DRAFT

Fast and Lean Immutable Multi-Maps on the JVM based on Heterogeneous Hash-Array Mapped Tries

Michael J. Steindorfer
Centrum Wiskunde & Informatica, The Netherlands
Michael.Steindorfer@cwi.nl

Jurgen J. Vinju
Centrum Wiskunde & Informatica, The Netherlands
TU Eindhoven, The Netherlands
INRIA Lille, France
Jurgen.Vinju@cwi.nl

Abstract

An immutable multi-map is a many-to-many thread-friendly map data structure with expected fast insert and lookup operations. This data structure is used for applications processing graphs or many-to-many relations as applied in static analysis of object-oriented systems. When processing such big data sets the memory overhead of the data structure encoding itself is a memory usage bottleneck. Motivated by reuse and type-safety, libraries for Java, Scala and Clojure typically implement immutable multi-maps by nesting sets as the values with the keys of a trie map. Like this, based on our measurements the expected byte overhead for a sparse multi-map per stored entry adds up to around 65 B, which renders it unfeasible to compute with effectively on the JVM.

In this paper we propose a general framework for Hash-Array Mapped Tries on the JVM which can store type-heterogeneous keys and values: a Heterogeneous Hash-Array Mapped Trie (HHAMT). Among other applications, this allows for a highly efficient multi-map encoding by (a) not reserving space for empty value sets and (b) inlining the values of singleton sets while maintaining a (c) type-safe API.

We detail the necessary encoding and optimizations to mitigate the overhead of storing and retrieving heterogeneous data in a hash-trie. Furthermore, we evaluate HHAMT specifically for the application to multi-maps, comparing them to state-of-the-art encodings of multi-maps in Java, Scala and Clojure. We isolate key differences using microbenchmarks and validate the resulting conclusions on a real world case in static analysis. The new encoding brings the per key-value storage overhead down to 30 B: a 2 x improvement. With additional inlining of primitive values it reaches a 4 x improvement.

1. Introduction

This paper is about the challenges of optimizing immutable multi-maps on the Java Virtual Machine (JVM) and how they can be solved using a general method of coding heterogenous hash-array mapped tries. A multi-map is a data structure which acts as an associative array storing possibly multiple values with a specific key. Typically multi-maps are used to store graphs or many-to-many relations.

Many-to-many relations or graphs in general occur naturally in application areas such as static analysis of object-oriented software. In some applications it is the case that the initial raw data is many-to-one, and further processing or exploration incrementally leads to a many-to-many mapping for some of the entries. In other applications the distribution of sizes of the range sets in the raw data is highly skewed, such as when representing scale-free networks, like academic citations, the web, online social networks, and program dependence graphs. The number of values associated with a specific key is then practically always very low, yet there are possibly numerous exceptions to cater for nevertheless, where many values end up being associated with the same key. A key insight in the current paper is that we can exploit these highly common skewed distributions to save memory for the most frequent cases.

On the JVM relations are not natively language-supported; rather the standard libraries of Java, Scala and Clojure either provide implementations of multi-maps, or the map and set Application Program Interfaces (APIs) allow programmers to construct multi-maps easily in a type-safe manner (i.e., using sets as the values of a normal polymorphic map). The goal of this paper is to overcome the limitations of these existing implementations of multi-maps, improving drastically on the memory footprint without loss of storage, lookup and iteration efficiency. Typically state-of-the-art multi-maps come with a mode of 65 B overhead per stored key/value item, while the most compressed new encoding in this paper reaches an optimum of 30 B. In general the encoding has 2 x smaller footprints (modal) when storing reference objects, and 4 x smaller footprints when storing Java primitive values.

On the JVM, immutable collections are used mostly by functional/object-oriented programmers from the Scala and Clojure communities. However, since Java 8 the functional and streaming APIs [4] are becoming mature, making immutable collections become more relevant in the Java context.

Immutability for collections implies referential transparency (without giving up on sharing data) and it satisfies safety requirements for having co-variant sub-types [12]. Because of these properties, immutable collections are also safely shared in presence of concurrency.

Our point of departure is the Hash-Array Mapped Trie (HAMT) data structure [2], which has proven to be an efficient immutable alternative to array-based implementations. In contrast to arrays, HAMTs enable fine-grained memory layout optimizations [17]. There exists an optimized encoding [18] of HAMTs tailored the JVM, named Compressed Hash-Array Mapped Prefix-tree (CHAMP). The CHAMP data structure allows for time and memory efficient immutable maps and sets. To efficiently encode multi-maps we propose a generalisation of the CHAMP data structure to allow for heterogeneous data shapes. The new resulting data structure, called Heterogeneous Hash-Array Mapped Trie (HHAMT), unifies design elements from both HAMT and CHAMP. A HHAMT allows for a type-safe API in which keys and values can be represented using different types of data within the same map. This allows for all kinds of optimized data structures, but we focus on multi-maps in this paper as the key purpose. A basic dichotomous HHAMT multi-map is used to either store an inlined single value, or a full nested set data structure. We propose an efficient encoding of HHAMT to mitigate the incurred overhead.

1.1 Contributions and Roadmap

We address the design and evaluation of HHAMT as follows:

- Section 2 describes the foundations of HHAMT and identifies the main sources of overhead that need to be mitigated.
- Section 3 outlines scalable encoding of source code specializations (and their necessary runtime support) to yield memory savings between 2 x and 4 x.
- Section 4 compares HHAMT against CHAMP (baseline) to understand the cost of turning a (homogeneous) map into a (heterogeneous) multi-map.
- Section 5 compares a specialized HHAMT multi-map against idiomatic solutions from Clojure and Scala.
- Section 6 compares the memory footprint of a specialized HHAMT multi-map against state-of-the-art primitive collection libraries (Goldman Sachs, FastUtil, Trove, Mahout).
- Section 7 compares the performance of multi-maps in HHAMT, Clojure, and Scala on a realistic case.

Section 8 discusses related work and Section 9 enumerates further use cases for heterogeneity, before we conclude in Section 10.

2. Heterogeneous Hash-Trie Data Layout

A general trie [8, 9] is a lookup structure for finite strings that acts like a Deterministic Finite Automaton (DFA) without any loops: the transitions are the characters of the strings, the internal nodes encode prefix sharing, and the accept nodes may point to values associated with the strings. In a HAMT, the strings are the bits of the hash codes of the elements stored in the trie. A HAMT is memory efficient not only because prefixes are shared, but also because child nodes are only allocated if the prefixes of two or more elements overlap.

```
1  abstract class HamtCollection {
2   HamtNode root; int size;
3   // 1-bit + runtime checks (e.g., instanceof)
4   class HamtNode {
5    int bitmap;
6    Object[] contentArray;
7   }
8  }
9  abstract class ChampCollection {
10  ChampNode root; int size;
11
12  // 2-bits (distributed)
13  class ChampNode {
14   int datamap;
15   int nodemap;
16   Object[] contentArray;
17  }
18 }
19 abstract class HeterogeneousHamtCollection {
20   HeterogeneousHamtNode root; int size;
21
22  // n-bits (consecutive)
23  class HeterogeneousHamtNode {
24   BitVector bitmap = new BitVector(n * 32);
25   Object[] contentArray;
26  }
27 }
```

Listing 1. Skeletons of a various HAMTs.

The first class in Listing 1 (lines 1–8) depicts a typical encoding of a HAMT in Java. A single 32-bit integer bitmap is used to encode which of the 32 trie-branches —and correspondingly which slots in the untyped array— are used, together with a mapping function that calculates offsets in the array by counting bits in the bitmap. In general, a HAMT must be able to distinguish between three possible states for each trie-branch: absence of data, and otherwise distinguishing the data category (either payload, or a sub-node). Because a single bit cannot differentiate three different states, additional dynamic checks —such as **`instanceof`**— are used for discriminating the data category. Note that data payload and sub-nodes occur in arbitrary order in the array.

The second class in Listing 1 (lines 9–18) depicts the skeleton of the CHAMP encoding [18], which operates like a HAMT but uses an explicit encoding to eliminate dynamic **`instanceof`** checks. With two bitmaps CHAMP improves the mapping function to regroup the array slots into two separate homogeneously-typed sequences: a sequence of data payload, followed by a sequence of sub-node references. Because each homogeneous sequence uses its own bitmap, CHAMP kept the bitmap processing identical to HAMTs.

Summary. In a HAMT, each trie node contains an arbitrary mix of data elements and sub-nodes, therefore array slots require type checks individually. In contrast, CHAMP splits HAMT's mixed data sequence into two homogeneous sequences, enabling optimizations that were not possible before. A key to performance —when iterating over or batch-processing elements of homogeneous or heterogeneous data structures— is that individual elements do not need to be checked for its specific type [6]. This is also one of the reasons why the CHAMP performs better than the HAMT. In short: the homogeneous CHAMP data structure provides a good starting point for heterogeneous collections.

2.1 Generalizing Towards a Heterogeneous HAMT

The third class in Listing 1 (lines 19–27) illustrates the proposed HHAMT skeleton. HHAMT uses a multi-bit encoding like CHAMP but reverts to a sequential representation: one larger bitmap that stores a sequence of 32 n-bit tuples consecutively, instead of at maximum k individual bitmaps. k denotes the maximum number of supported heterogeneous types while n denotes the number of bits needed in our encoding.

For any k, a HHAMT requires $n = \lceil log_2(k+2) \rceil$ bits at minimum per trie-branch to encode all of its possible states. The two additional states are needed for encoding the absence of a trie branch, and encoding sub-trees in case of hash-prefix collisions. For the sake of clarity we mainly focus on the $k = 2$ case in the evaluation (Sections 4, 5, 6 and 7), where the required number of bits $n = 2$. This case covers the scenario of distinguishing between a singleton value, and an arbitrarily sized nested set for multi-map implementations. However in the current section we detail the general design and code for arbitrary k. Note that fixing k does influence efficiency trade-offs: experimental results for $k = 2$ do not generalize directly to other values of k.

2.2 HHAMT API

Although this is not a core contribution, since we model data structures beyond the power of Java's type system, we should detail how to circumvent it. Java does not support union types, and a polymorphic wrapper (such as Scala's `Either`) would introduce overhead. To solve this we can either write or generate specialized code for fixed combinations of types, or use Java's method type *polymorphism* and judicious use of class literals (a.k.a. type tokens like `Integer.class`).

For multi-maps, which are heterogeneous only internally, a simple generic API will suffice. For other applications, such as when the keys or values of a map are type heterogeneous or primitive values are inlined, code generation for the wrapping API is possible. If we use Java's method polymorpism (cf. *Effective Java*, Item 29 [5]) instead we may avoid code generation at a certain cost. We use type tokens and their `cast` method to encode type heterogeneity. Up to Java 8 it is not possible to bind primitive types to type variables though, and care must be taken to avoid dynamic type errors. Casts can be avoided using either-typed (temporary) wrappers or a typed callback interface. Examples can be found in the Appendix. Note that the internals of the HHAMT can always decide upon the type a value with 100 % certainty.

2.3 Bitmap Encoding and Indexing

The heterogeneous skeleton in Listing 1 (lines 19–27) does not exhibit an optimal encoding. We specialize the `BitVector` code for obtaining better memory performance. Assuming $k = 2$, we use a single **`long`** field as bitmap, for a larger k we would use several consecutive **`int`** or **`long`** fields.

The way we index into the trie node array (for lookup, insertion or deletion) is a key design element. This indexing is different between the original CHAMP encoding and the new HHAMT encoding because there are k-cases to distinguish.

Listing 2 shows how CHAMP's original per-node-bitmap indexing would work if generalized to multiple entry types. By default CHAMP already distinguishes between payload data and nested nodes with separate bitmaps. This baseline (naive) design for heterogeneous hash tries carries on similarly to distinguish more types of references. The masking function (lines 1–3) selects the prefix bits based on the node level in the tree (`shift = 5 * level`). The index function (line 4–6) requires a `bitpos` variable with a single non-zero bit, designating one of the 32 possible branches. It then maps from the `bitmap`/`bitpos` tuple to a sparse-array index by counting the non-zero bits in `bitmap` on the right of `bitpos`. On line 9 a method template for lookup, insertion, and deletion is shown. Because for each of the three data categories a separate bitmap is used the processing happens in a linear-scanning manner until the right category for a hash-prefix is matched, or the default case applies (line 31).

Although lines 12, 18, and 24 suggest the use of separate bitmaps for each distinct type, two bitmaps are sufficient to distinguish between three cases:

```
1 int xxxxMap = rawMap1 & rawMap2;
2 int dataMap = rawMap2 ^ xxxxMap;
3 int nodeMap = rawMap1 ^ xxxxMap;
```

The above listing depicts how to retrofit three logical bitmaps onto two physical bitmaps. The fields for `datamap` and `nodemap` are renamed to `rawMap1` and `rawMap2`. Subsequently, the data structure infers three logical views from the two raw bitmaps. We further will refer to this retrofitted heterogeneous variant as Heterogeneous Compressed Hash-Array Mapped Prefix-tree (HCHAMP).

Listing 3 illustrates operations on the bitmap in the generalized data structure that is specialized to $k = 2$. The mask function can be reused, and the `index` function is scaled to using a **`long`**. The new template method retrieves the 2-bit wide `pattern` (line 12) and translates it to an enum value to switch on. Instead of having to search linearly, as in Listing 2, we now jump directly to the relevant case handler. Using a fast **`switch`** is even more beneficial with an increasing number of heterogeneous types ($k > 2$), and while iterating which is when type dispatch will be hot.

```
static final int mask(int hash, int shift) {
  return (hash >>> shift) & 0b11111;
}
static final int index(int bitmap, int bitpos) {
  return Integer.bitCount(bitmap & (bitpos - 1));
}

// processing in (Heterogeneous) CHAMP
void processAtNode(int keyHash, int shift) {
  int mask = mask(keyHash, shift);
  int bitpos = bitpos(mask);

  int nodeMap = nodeMap();
  if ((nodeMap & bitpos) != 0) {
    // process node
    int index = index(nodeMap, bitpos);
    ...code for lookup, insert or delete ...
  } else {
    int dataMap = dataMap();
    if ((dataMap & bitpos) != 0) {
      // process payload category 1
      int index = index(dataMap, bitpos);
      ...code for lookup, insert or delete ...
    } else {
      int xxxxMap = xxxxMap();
      if ((xxxxMap & bitpos) != 0) {
        // process payload category X
        int index = index(xxxxMap, bitpos);
        ...code for lookup, insert or delete ...
      } else {
        // process empty slot
        ...code for lookup, insert or delete ...
    }
  }
}
```

Listing 2. Processing of multiple bitmaps with 1-bit entries.

```
static final int index(long bitmap, long bitpos) {
  return Long.bitCount(bitmap & (bitpos - 1));
}

// processing in a Heterogeneous HAMT
void processAtNode(int keyHash, int shift) {
  long bitmap = bitmap();

  int mask = mask(keyHash, shift) << 1;
  long bitpos = 1L << mask;

  int pattern = (int) ((bitmap >>> mask) & 0b11);
  Type type = toEnum(pattern);

  switch (type) {
  case EMPTY:
    ...code for lookup, insert or delete ...
    break;
  case NODE:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_1:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  case PAYLOAD_CATEGORY_2:
    int index = index(filter(bitmap, type), bitpos);
    ...code for lookup, insert or delete ...
    break;
  }
}
```

Listing 3. Processing of one bitmaps with 2-bit entries.

2.4 Optimizing Bit-Counting

Extra bitwise operations are in the overhead of HHAMT which we need to mitigate. We explain three techniques to do so.

Relative Indexing into a Single Data Category. The purpose of the `index` function in Listing 3 is to calculate the relative index of a data element within its data category. Given a `type` enum and a trie-branch descriptor (`bitpos`), the `index` function calculates how often the given `type` pattern occurs in the bitmap before the `bitpos` position.

The Java standard library contains bit count operations for the types **`int`** and **`long`** that count the number of bits set to 1. These functions do not support n-bit patterns with $n > 1$. However, we want to reuse the aforementioned functions, because on the widespread *X86/X86_64* architectures they map directly to hardware instructions. We introduce some bitmap pre-processing with filters to get to that point where we can use the native bit counters. Listing 4 illustrates how such a filter reduces a matching 2-bit wide pattern to a single bit set to 1, while resetting all other bits to 0.

Distribution of Heterogeneous Elements. While lookup, insertion, and deletion only require indexing into a single data category, on the other hand iteration and streaming require information about the types of all elements in a trie node: their frequency per node. Studies on homogeneous data structures [6] have shown avoiding checks on a per elements basis is indeed relevant for performance.

To also avoid such checks in HHAMT we introduce the use of histograms, on a per node basis, that are calculated in constant time (for a given branch factor). The computation is independent of the number of heterogenous types:

```
int[] histogram = new int[2^n];

for (int branch = 0; branch < 32; branch++) {
  histogram[(int) bitmap & mask]++;
  bitmap = bitmap >>> n;
}
```

```
1  static final long filter(long bitmap, Type type) {
2   long mask = 0x5555555555555555L;
3
4   long masked0 = mask & bitmap;
5   long masked1 = mask & (bitmap >> 1);
6
7   switch (type) {
8   case EMPTY:
9    return (masked0 ^ mask) & (masked1 ^ mask);
10  case NODE:
11   return masked0 & (masked1 ^ mask);
12  case PAYLOAD_CATEGORY_1:
13   return masked1 & (masked0 ^ mask);
14  case PAYLOAD_CATEGORY_2:
15   return masked0 & masked1;
16  }
17 }
```

Listing 4. Filtering of multi-bit patterns (for $k = 2$).

The former listing abstracts over the number of heterogeneous elements and executes in 32 iterations. `n` and `mask` are constants, where `mask` has the lowest n bits set to 1. In its generic form, the code profits from default compiler-level optimizations —such as scalar replacement [15] to avoid allocating the array on the heap, and loop unrolling.

We assigned the bit-pattern `EMPTY = 0` and `NODE` = 2^{n-1}, the various remaining heterogenous types are assigned consecutively in the middle. For iteration, streaming, or batch-processing data, histograms avoid expensive repetition of indexing individual categories: k bit-count operations, where each one requires applying a filter to the bitmap. For example, the total number of elements, regardless of their types, can be calculated with `32 - histogram[EMPTY] - histogram[NODE]`. The otherwise complex code for trie-node iteration reduces to looping through the two-dimensional histogram using two integer indices. The added benefit is that inlined values although stored out of order, will be iterated over in concert, avoiding spurious recursive traversal and its associated cache misses [18]. Finally, iteration can exit early when the running counter reaches the maximum branching factor of 32 to avoid iterating over empty positions in the tail. Note that for fixed k the code can be partially evaluated (i.e., by a code generator) to avoid the intermediate histogram completely.

Reversing the Bitmap Encoding: Extracting Index and Type. For enabling fast structural equality comparisons [18] maintaining a canonical form of the hash-trie is essential, also after the delete operation. For HHAMT and especially for HHAMT multi-maps this takes an extra effort: the deletion operation does know the index and type of the removed element, however it does not know the index and type of the remaining elements. Upon deletion, canonicalization triggers inlining of sub-tree structures with only a single remaining payload tuple. Efficiently recovering the index an type of the only remaining tuple is important for the overall efficiency of the deletion operation. We devised a recovery function for bitmaps with n-bit tuples, based on Java's standard library functions: `Long.numberOfTrailingZeros(bitmap)/n*n`. By first counting the number of trailing zeros, we approximate the region within the bitmap that contains bit-pattern information. We subsequently adjust the non-zero count to our n-bit pattern alignment with an integer division followed by an multiplication. As a result, we recovered the `mask` that allows retrieving the type of the remaining element (cf. Listing 3, lines 10–13).

Outlook. We have now discussed all techniques to mitigate Central Processing Unit (CPU) overhead caused by a more complex indexing. The remaining challenge is saving memory, which is discussed next.

3. Lean Specializations

Specialization for a fixed number of heterogeneous types will prove essential for both memory efficiency and CPU performance. In this section we take the perspective of the general k-heterogeneous HHAMT. The effect of these techniques will be evaluated in Sections 4, 5, 6 and 7 in different contexts.

For a HHAMT with k different types, there exist $arity_{nodes} \times \prod_{i=1}^{k} arity_i$ possible strongly-typed variants in theory, with the constraint that $arity_{nodes} + \sum_{i=1}^{k} arity_i <= 32$. We can reduce this complexity by grouping different heterogeneous types together into a section that is represented by their least upper bound type. Ultimately, we can group together all reference types and sub-nodes into one section, and all primitive types into another section [20], to achieve a quadratic upper bound that overcomes the dichotomy of reference and primitive types. Therefore, in the remainder of this section we will focus on the most common case of $k = 2$ that also satisfies our use case of multi-maps. Note that due to the bitmap encoding we always know the precise type of an object, using more general types for internal storage is solely used to reduce the total number of specializations.

There exist empirical evidence [17] for $k = 1$ that specializing up to arities of 8 or 12 balances impact on memory performance best with the necessary amount of generated code. However with heterogeneity $k > 1$ this may not hold, and to exploit inlining primitive types for saving more memory we should support specializing the full bandwidth up to 32.

We now present an improved approach for code generation that allows fully specialized collections (i.e., "array-less" data structures) with very low memory footprints. It aims to overcome the following issues that typically compromise performance of specialized code:

Additional Polymorphism: Turning a generic data type into a set of distinct specializations compromises trace-based inlining strategies of a Just-in-time (JIT) compiler. By introducing specializations, previous monomorphic call-

```
abstract class Set1 implements Set {
 final Object slot0;

 Set add(Object value) {
  if (slot0.equals(value)) {
   return this;
  } else {
   return new Set2(slot0, value);
  }
 }
}
```

Listing 5. Interlinking of specializations prohibits generic methods: `Set1` contains a static reference to `Set2`.

sites are turned into polymorphic call-sites. Thus a JIT compiler has to fallback to dynamic dispatch for method method calls that were previously resolved to direct calls.

Code Bloat: Substituting a dynamic structure with specialiations often demands the specialization of operations as well. In the case of hash-tries, we specialize for constant array sizes [17]: instead of referencing a heap structure, we inline the array-fields into a trie-node. Unfortunately the resulting memory saving come at a price: suddenly array operations (i.e., allocation, copy, get, set, length) must be specialized as well.

Interdependency of Specializations: In general, each specialized data type contains static references to other specializations that represent possible next states. Listing 5 exemplary lists the `add` method of set data structure specialized for one element that might return a set specialized for two elements. The switching between specialized representations, puts strain on the JIT compiler at run-time due to incremental class loading and the constant need to compile methods of specializations during a data structure builds up, further, it is one source of code bloat.

In the remainder of this section we detail our approach of specialization that remedies the aforementioned overheads. In our design, a specialization represents purely a heterogeneous trie-node, specialized for a certain content size. It contains pre-evaluated content stored in static fields and instance fields (storing the bitmap, and the inlined array content), however does not override methods.

3.1 Indexing and Selecting Specializations

We replace the use of arrays, which can be allocated using an arbitrary length parameter, with fields inlined in specialized classes. Commonly, for each specilization a unique constructor must be called (cf. Listing 5, specialization interlinking). Which constructor must be called depends on the current state of a trie node and the operation applied to the data structure.

To enable class selection at run-time, we introduce a global and static two-dimensional `Class[][] specializations` array, indexed by the number of primitive data fields (t) and the number of reference fields (u). This lookup table solves the interdependency problem of specialization: when adding a key-value tuple of reference type the next specialization can be be determined with `specializations[t][u + 2]`, or respectively with `specializations[t - 2][u]` when a tuple of primitive type is deleted. Once a specialization is selected, it can be initialized by invoking its default constructor: `Object instance = specialization[t][u].newInstance()`.

Since the array is often used and relatively small, we found it runs faster than distributing code over the specialized classes. This also allows for more generic code in base classes which is therefore used more often and more likely to be optimized by the JIT compiler.

3.2 Initializing Instances of Specialized Classes

For the generic representation that operates on arrays, we would use `System.arraycopy` initializing a new trie node, which is really fast. Now we want to try and approach similar efficiency for initializing the fields of our specialized classes.

Our solution is to introduce a `arraycopy`-like operation that is capable of copying consecutive fields between object instances: an `ArrayView` on an object layout is an abstraction which logically maps an arbitrary region within objects to an array. To ensure safety we check whether the JVM indeed maps the fields in a consecutive region at class loading time. Using a primitive `ArrayView.copy` we achieve similar performance to `System.arraycopy`. We measured the effect using a micro-experiment: the new primitive is about 20–30 % faster than field-by-field copying. Since eventually copying trie nodes is the primary bottleneck we may expect around similar speedups for insertion- and deletion-intensive code of HHAMT and less for lookup intensive code.

Listing 6 shows how we can model an array view of a range of fields within a heap object. Once we have obtained a reference to an `ArrayView`, we can invoke corresponding `(getFrom|setIn)HeapRegionArrayView` methods that either retrieve or a set a value of a `ArrayView`. To mimic `System.arraycopy` on an `ArrayView`, we use `sun.misc.Unsafe.copyMemory`. For our experiments, we extended the `copyMemory` function to support copying from/to objects while catering for card marking, i.e., signaling the Garbage Collector (GC) that references changed.

Relationship to `VarHandle` API of the Upcoming JDK 9. The Java Development Kit (JDK) 9 will introduce an API for uniformly referencing, accessing and modifying fields. Thus, independently of the realization of a variable —static field, instance field, or array— a handle to the field reference can be obtained. In earlier versions of Java, the granularity of references was restricted to objects; a `VarHandle` in contrast enables addressing fields or arrays (at a finer granularity) inside an object. The `VarHandle` API furthermore contains abstractions to view and process off-heap memory regions as arrays. However, it does not provide likewise abstractions for obtaining array views on on-heap regions.

```
class TrieNode_T4_U3 implements Node {
  long bitmap;

  int key0; int val0;
  int key1; int val1;

  Object slot0;
  Object slot1;
  Object slot2;

  static ArrayView getArrayView_T4() {
    return createHeapRegionArrayView(
        TrieNode_T4_U3.class, "key0", "val1");
  }

  static ArrayView getArrayView_U3() {
    return createHeapRegionArrayView(
        TrieNode_T4_U3.class, "slot0", "slot2");
  }
}
```

Listing 6. `ArrayView` on regions of specialized trie node.

The aforementioned `ArrayView` implementation we used provides a proof-of-concept implementation on how to extend the `VarHandle` API to support array views for on-heap regions.

3.3 Summary

In the context of collections, we eliminated issues that typically compromise the performance of specialized code. We will evaluate the effects of these techniques in Section 4.

4. Assessing the Cost of Multi-Maps

In this section we evaluate the performance characteristics of the various implementations of multi-maps on top of the HCHAMP, HHAMT, and specialized HHAMT encodings, comparing them against the basic homogeneous CHAMP map data structure as state-of-the-art baseline [18]. We are interested in isolating the effects that are incurred by adding the heterogeneity feature:

- HCHAMP is close to CHAMP (same logic, but derives three bitmap views from two physical bitmaps);
- HHAMT generalizes the heterogeneous bitmap encoding;
- and specialized HHAMT improves memory footprints by dynamically selecting statically known specializations.

In the case of multi-maps, heterogeneity lies in the internal distinction between $1:1$ and $1:n$ mappings.

Assumptions. We evaluate the pure overhead of operations on the data structures, without considering cost functions for `hashCode` and `equals` methods. This performance assessment should reveal the overhead of adding heterogeneity to CHAMP, the effect of the specialization approach and the effect of accessing the heterogeneous data elements.

Hypotheses. We expect HHAMT's runtime performance of lookup, deletion, and insertion to be similar comparable to CHAMP's runtime performance, but never better. Running times should not degrade below a certain threshold —we feel that $25\,\%$ for median values and $50\,\%$ for maximum values would about be acceptable as a trade-off— (Hypothesis 1).

Iteration over a multi-map is more complex than iterating over a map. Iteration (Key) has to distinguish between heterogeneous categories, whereas Iteration (Key) has to distinguish heterogeneous categories, Iteration (Entry) additionally has to flatten nested sets to obtain a tuple view on multi-maps. Consequently, we assume costs of about $25\,\%$ for median values and $50\,\%$ for maximum values as well (Hypothesis 2).

Based on related work in the domain of specializing HAMTs [17], we expect that specializing may introduce runtime overhead. However, we expect lower overhead (than the reported 20–40 % degradations for lookup) due to our mitigation strategies outlined in Section 3 (Hypothesis 3).

Furthermore, memory footprints of HCHAMP and HHAMT should in practice match CHAMP's footprints, because all variants use in total 64-bits for bitmaps (Hypothesis 4).

4.1 Experiment Setup

We use a machine with Apple OS X (10.11.3) and 16 GB RAM. It has an Intel Core i7-3720QM CPU, with 2.60 GHz, and an 6 MB Last-Level Cache (LLC). Frequency scaling was disabled. For testing, we used an OpenJDK (JDK 8u65) JVM configured with a fixed heap size of 8 GB. We measure the exact memory footprints of data structures with Google's memory-measurer library.[1] Running times of operations are measured with the Java Microbenchmarking Harness (JMH), a framework to overcome the pitfalls of microbenchmarking.[2] For all experiments we configured JMH to perform 20 measurement iterations of one second each, after a warmup period of 10 equally long iterations. For each iteration we report the median runtime, and measurement error as Median Absolute Deviation (MAD), a robust statistical measure of variability that is resilient to small numbers of outliers. Furthermore, we configured JMH to run the GC between measurement iterations to reduce a possible confounding effect of the GC on time measurements.

In our evaluation we use collections of sizes 2^x for $x \in [1, 23]$. Our selected size range was previously used to measure the performance of HAMTs [2, 18]. For every size, we fill the collections with numbers from a random number generator and measure the resulting memory footprints. Subsequently we perform the following operations and measure their running times:

Lookup, Insert and Delete: Each operation is measured with a sequence of 8 random parameters to exercise different trie paths. For Lookup and Delete we randomly selected from the elements that were present in the data

[1] `https://github.com/DimitrisAndreou/memory-measurer`

[2] `http://openjdk.java.net/projects/code-tools/jmh/`

structures.[3] For Insert we ensured that the random sequence of values was not yet present.

Lookup (Fail), Insert (Fail) and Delete (Fail): Measuring unsuccessful operations. The setup equals the aforementioned setting, however with the difference that we swap the sequences of present/not present parameters.

Iteration (Key): Iterating over the elements of a set or the keys of a map respectively.

Iteration (Entry): Iterating over a multi-map, flattening and yielding tuples of type `Map.Entry`.

We repeat the list of operations for each size with five different trees, starting from different seeds. This counters possible biases introduced by the accidental shape of the tries, and accidental bad locations in main memory. Evaluating HAMT data structures containing simply random integers accurately simulates any application for which the elements have good uniformly distributed hash codes. A worse-than-uniform distribution would —regardless of the HAMT-like implementation— overall reduce the memory overhead per element and increase the cost of updates (both due to clustering of elements). We consider a uniform distribution the most representative choice for our comparison.

4.2 Experiment Results

We first report the precision of the individual data points. For 99 % of the data points, the relative measurement error amounts to less than 1 % of the microbenchmark runtimes, with an overall range of 0–4.8 % and a median error of 0 %.

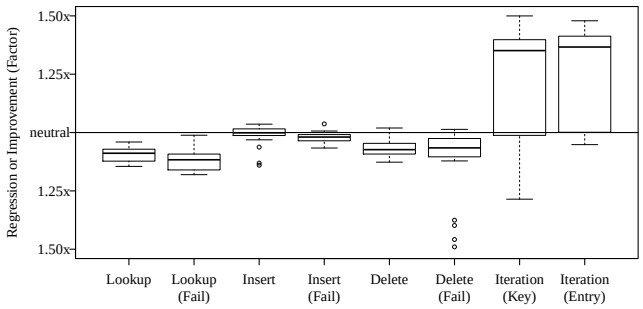
We summarize the data points of the runs with the five different trees with their medians. Then Figure 1a, and 1b report for each benchmark the ranges of runtime improvements or degradations. For brevity, the effects on memory footprints and of specialization are not contained in the boxplots, but are discussed in text. Each boxplot visualizes the measurements for the whole range of input size parameters. For improvements we report speedup factors above the neutral line ($\text{measurement}_{\text{CHAMP}} \,/\, \text{measurement}_{\text{HHAMT-Variant}}$), and degradations as slowdown factors below the neutral line , i.e., the inverse of the speedup equation. From this data we learn the following:

Confirmation of Hypothesis 1: The cost of converting a map to a multi-map stayed within the specified bounds for both HCHAMP and HHAMT.

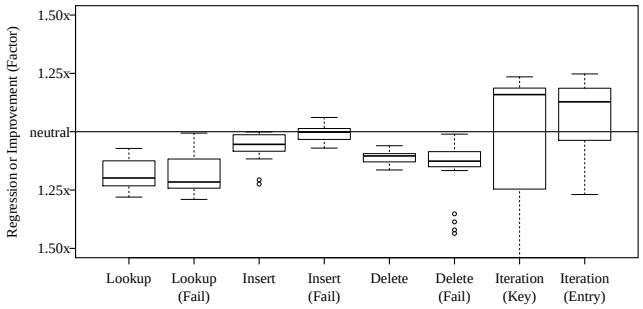
For HCHAMP, Lookup, Insert and Delete added a median slowdown of 9 %, 0 %, and 7 % respectively, and Lookup (Fail), Insert (Fail) and Delete (Fail) added 12 %, 2 % and 7 % respectively. With exception to single outliers produced Delete (Fail), the maximum slowdown are lower than 18 % at most.

For the generalized HHAMT, the costs for multi-maps over maps are higher. Lookup, Insert and Delete added a median slowdown of 20 %, 5 %, and 10 % respectively, and Lookup (Fail), Insert (Fail) and Delete (Fail) added 22 %, 0 % and 13 % respectively. With exception to single outliers produced Delete (Fail), the maximum slowdown are lower than 29 % at most.

[3] For < 8 elements, we duplicated the elements until we reached 8 samples.



(a) HCHAMP multi-map versus CHAMP map (baseline).



(b) HHAMT multi-map versus CHAMP map (baseline).

Figure 1. Visualizing the overhead of various multi-map implementations over a CHAMP map implementation.

(Partial) Confirmation of Hypothesis 2: Compared to our baseline, and counter to our intuition, HCHAMP improved Iteration (Key) by a median 35 % and Iteration (Entry) by 37 %. The more general HHAMT improved Iteration (Key) by a median 16 % and Iteration (Entry) by 13 %. However, according to Figure 1, the value spread appears large and the maximum bounds are violated for Iteration (Key).

(Partial) Confirmation of Hypothesis 3: On average, we observed an overhead of 3 % for Lookup and 6 % for Lookup (Fail) when comparing a specialized HHAMT against its regular HHAMT counterpart. These numbers confirm our intuition and are lower then the 20–40 % overhead reported by Steindorfer and Vinju [17]. The median costs for Insert (24 %) and Delete (31 %) however match their results. Concerning memory consumption, specializations improved memory consumption by at least 38 % for data structures with 32 or more entries.

Confirmation of Hypothesis 4: Memory footprints of HCHAMP and HHAMT (omitted in Figure 1) match exactly the footprint of CHAMP, when using multi-maps as maps.

Discussion. A more detailed investigation revealed that for Iteration (Key) measurements at sizes 2^1 and 2^5 showed significant deviation from the remaining measurements. These two measurements were not statistically identified as outliers due to the small sample size of 23 (sizes 2^x for $x \in [1, 23]$). When removing these two measurements, the upper bound of slowdowns is $6\,\%$ for HHAMT and $36\,\%$ for HCHAMP.

While not impacting lookup performance, specializing trades the runtime performance of insertion and deletion for gaining savings of approximately $1.4\,\text{x}$.[4] Because only operations that allocate new tree nodes are affected, we attribute slowdowns to the lookup table we introduced (adding two memory indirection). Nevertheless, specializing is of key importance when optimizations for primitive data types; we evaluate that effect separately in Section 6.

Summary. Despite its more complex and heterogeneous encoding, HHAMTs achieves excellent runtimes across all tested operations. Converting a map into a multi-map with the means of a heterogeneous encoding had usually less costs associated than we expected beforehand. Our specialization approach could successfully mitigate overhead for lookups while reducing memory footprints. However, using a lookup table for our specializations still impacts insertion and deletion, when compared to regular array allocations that do not require a lookup table.

5. Comparing Immutable Multi-Maps

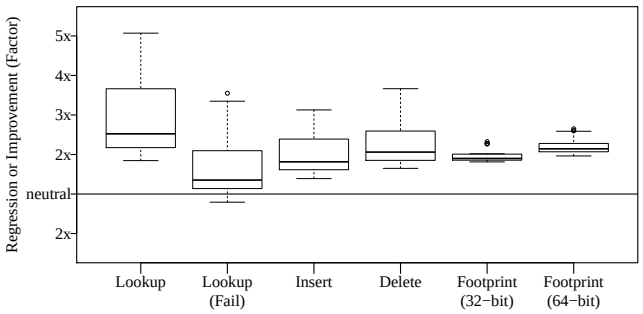
We further evaluate the performance characteristics of our specialized HHAMT multi-map against implementations from Clojure and Scala. Both languages do not provide native immutable multi-maps in their standard libraries, however suggest idiomatic solutions to transform maps with nested sets into multi-maps.

VanderHart [21, p. 100–103] proposes a solution for Clojure based on "protocols". Values are stored untyped as either a singleton, or a nested set. Consequently, the protocol extension handles the possible case distinctions —not found, singleton, or nested set— for lookup, insertion, and deletion.

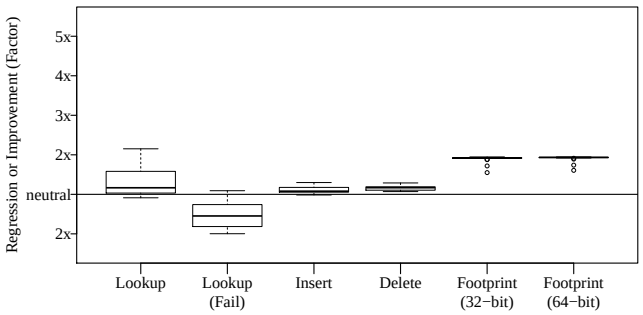
Scala programmers would idiomatically use a trait for hoisting a regular map to a multi-map. However, the Scala standard library only contains a trait for mutable maps; we therefore ported the standard library program logic of the trait to the immutable case, nesting typed sets into maps.

Hypotheses. We expect specialized HHAMT's runtime performance of lookup, deletion, and insertion to equal the competitors performance, because we tried hard to mitigate the incurred overhead, and the idiomatic solutions require some overhead as well. Runtimes should not degrade below a certain threshold —say $10\,\%$ for median values and $20\,\%$ for maximum values would just be acceptable— (Hypothesis 5). However, for negative lookups we expected that specialized HHAMT performs worse than Scala (Hypothesis 6). This hypothesis is based on related work [18] that explains the inherent differences between CHAMP and Scala when it comes to memoizing hash codes. Our hypothesis expects memory improvements by at least $50\,\%$ on average due to omitting nested collections for singletons (Hypothesis 7).

[4] Note that only the outer multi-map structure was specialized and not the nested sets. A further specialization of the nested sets would yield even more substantial memory savings.



(a) Specialized HHAMT multi-map versus Clojure's multi-map (baseline).



(b) Specialized HHAMT multi-map versus Scala's multi-map (baseline).

Figure 2. Performance comparison of a specialized HHAMT multi-map against implementations in Clojure and Scala.

5.1 Experiment Setup

Data generation is derived from the experimental setup outlined in Section 4.1. We keep the number of unique keys equal —2^x for $x \in [1, 23]$— but instead of using distinct data in each tuple, we now use $50\,\%$ of $1:1$ mappings, and $50\,\%$ of $1:2$ mappings. Fixing the maximal size of right-hand side of the mapping to 2 may seem artificial, but it allows us to precisely observe the singleton case, the case for introducing the wrapper and the overhead per additionally stored element. The effect of larger value sets on memory usage and time can be inferred from that without the need for additional experiments.

Insert: We call insertion in three bursts, each time with 8 random parameters to exercise different trie paths. Firstly we provoke full matches (key and value present), secondly partial matches (only key present), and thirdly no matches (neither key nor value present). Next to insertion of a new

key, this mixed workload also triggers promotions from singletons to full collections.

Delete: We call deletion in two bursts, each time with 8 random parameters. Provoking again, full matches and partial matches. Next to deletion of a key, this mixed workload also triggers demotions from full collections to singletons, and canonicalization where applicable.

Lookup: Similar to Delete we call lookup in two bursts to exercise full and partial matches.

Lookup (Fail): In a single burst with 8 random parameters we test negative lookups (neither key nor value present). We assume this test equivalent to Delete with no match.

5.2 Experiment Results

Figures 2a and 2b show the relative differences of specialized HHAMT multi-map compared to the implementations in Clojure and Scala. From the data we can evaluate our hypotheses:

Confirmation of Hypothesis 5: Runtimes unexpectedly improve over the competition. Lookup, Insert, and Delete perform similar to Scala (by a median 12 %, 9 %, and 16 % faster), and clearly better than Clojure (by a median speedup of 2.51 x, 1.75 x, and 2.05 x). Compared to Scala we observed individual data points that exhibited minimal slowdowns of less than 9 % at larger input sizes.

Confirmation of Hypothesis 6: HHAMT performs worse than Scala for negative lookups. Runtimes increased by a median 39 % and roughly doubled at maximum with a 106 % increase. In contrast, when compared to Clojure we do not see a negative impact.

Confirmation of Hypothesis 7: Memory footprints improve by a median factor of 1.92 x (32-bit) and 1.93 x (64-bit) over the implementation in Scala, and over in Clojure by a median factor of 1.9 x (32-bit) and 2.14 x (64-bit).

Discussion. We were surprised that the memory footprint consumptions of Clojure's and Scala's multi-map implementations are essentially equal. From related work [18] we knew the typical trade-offs of both libraries: Scala mainly optimizes for runtime performance, while Clojure optimizes for memory consumption. Code inspection revealed the cause of Scala's improved memory performance: their immutable hash-sets contains a specialization for singleton sets.

All three libraries libraries follow different paradigms for avoiding code duplication in their collection libraries. While Scala and Clojure use extension mechanisms (i.e., traits and protocols respectively), HHAMT avoids duplication by supporting internal heterogeneity.

Summary. With microbenchmarks we were able to measure the performance of individual operation, and further to measure the footprint of synthetically generated structures of different sizes. In this setting the heterogeneous design of specialized HHAMT proved to be better in general: improved runtimes of lookup, insertion, and deletion —with the notable exception of negative lookups when compared to Scala— and most importantly memory improvements of 1.9–2.14 x.

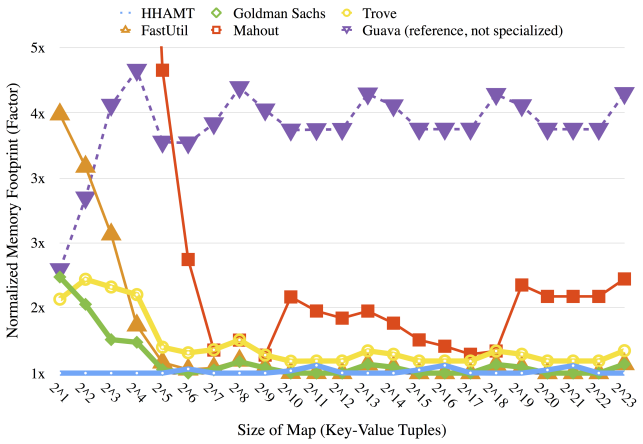


Figure 3. Comparing memory footprint of HHAMT specialized for `int` against state-of-the-art primitive `Map<int, int>` structures.

6. Case Study: Primitive Collection Footprint

A type-safe heterogeneous HAMT encoding shines most with bounded numerical data: it allows to exploit the difference between primitive (value-based) data types and reference types. More specifically, a `Multimap<int, int>` can leverage storing unboxed inlined singletons. Any non-heterogeneous immutable collection structure would have to store boxed integer objects instead, if not singleton sets of boxed integers. So, instead, as a fair point-of-reference we will compare to the state-of-the-art hand-optimized specialized immutable data structures for primitive values.

We are not aware of any comparable persistent or immutable primitive collection library which is optimized for primitive data types on the JVM. While there are many specialized primitive collection libraries for the JVM, only some contain (slower) copy-on-write immutable data structures implemented as facades over their mutable counterparts. With respect to primitive multi-maps, we did not find any implementation, neither mutable nor mutable.

So, we concentrate on comparing the memory footprint of `Map<int, int>`, implemented in specialized HHAMT (with $1 : 1$ mappings) compared to the most efficient primitive *mutable* collections we are aware of, namely: Goldman Sachs Collections, FastUtil, Trove, and Mahout. As a point of reference we also include Guava's `RegularImmutableMap` because it is a well-known library (but commonly known to be non-optimal in terms of memory consumption).

6.1 Experiment Results

Figure 3 illustrates observed memory footprints for maps for sizes 2^x for $x \in [1, 23]$. At each size, measurements are

normalized with respect to the minimum memory footprint (retained size of heap graph). Consequently, the minimum value 1 depicts the smallest data structure, whereas all other data points are displayed in their relative distance (factor of how much more memory they consume).

The results show that HHAMT consistently consumes the least amount of memory (median 1.00 x, range 1.00–1.10 x), followed by Goldman Sachs (median 1.04 x, range 1.00–2.18 x) and FastUtil (median 1.07 x, range 1.00–4.18 x). Trove exhibits constant results within a small bandwidth (median 1.23 x, range 1.15–2.15 x). In contrast to Trove's constant results, Mahout delivered surprisingly inconsistent results (median 1.94 x, range 1.22–29.64 x) —we capped the plot in Figure 3 to display maximal deviations up to 5 x. With overheads of 29.64 x, 25.08 x, 19.18 x, 11.24 x and 4.72 x for the data points 2^1–2^5, Mahout exceeds the footprints of our generic reference data structure from Guava (median 4.00 x, range 2.27–4.72 x).

Discussion. Compared to all other primitive collections, HHAMT excelled especially at small collections up to 32 elements. Given that in practice most collections are small [14] these improvements look promising. Primitive collections in general have the problem how to mark which slots are in use (there is no `null` equivalent in value types). Several encodings —e.g., sentinel values, or bitmaps— exist to circumvent this limitation. HHAMT performs well with respect to primitive collections, because HHAMT inherently encodes information about the presence and type of (primitive) values on a per node basis an therefore obsoletes special encodings for sentinel values. Further applications and benefits of heterogeneous data structures are discussed in Section 9.

Summary. In our measurements, HHAMT multi-maps that are specialized for `int` consume (with $1:1$ data) a median 4 x less memory than generic map data structures. HHAMT further achieves the same small footprints as class-leading primitive `Map<int, int>` data structures, while providing the additional functionality of allowing $1:n$ mappings.

7. Case Study: Static Program Analysis

The above experiments isolate important factors, but they do not show the support for the expected improvements on an algorithm "in the wild". To add this perspective, we selected computing control flow dominators using fixed point computation over sets [1]. The nodes in the graphs are complex recursive ASTs with arbitrarily complex (but linear) complexity for `hashCode` and `equals`. More importantly, the effect of the heterogenous encoding does depend on the accidental shape of the data, as it is initially produced from the raw control flow graphs, and as it is dynamically generated by the incremental progression of the algorithm.

Code. Although we do not claim the algorithm in this section to be representative of all applications of multi-maps, it is a basic implementation of a well known and fundamental algorithm in program analysis. It has been used before to evaluate the efficiency of hash-array mapped tried [18]. We implemented the following two equations directly on top of the multi-maps:

$$\mathrm{Dom}(n_0) = \{n_0\}$$
$$\mathrm{Dom}(n) = \left(\bigcap_{p \in \mathrm{preds(n)}} \mathrm{Dom}(p) \right) \cup \{n\}$$

Our code uses set union and intersection in a fixed-point loop: Dom and preds are implemented as multi-maps. The big intersection is not implemented directly, but staged by first producing a set of sets for the predecessors and intersecting the respective sets with each other.

Table 1. Runtimes of HHAMT for the CFG dominators experiment per CFG count, and statistics over preds relation about shape of data (unique keys, tuples, $1:1$ mappings).

#CFG	CHAMP	HHAMT	#Keys	#Tuples	% 1 : 1
4096	173 s	174 s	315 009	331 218	91 %
2048	84 s	85 s	162 418	170 635	91 %
1024	64 s	62 s	88 952	93 232	92 %
512	28 s	28 s	43 666	45 743	92 %
256	19 s	18 s	21 946	22 997	92 %
128	14 s	14 s	13 025	13 583	93 %

Hypotheses. On the one hand, since Dom is expected to be many-to-many with large value sets it should not generate any space savings but at least it should not degenerate the runtime performance either compared to CHAMP (Hypothesis 8). On the other hand we expect preds to be mostly one-to-one and we should get good benefit from the inlining of singletons (Hypothesis 9). Since CHAMP was reported to outperform existing state-of-the-art implementations in Scala and Clojure on the same case, there is no need to further include these [18].

Data. For our experiment, we used ± 5000 control flow graphs for all units of code (function, method and script) of Wordpress,[5] by using the PHP AiR framework [11]. Like before, we used JMH to measure CPU time. We ran the dominator calculations on a random selection of the aforementioned graphs. The set of selected graphs range between a size of from 128 to 4096 in exponential steps. Since smaller graphs occur much more frequently, we selected samples with exponentially increasing sizes from 128 to 4096. We furthermore measured the number of many-to-one and many-to-many entries in the preds relation.

[5] `https://wordpress.com`

Results. The results were obtained with a Linux machine running Fedora 20 (kernel 3.17). It featured 16 GB RAM and an Intel Core i7-2600 CPU with 3.40 GHz (8 MB LLC with 64-byte cache lines). Frequency scaling was disabled.

Table 1 shows the mean runtimes of the experiment for CHAMP and HHAMT. Both perform almost identically, with at most ± 2 s difference. Due to equal runtimes, HHAMT retains the same magnitude of speedups that CHAMP yielded over Clojure and Scala [18], from minimal 9.9 x to 28.1 x. We also observed that the shape of data in the preds relation contains a high number of $1:1$ mappings (median 92 %) and that the average ratio of unique keys to tuples is 1.05 x. In the case of Wordpress, the CFG algorithm turns out to profit over CHAMP in terms of memory savings from the heterogeneous opimizations for $1:1$ mappings. We conclude both Hypothesis 8 and 9 to be confirmed.

8. Related Work

Reducing the Memory Footprint of Collections is a goal of other people as well. Gil et al. [10] identified sources of memory inefficiencies in Java's mutable collections and proposed memory compaction techniques to counter them. They improved the memory efficiency of Java's `Hash{Map,Set}` and `Tree{Map,Set}` data structures by 20–77 %. We observed that even with added heterogeneity, HHAMT multi-maps achieve lower memory footprints than the class-leading primitive collection libraries, and in the generic case on average 4 x improvements over Guava's maps.

Steindorfer and Vinju [17] specialized internal trie nodes to gain memory savings of 55 % for maps and 78 % for sets at median while adding 20–40 % runtime overhead for lookup. Their approach minimized the amount of specializations to mitigate effects on code bloat and run-time performance. In contrast, we targeted the root causes of inefficiency one-by-one allowing full specialization at all arities.

Optimizing Collections in Dynamically-Typed Languages. Runtimes of dynamically typed languages often introduce a significant run-time and memory overhead [19] due to generic collection data structures that could at run-time hold a heterogeneous mix of data.

Bolz et al. [6] introduced a technique dubbed *storage strategies* that enables dynamic conversion of data representations. A set of interlinked strategies form a fine-grained type lattice that is based on known optimizations. Strategies mostly include support for collections of a homogeneous (primitive) type. An exemplary lattices for a `Set` data structure could be `EmptySetStrategy <-> (Integer|Float|...)SetStrategy <-> ObjectSetStrategy`. Resulting performance improvements mainly stem from object layouts that specialize for a homogeneous primitive types and corresponding optimized operations (e.g., direct value comparisons instead of calling `equals` methods).

Bolz [6] showed that with Python on average 10% of collections dehomogenize, mostly at small sizes. These results suggest that even in the absence of strict typing, collections are often used homogeneously. Heterogeneous data structures are orthogonal to homogeneous storage strategies. On one hand, heterogeneous data structures could diversify current strategy approaches, e.g., when homogeneous strategies are not applicable, or when many conversion occur. On the other hand, they have the potential to replace homogeneous strategies when flexibility in mixing data is required upfront. Furthermore, HHAMT optimizes internal heterogeneity that occurs in general purpose data structures such as multi-maps.

Specializations and Generics for Primitives Reducing Code-Bloat. Specializing for primitives can lead to a combinatorial explosion of variants amplifying code-bloat. Due to the object vs. primitive dichotomy, Java does not offer solutions countering a combinatorial explosion of code duplication when specializing for primitives. Java 10 or later will solve this issue by supporting generics of primitive types.[6]

Ureche et al. [20] presented a compiler-based specialization transformation technique called *miniboxing*. Miniboxing adds automatic specializations for primitive JVM data types to the Scala compiler while reducing the generated bytecode. Combinatorial code-bloat is tackled by specializing for the largest primitive type **`long`**, together with automatic coercion for smaller-sized primitives. While not always memory-optimal due to always utilizing **`long`** variables, miniboxing is a practical approach to tackle combinatorial code explosion.

HHAMT's contribution is orthogonal to both previously discussed techniques, because it generalizes the encoding of heterogeneous data stored together in a collection. HHAMT's specializations currently do duplicate code for different (primitive) type combinations. Using primitive generics in later versions of Java —or miniboxing in Scala— could bring this down to a single generic specialization per trie node arity.

(Partial) Escape Analysis. Escape analysis enables compilers to improve the run-time performance of programs: it determines whether an object is accessible outside its allocating method or thread. Subsequently this information is used to apply optimizations such as stack allocation (in contrast to heap allocation), scalar replacements, lock elision, or region-based memory management [16]. Current JVMs use partial escape analysis [15], which is a control-flow sensitive and practical variant tailored toward JIT compilers.

Our scalable encoding of specializing is a memory layout optimization for value-type based data types: trie nodes are specialized for arrays of constant sizes that do not escape. We use code generation to conceptually apply object inlining [22] of these statically sized (non-escaping) arrays into the memory layout of their corresponding trie nodes. Memory layout sensitive inlining as we perform could be applied in Virtual Machine (VM) based on information obtained from escape analysis. We hope that future compilers and language runtimes are capable of doing so out-of-the-box.

[6] `http://openjdk.java.net/projects/valhalla/`

9. Further Applications of HHAMT

We extrapolate some client applications which would benefit from HHAMT.

Libraries or Languages Supporting Data or Code Analysis on the JVM would benefit from more efficient in-memory multi-maps. Typical examples are frameworks such as KNIME [3] for general purpose data analysis or Rascal for program analysis [13], and MoDisCo [7] for software reengineering and reverse engineering, especially when their algorithms require frequent lookup and thus will benefit from an efficiently indexed relation such as a multi-map.

Unifying Primitive and Generic Collections. Looking at specialized collections for primitives from the programming language designer's perspective, they are a necessary evil implied by the dichotomy between objects and primitive values. Primitive values give programmers access to low level and memory-efficient data representations, but the impact of having them leaks through in the type systems and the design of standard libraries of programming languages supporting them. The current paper describes a heterogeneous framework that can be used for implementing data structure which allow storing either primitive data values or their boxed counterparts next to each other, while the client code remains practically oblivious. For statically-typed languages this implies we can have a generically typed library for both primitive and object values. For dynamically-typed languages it implies a much lower overhead for the builtin dictionaries.

Big Integers for Big Data. Most programming languages feature a library for representing arbitrary-sized integers. We use these to avoid overflow, especially in the context of scientific computing applications. The drawback of using these libraries for data science is that large homogeneous collections immediately blow up, even if the big numbers are exceptional. We want to use smaller Fixed-Width Numerics (FIXNUMs) were possible, and Big Object-Represented Numerics (BIGNUMs) only when necessary.

This application is where HHAMT could potentially have a rather big impact. Sets and maps filled with mostly inlined FIXNUM's and an occasional BIGNUM without having to a priori allocate space for BIGNUMs, and without having to migrate at run-time. Even if the entire collection accidentally ends up filled with BIGNUMs, HHAMT still yields more memory efficient representations than common array-based hash-maps.

Cleaning Raw Data in a Type-Safe Manner. The HHAMT data structure enables efficient storage and retrieval of objects of incomparable types without memory overhead (no need to wrap the objects) and without dynamic type checks. In Java there exist no "union" types like in C, but using HHAMT we can approach this in the context of collections. A typical use case would be reading in raw data from Comma-Separated Values (CSV) files (or spreadsheets) in Java where the data is not cleansed and some cells contain integers while the other contain decimal numbers or even empty cells, depending on the original manual and unvalidated input of the user. A CSV parser could output a HHAMT, inferring the most accurate value for each cell from the used notation, and allowing for further processing the data downstream in a manner both type-safe and efficient.

In general, homogeneous collections storing numeric data struggle with representing empty cells. Sentinel values (e.g., integer constant -1) are a viable solution if and only if the data does not use the data type's full value range. Cases where the data range is used exhaustively require additional auxiliary data structure (e.g., an array of booleans) to encode if a value is initialized. In contrast to homogeneous collections, HHAMTs by design supports mixing sentinel values of a different type (e.g., `static final EMPTY_CELL = new Object()`) with the full value range of primitives.

10. Conclusion

We proposed HHAMT, a new design for hash-array mapped tries which allows storage, retrieval and iteration over maps which store heterogeneously typed data. In particular we motivate this data structure by applying it to efficiently implement multi-maps, and it also shines when used to cater for inlining unboxed primitive values.

The evaluation compared to the state-of-the-art: comparing to other hash-trie data structures with and without the many-to-many feature, comparing against state-of-the-art encodings of multi-maps in Scala and Clojure and comparing to hand-optimized maps for primitive values. Even when compared unfairly to implementations which do not feature heterogeneity, HHAMT compares well. We safe a lot of memory (2–4 x) at relatively low costs in runtime overhead.

We hope multi-maps based on these results will be available in the future in the standard libraries for collections on the JVM, since that would make the JVM even more attractive for safely computing with large immutable datasets.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] P. Bagwell. Ideal Hash Trees. Technical report, Ecole polytechnique fédérale de Lausanne, Oct. 2001.

[3] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel. Knime - the konstanz information miner: Version 2.0 and beyond. *SIGKDD Explor. Newsl.*, 11(1):26–31, Nov. 2009. ISSN 1931-0145. .

[4] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams a la carte: Extensible Pipelines with Object Algebras. In *ECOOP '15: Proceedings of the 29th European conference on Object-Oriented Programming*. Schloss Dagstuhl, 2015.

[5] J. Bloch. *Effective Java, Second Edition*. Boston, MA, 2008.

[6] C. F. Bolz, L. Diekmann, L. Tratt, C. F. Bolz, and L. Diekmann. *Storage strategies for collections in dynamically typed languages*, volume 48. ACM, Nov. 2013.

[7] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0116-9. .

[8] R. De La Briandais. File Searching Using Variable Length Keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298, New York, New York, USA, 1959. ACM Press.

[9] E. Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[10] J. Gil and Y. Shimron. Smaller Footprint for Java Collections. In *ECOOP '12: Proceedings of the 26th European conference on Object-Oriented Programming*. Springer-Verlag, 2012.

[11] M. Hills and P. Klint. PHP AiR: Analyzing PHP systems with Rascal. In *Proceedings of IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE, 2014.

[12] A. Igarashi and M. Viroli. On Variance-Based Subtyping for Parametric Types. In *ECOOP '02: Proceedings of the 16th European conference on Object-Oriented Programming*. Springer-Verlag, 2002.

[13] P. Klint, T. van der Storm, and J. Vinju. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009.

[14] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM, 2007.

[15] L. Stadler. *Partial Escape Analysis and Scalar Replacement for Java*. PhD thesis, Johannes Kepler University Linz, May 2014.

[16] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and efficient hybrid memory management for java. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 81–92, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3589-8. .

[17] M. J. Steindorfer and J. J. Vinju. Code specialization for memory efficient hash tries (short paper). In *Proceedings of Generative Programming Concepts & Experiences (GPCE)*. ACM, Sept. 2014.

[18] M. J. Steindorfer and J. J. Vinju. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015. ACM, 2015.

[19] L. Tratt. Dynamically Typed Languages. In *Advances in Computers*, pages 149–184. Elsevier, 2009.

[20] V. Ureche, C. Talau, and M. Odersky. *Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations*, volume 48 of *improving the speed to code size tradeoff in parametric polymorphism translations*. ACM, Nov. 2013.

[21] L. VanderHart and R. Neufeld. *Clojure Cookbook: Recipes for Functional Programming*. O'Reilly Media, 2014. ISBN 9781449366414. URL `https://books.google.nl/books?id=xX38AgAAQBAJ`.

[22] C. Wimmer. *Automatic Object Inlining in a Java Virtual Machine*. PhD thesis, Johannes Kepler University Linz, February 2008.

```
interface HeterogeneousMap {
  // pull-based dispatch on type
  <K, V> TypedObject<?> put    (Class<K> keyType, K key, Class<V> valueType, V value);
  <K, V> TypedObject<?> remove (Class<K> keyType, K key);
  <K, V> TypedObject<?> get    (Class<K> keyType, K key);

  // push-based dispatch on type
  <K, V> void put    (Class<K> keyType, K key, Class<V> valueType, V value, CallbackMap callbacks);
  <K, V> void remove (Class<K> keyType, K key, Class<V> valueType, V value, CallbackMap callbacks);
  <K, V> void get    (Class<K> keyType, K key, Class<V> valueType, V value, CallbackMap callbacks);
}

interface TypedObject<T> {
  Class<T> getType();
  T get();
}

interface CallbackMap {
  <E> Consumer<E> put (Class<E> elementType, Consumer<E> consumer);
  <E> Consumer<E> get (Class<E> elementType);
}
```

Figure 4. Generic HHAMT interface, based on *Item 29: Consider typesafe heterogeneous containers* of *Effective Java* [5].

```
public void heterogeneousInterfaceTest() {
  put(String.class, "abc", int.class, 5);      // accepted by guard condition
  put(String.class, "abc", Integer.class, 5);  // accepted by guard condition

  put(String.class, "abc", long.class, 5L);    // rejected by guard condition
  put(String.class, "abc", Long.class, 5L);    // rejected by guard condition
}

static <T, U> void put(Class<T> keyType, T keyInstance, Class<U> valueType, U valueInstance) {
  switch(keyType.getName()) {
    case "java.lang.String":
      switch(valueType.getName()) {
        case "int":
          put((String) keyType.cast(keyInstance), (int) valueInstance);
          return;
        case "java.lang.Integer":
          put((String) keyType.cast(keyInstance), (Integer) valueInstance);
          return;
      }
  }

  System.out.println("Unsupported Type");
}

static void put(String keyInstance, Integer valueInstance) {
  System.out.println("put(String keyInstance, Integer valueInstance)");
}

static void put(String keyInstance, int valueInstance) {
  System.out.println("put(String keyInstance, int valueInstance)");
}
```

Figure 5. The method `heterogeneousInterfaceTest` illustrates a possible way to map a generalized HHAMT interface to specialized functions with type guards (cf. **`switch`** statement).