים במכונה ובה 32 סיביות, ובטווחים שבין 60 אחוזים ועד 70 אחוזים במכונת 64 סיביות. מימוש זה אינו מתאמץ לטפל במקרים נדירים בהם פיזור המפתחות גרוע שם נוצרות שרשראות ארוכות, ואולם ניתן להרחיבו לטפל במקרים אלו.

השאלה החשובה המתבקשת היא כיצד משפיעים שיפורים ניכרים אלו על זמני הריצה בפועל. באופן כללי לא ראינו שיפור/הרעה בצורה ניכרת או חד-משמעית כאשר הרצנו תוכניות כלליות שנועדו לבדוק ביצועים של מכונות וירטואליות של שפת ג'אווה ובהן נעשה שימוש מסוים במחלקות ששינינו. חלק מתוצאות המדידות הציגו מובהקות סטטיסטית לעיתים לטובת שיפור מינימאלי ולעיתים לטובת הרעה מינימאלית. ניסיון לביצוע בדיקה יותר קפדנית ומעמיקה ברמת המיקרו (Micro-benckmarking) לא הניב תוצאה חד-משמעית לגבי האצה או האטה של פעולות פרטניות במבנה הנתונים ולמעשה נכשל בגלל מגבלות שפת ג'אווה בחוסר העקביות של זמני הריצה של תוכניות ג'אווה ובפרט של פעולות קצרות מאוד. קשיים אלו מתוארים בהרחבה במאמר [16] אשר לקחתי חלק בכתיבתו.

אחת התוצאות המעניינות היא (וגם לגבי החלק הבא של המחקר) שכאשר מבצעים מימוש קפדני בעזרת הטכניקות שלנו, ובזכות ההבדלים שנרשמו באחוזי השיפור בתקורת הזיכרון בין המחלקות `HashMap`, `HashSet`, ישנה הצדקה לכך שיהיו שני מימושים שונים למחלקות אלו ולא מימוש אחד כפי שקיים היום, בו המחלקה `HashSet` עושה שימוש סמוי במחלקה `HashMap` ואינה ממשת בעצמה את אלגוריתם הערבול. יתרה מזאת, ישנה הצדקה למימושים שונים בסביבות של 32 סיביות ו- 64 סיביות.

בהמשך המחקר אנו מרחיבים את הטכניקות שהצגנו לעיל ומכילים אותם גם על המימושים הידועים של העצים האדומים-שחורים בשפת ג'אווה והם מתוארים במחלקות `TreeMap`, `TreeSet`. גם עבור מחלקות אלו אנו מתארים שני מימושים אפשריים, אך מאוד שונים זה מזה, המובילים לתוצאות משמעותיות ביותר בכל הקשור בצמצום תקורת הזכרון. המימוש הראשון עושה שימוש בטכניקות דומות לזה שעושה המימוש הראשון עבור טכניקות ערבול ומשיג חסכון של כ- 43 אחוזים עבור עץ-מיפוי (`TreeMap`) וכ- 59 אחוזים עבור עץ-אוסף (`TreeSet`). כל זאת נכון לסביבת מכונה וירטואלית ובה 32 סיביות. בסביבה ובה 64 סיביות החיסכון עולה לכ- 55 אחוזים וכ- 61 אחוזים עבור עץ-מיפוי ועץ-אוסף בהתאמה. מימוש זה משפיע לרעה בצורה ניכרת על הביצועים מכיוון שהוא מסובך מאוד למימוש בשפת ג'אווה אך

הוא עדיין נשאר פרקטי מבחינת זמני ריצה ובהחלט מציג חיסכון משמעותי. המימוש השני מרחיב טכניקה אחרת מקודמו ומציג תוצאות גבוהות יותר, אך מעלה מספר קשיים בנוגע לשימוש בו. הוא משיג חסכון של כ- 46 אחוזים עבור עץ-מיפוי וכ- 73 אחוזים עבור עץ-אוסף בסביבה ובה 32 סיביות. בסביבה ובה 64 סיביות החיסכון עולה לכ- 54 אחוזים וכ- 77 אחוזים עבור עץ-מיפוי ועץ-אוסף בהתאמה. יתרה מזאת אנו מציגים אפשרויות הרחבה לטכניקה האחרונה שמובילים לתוצאות חיסכון גבוהות יותר (תוך סיבוך המימוש) היכולות להגיע ליותר מ- 83 אחוזים בסביבת 32 סוביות וליותר מ- 92 אחוזים בסביבת 64 סיביות, וכל זאת בעבור אוספים המוגבלים יותר מבחינת יכולת הקיבולת.

מתוצאות המחקר בולטת הבעייתיות של המימושים הקיימים בכל האוספים שלעיל בשפת ג'אווה והשפעתם על צריכת הזכרון העודפת ללא סיבות מוצדקות והשפעתה התיאורתית על תוכניות גדולות. באופן כללי תוכניות ''כבדות'' בזכרון עלולות להשפיע לרעה על הביצועים שלהן ביחס לתוכניות ''קלות'' יותר, ועל הנכונות להשתמש בהן בתנאים מסוימים. לאורך כל המחקר הקפדנו כי האלטרנטיבות שאנו מציעים ישמרו על האלגוריתמים הקיימים והייצוג הסמנטי של האוספים כך שלמעשה ההחלפה של המימושים הקיימים היא אפשרית ללא ''שבירת'' קוד קיים המשתמש בהם.

נדרש מחקר נוסף על מנת לאמוד את ההשפעה של מימושים אלטרנטיביים לאוספים שנמצאים בשימוש נפוץ כל-כך ובפרט של אוספים קטנים שם יתכן שגם נגדיל את צריכת הזכרון ושם גם כנראה המימושים שלנו אינם מהירים יותר.