Optimizing Hash-Array Mapped Tries for Fast and Lean Immutable Collections

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

The data structures under-pinning collection API (e.g. lists, sets, maps) in the standard libraries of programming languages are used intensively in many applications.

The standard libraries of recent Java Virtual Machine languages, such as Clojure or Scala, contain scalable and well-performing immutable collection data structures that are implemented as Hash-Array Mapped Tries (HAMTs). HAMTs already feature efficient lookup, insert, and delete operations, however due to their tree-based nature their memory footprints and the runtime performance of iteration and equality checking lag behind array-based counterparts. This particularly prohibits their application in programs which process larger data sets.

In this paper, we propose changes to the HAMT design that increase the overall performance of immutable sets and maps. The resulting general purpose design increases cache locality and features a canonical representation. It outperforms Scala's and Clojure's data structure implementations in terms of memory footprint and runtime efficiency of iteration (1.3–6.7 x) and equality checking (3–25.4 x).

Categories and Subject Descriptors E.2 [*DATA STORAGE REPRESENTATIONS*]: Hash-table representations

Keywords Hash trie, persistent data structure, immutability, performance, cache locality, Java Virtual Machine.

1. Introduction

In this paper we reduce memory overhead and runtime performance overhead from the implementations of immutable collections on the Java Virtual Machine (JVM). Collections under-pin many (if not most) applications in general purpose programming languages as well as domain specific languages that are compiled to the JVM. Optimizing collections implies optimizing many applications.

Immutable collections are a specific area most relevant to functional/object-oriented programming such as practiced by Scala[1] and Clojure[2] programmers. With the advance of functional language constructs in Java 8 and functional APIs such as the stream processing API [6], immutable collections become more relevant to Java as well. Immutability for collections has a number of benefits: it implies referential transparency without giving up on sharing data [17]; it satisfies safety requirements for having co-variant sub-types [17]; it allows to safely share data in presence of concurrency.

The prime candidate data structure for efficient immutable collections is the Hash-Array Mapped Trie (HAMT) by Bagwell [3]. This data structure design was already ported from C++ to the JVM platform and is used especially in the Scala and Clojure communities. However, these ports do not perform optimally, because the performance characteristics of the JVM are substantially different from the performance characteristics of C/C++ runtimes. The unique properties of the JVM have not sufficiently been exploited yet. The following properties hinder efficiency of data structures that are ported directly from C/C++ to the JVM:

- The JVM currently does not allow custom memory layouts. A HAMT forms a prefix tree and therefore consists of linked nodes. Each HAMT node is represented as an object that is arbitrarily positioned in memory, resulting in an increased number of cache misses.
- Arrays are objects too. HAMT nodes use arrays to compactly store references to sub-trees. Hence, arrays introduce here yet another level of memory indirections.

[1] `http://scala-lang.org`

[2] `http://clojure.org`

In contrast, C/C++ allows custom memory layouts that are tailored to the shape of the data, without the need of memory indirections. With possibly better data locality than on the JVM, C/C++ runtimes directly place the content of statically sized arrays into an object's or struct's memory region.

Fine-tuning data structures for cache locality usually improves their runtime performance [19]. However, HAMTs inherently feature many memory indirections due to their tree-based nature, notably when compared to array-based data structures such as hashtables. Therefore HAMTs presents an optimization challenge on the JVM.

Our goal is to optimize HAMT-based data structures such that they become a strong competitor of their optimized array-based counterparts in terms of speed and memory footprints.

1.1 Contributions

Our contributions can be summarized as follows:

- We introduce a new generic HAMT encoding, named Compressed Hash-Array Mapped Prefix-tree (CHAMP). CHAMP maintains the lookup, insertion, and deletion runtime performance of a HAMT, while reducing memory footprints and significantly improving iteration and equality checks.
- In relation to CHAMP, we discuss the design and engineering trade-offs of two wide-spread HAMT implementations on the JVM that are part of Clojure's and Scala's standard libraries.
- In our evaluation, we compare the memory footprints and runtime performance of CHAMP against immutable sets and maps from the aforementioned two libraries.

Speedups of CHAMP range between 9.9 x and 28.1 x, for an example program analysis algorithm that uses classical set and relational calculus in a fixed-point loop. Using micro-benchmarks we show that CHAMP reduces the memory footprint of maps by $64\,\%$, and of sets by a median of $52\,\%$ compared to Scala. Compared to Clojure, CHAMP achieves a median memory footprint reduction of $15\,\%$ for maps, and of $31\,\%$ for sets. Compared to both, iteration speeds up 1.3–6.7 x, and equality checking speeds up 3–25.4 x (median).

1.2 Roadmap

The paper is structured as follows:

- Section 2 discusses background and opportunities for optimizing current HAMT implementations on the JVM.
- Section 3 describes the foundations of CHAMP: the differently ordered data layout of the tree nodes, and compression via an additional bitmap.
- Section 4 introduces an efficient algorithm to keep a HAMT in a compact and canonical form upon deletion, and how to use this in the implementation of equality checking.
- Section 5 discusses relevant design and implementation trade-offs that are related to our core contributions and necessary to discuss the evaluation results later.
- Section 6 compares CHAMP against Clojure's and Scala's data structures in terms of memory footprint and runtime efficiency with the help of microbenchmarks.
- Section 7 compares the performance of all three data structure libraries on a realistic case.
- Section 8 discusses threats to validity and notable differences between CHAMP and Clojure's and Scala's HAMTs.
- Section 9 discusses related work, before we conclude in Section 10.

All source code of data structures and benchmarks discussed in this paper is available online.[3]

2. Background

Although sets and maps need no motivation, immutable collections are not as commonplace in the object-oriented domain yet. We enumerate several applications here to convince the reader of the interest in optimizing them.

A number of (semi-)formal mechanisms for analyzing sets and relations, based on Codd's relational calculus or Tarski's relational algebra have or can have immutable collections under-the-hood. Example Java-based systems in this category are JRelCal [25], JGraLab [12] and Rascal [18]. Implementations of logic programming formalisms like, Datalog [15], can also benefit from optimizations of these data-structures [26]. Both categories of formalisms are used actively in static analysis of object-oriented programs. Another application domain is the representation of graphs and models in model-driven engineering.

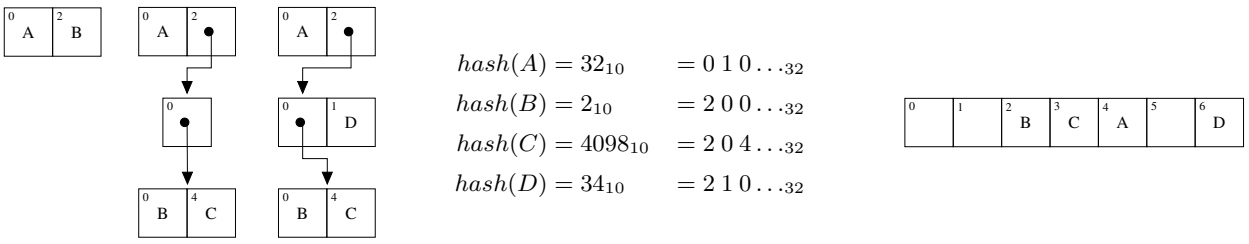
Making immutable collections efficiently available in object-oriented languages transfers the benefits of the aforementioned formal languages (Prolog, Datalog, Relational Calculus) to mainstream programming: immutability enables equational reasoning.

2.1 HAMTs Compared to Array-based Hashtables

A general trie [10, 13] 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. Depending on the branching factor we may use different sizes of chunks from the hash code for indexing into the (sparse) array of child nodes.

Figures 1a, 1b, and 1c graphically illustrate the structure of a small HAMT set with branching factor 32 after step-by-step inserting the objects A, B, C, and D. HAMTs encode the hash codes of the elements in their tree structure (cf. Figure 1d). The prefix tree structure grows lazily upon insertion until the new element can be distinguished unambiguously from all other elements by its hash code prefix. The index numbers

[3] `http://michael.steindorfer.name/papers/oopsla15-artifact`



(a) A and B (b) C (c) D (d) hash codes in base 32 (e) equivalent array-based hash set

Figure 1. Inserting three integers into a HAMT-based set (1a, 1b, and 1c), on basis of their hashes (1d). Figure 1e shows an equivalent and collision-free array-based hash set, with prime number table size 7 and load factor of 75 %.

in the left top corner of each node refer to the positions of elements in an imaginary sparse array. This array is actually implemented as a 32-bit bitmap and a completely filled array with length equal to the node's arity. To change a HAMT set into a map, a common method is to double the size of the array and store references to each value next to each key.

Figure 1e illustrates the same data stored in a more commonly known data structure, an array-based hashtable with table size 7 and load factor of 75 %. The buckets assigned to elements are calculated by $\text{hashcode} \mod 7$. Comparing these two figures we highlight the following inherent drawbacks of HAMTs against array-based hashtables:

Memory overhead: Each internal trie node adds an overhead over a direct array-based encoding, so finding a small representation for internal nodes is crucial. On the other hand, HAMTs do not need expensive table resizing and do not waste (much) space on `null` references.

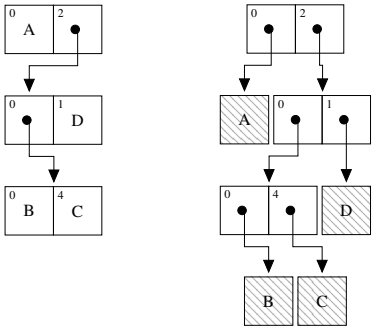
Degeneration on delete: Any delete operations can cause a HAMT to deviate from its most compact representation, leading to superfluous internal nodes harming cache locality, adding indirections for lookup, insertion and deletion, and increasing memory size. Delete on most hashtable implementations is a lot less influential.

Non-locality slows down iteration: When iterating over all elements, a hashtable benefits from locality by linearly scanning through a continuous array. A HAMT, in comparison, must perform a depth-first in-order traversal over a complex object graph (going up and down), which increases the chance of cache misses.

Equality is punished twice: While iterating over the one HAMT and looking up elements in the other, the non-locality and a possibly degenerate structure make equality checking an expensive operation.

2.2 Mutable and Immutable Update Semantics

HAMTs are suitable to implement data structures with mutable and immutable update semantics. The two variants differ in how and when nodes have to be reallocated. Mutable HAMTs reallocate a node if and only if the node's arity changes [3]. Otherwise, values or sub-node references are updated in-place without reallocation. In contrast to mutable HAMTs, immutable HAMTs perform path-copying on updates [22, 27]: the edited node and all its parent nodes are reallocated. The resulting new root node satisfies the immutability property by resembling the updated path-copied branch and, for the remainder, references to unmodified branches.



(a) Internal (b) Leaf

Figure 2. HAMT-based sets with values in internal nodes versus values at the leaves only.

2.3 Memory Layouts and Hash Code Memoization

Figure 2 illustrates the two principle choices for a HAMT's memory layout: storing values next to sub-nodes directly in *internal* nodes, as opposed to storing them at the *leaf* level.

The former approach originates from Bagwell [3], it increases locality and saves space over the latter. Because a HAMT encodes the prefixes of hash codes implicitly in its tree structure, it can avoid storing full 32-bit hash codes. As a result, this design yields a very low memory footprint at the potential cost of increased runtimes of update operations.

The latter approach stores elements in leaf nodes, separated from inner prefix tree nodes. While leaf nodes increase the data structure's memory footprint, they enable storage of additional information along the elements. Scala for example memoizes the hash codes of elements inside the leafs. Memoized hash codes consequently enable fast failing on negative lookups by first comparing hash codes, and avoid recalculation of hash codes upon prefix expansion.

```
abstract class HamtCollection {
  HamtNode root; int size;

  class HamtNode {
    int bitmap;
    Object[] contentArray;
  }
}
```

Listing 1. Skeleton of a HAMT in Java with internal values.

CHAMP builds upon the internal nodes design and primary optimizes for smaller footprints and cache locality. Nevertheless, to reconcile both designs in our evaluation section, we will also analyze a CHAMP variant that supports hash code memoization in Section 5.2.

3. Increasing Locality

Listing 1 shows a Java source code skeleton that is the basis for implementing HAMT collections with the internal nodes design. Traditionally [3], a single 32-bit integer bitmap is used to encode 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. For set data structures, we can use Java's `instanceof` operator to distinguish if an array slot holds either an element, or a sub-node reference. This encoding orders values by their hash codes—the bitmap compression solely eliminates empty slots.

In the following, we first devise a new compact data layout for internal trie nodes which enables faster iteration and equality checking. The causes of the performance increase are compactness and locality, both leading to better cache performance, and avoiding the use of the `instanceof` operation.

3.1 Partitioning Values and Sub-Nodes

Maps with internal values (as opposed to sets) require additional storage room for the values. To cater for this the dense array can be allocated at twice the normal size such that next to each key a reference to a value may be stored. Each index is then multiplied by 2 to skip over the extra slots.

Figure 3a exemplifies a HAMT-based map with internal values as found in Clojure. Accidentally, the root node has two child nodes, a few empty slots with `null` references and a key/value pair in the middle. Due to the fixed tuple length of two, an empty array slot is wasted per sub-node reference. Note that the C/C++ HAMT implementation of Bagwell [5] also uses fixed length tuples of size two for maps, however C/C++ offers better reuse of the empty slots, e.g., with union types to store other data instead.

HAMT Design Results in Poor Iteration Performance. One aspect that is directly influenced by the order of array elements is iteration performance. The order of child nodes and values depends entirely on the data structure's content; sub-nodes and internal values may alternate arbitrarily. An in-order traversal for iteration will switch between the trie nodes a few times, because values and internal nodes alternate positions in the array, causing a bad cache performance. For iteration, the HAMT design requires to go through each node at most $m + n$ times, where n equals the HAMT's total number of sub-tree references and m equals the total number of references to internal data entries.

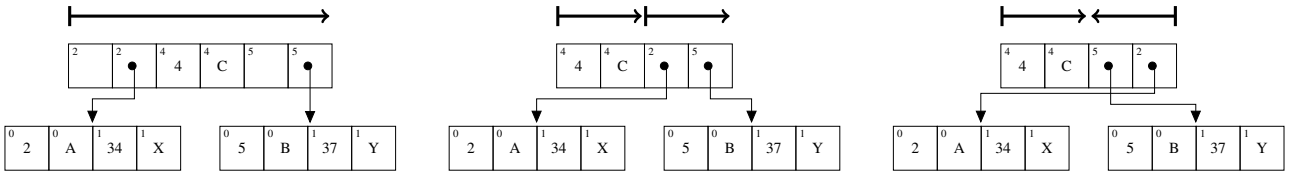
CHAMP Design Improves Iteration Performance due to a Cache-Oblivious Reordering with Compression. To increase data locality during iteration —and further to remove the need for empty array slots— we propose to split the single untyped array conceptually into two typed arrays (cf. Figure 3b): one array holding sub-node references, and another one holding values. This split effectively reorders the elements, while the ordering within each group remains. The element order is irrelevant anyway in already unordered set and map data structures. We remove the need for using `instanceof`, by introducing an extra bitmap that makes the separation between sub-nodes and internal values explicit. For iteration, the proposed CHAMP design reduces the complexity from $\mathcal{O}(m + n)$ to $\mathcal{O}(n)$. A full traversal in CHAMP requires exactly n node visits, because it can yield all internal values before descending for each sub-node exactly once.

Mitigating Memory Overhead. The CHAMP design conceptually requires two arrays and two accompanying bitmaps. A naive CHAMP implementation would introduce significant overhead compared to a HAMT. Figure 3b shows how we mitigate the incurred overhead by sharing one array for the two compressed array sequences. The second arrow indicates the need for the second bitmap. Two bitmaps are necessary to compute indices for either kind of value. We propose to use one 32-bit bitmap, called `datamap`, to store if a branch is either absent or a value reference, and another 32-bit bitmap, called `nodemap`, to store if a branch is either absent or a sub-node reference. Especially for maps we can spare the extra bitmap because we are saving an empty array cell for every sub-node due to our explicit encoding.

3.2 Mitigating Runtime Overhead

We have increased data locality at the cost of a more expensive index calculation that requires an extra bitmap and additional bit-level operations. This directly influences the runtime performance of lookup, insertion, and deletion. Each of these operations now requires a case distinction with, in worst case, two separate lookups in the two distinct bitmaps, to decide in which group an element is present. Because we compactly store both groups in a single array, we need to perform offset-based indexing when accessing the second group of sub-node references. Both the offset and the index calculations require more bit-level operations on the `datamap` and `nodemap`, as explained below.

Listing 2 illustrates Java snippets of how the indexing is usually implemented. Lines 1–3 shows the masking function that selects the prefix bits based on the node level in the



(a) Current HAMT ordering with 1-bit state. (b) Conceptual CHAMP ordering with 2-bit state. (c) Final CHAMP ordering with 2-bit state.

Figure 3. HAMT-based map implementations with values in internal nodes (various variants). The index numbers in the top left corners denote the logical indices for each key/value entry and not their physical indices. Figure 3a specifically shows Clojure's HAMT implementation that indicates sub-nodes by leaving the array slot for the key empty.

tree (`shift == 5 * level`). The index function (lines 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`.

Lines 8–18 illustrate indexing into a traditional HAMT that requires only a single call to `index`. For sets (`WIDTH = 1`), the content array does not have empty cells with `null`, for maps (`WIDTH = 2`) it follows the convention that an empty key slot indicates that the value slot is a sub-node reference.

Lines 20–34 show our proposal that requires different methods for accessing keys and sub-nodes. Accessing keys works equally to current HAMTs, however we call the `index` function with the `datamap`. For accessing sub-nodes we first calculate an `offset` (with `datamap`) before calling the `index` function with `nodemap`.

One option to remove the overhead of the offset calculation is to cache its value in a `byte` field. However, the extra `byte` pushes the memory layout of an internal node right over the JVM's 8-byte alignment edge, which seriously influences the memory footprint. Instead we remove the need for the additional field as follows. In our optimized and final CHAMP encoding, as displayed in Figure 3c, we reverse the storage order of the nodes in the back of the array. We can then perform `contentArray.length - 1 - index` instead of the previous `offset + index` calculation. Since the `length` field is there anyway in the array we pay no extra cost in memory. CHAMP's optimized index calculation code that mitigates overhead is displayed in Listing 2, lines 36–42.

3.3 Summary

We have increased locality by reshuffling the references in a trie node, at the cost of more bit-level arithmetic. Otherwise the lookup, insert, and delete operations are unchanged. For iteration, the proposed CHAMP design reduces complexity from $\mathcal{O}(m+n)$ to $\mathcal{O}(n)$. For maps we avoid empty slots in the arrays and thus save memory. In the evaluation section we will show that the net result of our design is satisfying.

4. Canonical Tries

Another way to increase locality and to further save memory is to keep a HAMT in a compact canonical representation, even after deleting elements. For example, in Figure 1b removing object C from the deepest node would yield a perfectly valid HAMT. However, in the optimal case, deletion would restore the state of Figure 1a, resulting in a smaller footprint and less dereferencing upon lookup, insertion, and deletion.

Clojure's HAMT implementations do not compact on delete at all, whereas Scala's implementations do. In the remainder of this section we contribute a formalization (based on predicates and an invariant) that details how a HAMT with inline values can efficiently be kept in a compact canonical form when deleting elements. Bagwell's original version of insert is enough to keep the tree canonical for that operation. All other operations having an effect on the shape of the trie nodes can be expressed using insertion and deletion.

4.1 Contract for Canonicalization

We formalize the canonical form of internal trie nodes by a strict invariant for trie nodes. The reasons are that canonicalization depends on the children of trie nodes to be in canonical form already and the many operations on HAMTs are somewhat complex. An explicit invariant helps in implementing the canonical form correctly and in optimizing the code. We need two basic properties to express the invariant:

Arity (local): The arity of a node designates the number of outgoing edges. In CHAMP the `arity` equals the sum of `nodeArity` and `payloadArity`, counting bits set to 1 in `nodemap` and `datamap` respectively.

Branch Size (non-local): The branch size equals the total number of elements that are transitively reachable from a node. Later we will find an approximation for branch size which can be computed locally.

We assume both properties are available as methods on the objects of the internal nodes. The following class invariant asserts canonically minimal trie nodes:

CHAMP invariant:

$$\texttt{branchSize} \geq 2 * \texttt{nodeArity} + \texttt{payloadArity}$$

The invariant states that sub-trees with arity less than 2 are not allowed. This implies that single elements should always be inlined and singleton paths to sub-tries should be collapsed. The invariant holds for all nodes on all levels.

```
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));
}

// HAMT indexing with 1-bit state
Object getKeyOrNode(K key, int hash, int shift) {
 int bitpos = 1 << mask(hash, shift);

 int index = WIDTH * index(this.bitmap, bitpos);
 if (contentArray[index] != null) {
  return contentArray[index];
 } else {
  contentArray[index + 1];
 }
}

// Proposed CHAMP indexing with 2-bit state
K getKey(K key, int hash, int shift) {
 int bitpos = 1 << mask(hash, shift);

 int index = WIDTH * index(this.datamap, bitpos);
 return (K) contentArray[index];
}

Node getNode(K key, int hash, int shift) {
 int bitpos = 1 << mask(hash, shift);

 int offset = WIDTH *
   Integer.bitCount(this.datamap);
 int index = offset + index(this.nodemap, bitpos);
 return (Node) contentArray[index];
}

// Optimized CHAMP indexing into sub-nodes
Node getNode(K key, int hash, int shift) {
 int bitpos = 1 << mask(hash, shift);

 int index = contentArray.length - 1 -
   index(this.nodemap, bitpos);
 return (Node) contentArray[index];
}
```

Listing 2. Index calculations for the various designs.

4.2 Deletion Algorithm

Deletion is a recursive operation on the tree. To satisfy the invariant, delete on a trie node structure should be the exact inverse operation of insert.

Listing 3 contains a pseudocode description of the delete operation. For a given key the search starts at the root node. If the node contains the search key locally, the operation removes the data tuple from the node and returns an updated node. Otherwise, if the node contains a sub-tree for a given hash-prefix, the operation will descend recursively.

```
delete(node: Node, key: Object): (Boolean, Node) {
 if (key in datamap) {
  if (node.arity == 1)
   return (true, EMPTY_NODE)
  else
   return (true, node without key)
 }

 if (∃subNode for which key is in nodemap) {
  (isModified, resultNode) = delete(subNode, key)

  if (isModified == false) // short-circuit
   return (false, node)

  if (node.arity == 1)
   if (resultNode.branchSize == 1) // propagate
    return (true, resultNode)
   else
    return (true, node updated with resultNode)
  else if (resultNode.branchSize == 1) // inline
   return (true, (node without subNode) with key)
  else
   return (true, node updated with resultNode)
 }

 return (false, node) // key not found
}
```

Listing 3. Pattern match logic for deletion in pseudocode.

If the hash prefix is in datamap (line 2) and the value stored matches the key we are looking for then we can remove a value right here. In case a CHAMP instance consists of a single root node with a singe element, an EMPTY_NODE constant is returned, otherwise we return a copy of the node without the current element. We may temporarily generate a singleton value node (which is allowed by the invariant since it does not have a parent yet) but later in line 21 on the way back from the recursion this singleton will be collapsed. A singleton can only fall through, when delete collapses a multi-node tree of two elements to a tree with a single root node with a single element.

Note that both operations in line 21 (removal of subNode and inlining of key) need to be executed atomically to ensure the invariant. We provide a copying primitive (named copyAndMigrateFromNodeToInline) in our implementation that performs both modifications with a single array-copy.

If the hash prefix is in the nodemap (line 9) then delete is called recursively. The compaction to satisfy the invariant happens on the way back from the recursion, dispatching on the values of arity and branch size of the current node and the received new node. If the current node's size is 1 then we may pass the new node to our own parent if it only contains values (line 17). If the received node has a bigger size, then we make a copy of the current singleton node with the new child at the right position (line 19). The final case is when

```
byte sizePredicate() {
  if (this.nodeArity() == 0)
    switch (this.payloadArity()) {
      case 0: return SIZE_EMPTY;
      case 1: return SIZE_ONE;
      default:return SIZE_MORE_THAN_ONE;
    }
  else return SIZE_MORE_THAN_ONE;
}
```

Listing 4. `sizePredicate` method used for compaction.

the new child node wraps a value, generated by line 6, and we can inline it here right now. Line 23 implements the case where no compaction or inlining takes place at all.

Deletion Implementation Details. The object-oriented implementation of the delete algorithm involves careful consideration. It should avoid re-computations of hash codes, avoid allocation of temporary objects and it should maintain the class invariant. Also, computing the predicates used in the algorithm may become a bottleneck, especially computing the branch size of a trie node is problematic since this operation is linear in the size of the sub-tree. The following design elements are necessary to make the delete operation efficient:

- Passing a single state object through the recursive calls of the delete method, like a reference parameter. The recursive calls record whether a modification was made.
- Inspecting the state of a received node using the `arity` and `branchSize` properties, dispatching on their values. Using these properties we avoid use of `instanceof`.
- Abstracting `branchSize` into an over-approximation which can be computed without recursion or caching. For the deletion algorithm (cf. Listing 3) we need to know if a sub-tree has no elements, exactly one element, or more. Thus we can substitute calls to `branchSize` with calls to the `sizePredicate` method (cf. Listing 4) that returns a `byte` representation of the aforementioned three states.

4.3 Structural Equality Checking

Tree compaction on delete lays the groundwork for faster (structural) equality checking. In an ideal world without hash collisions we would short-circuit recursive equality checks without further ado: if two nodes that are reachable by the same prefix have different bitmaps it is guaranteed that their contents differ. Together with short-circuiting on equal references (then the sub-tries are guaranteed equal), the short-circuiting on unequal bitmaps makes equality checking a sub-linear operation in practice.[4] Only when the heap graphs of two equal HAMTs are disjunct, then a full linear traversal is necessary to assert equality.

Still, hash collisions do occur in practice. In a HAMT, a collision is detected when all hash code bits are consumed and two elements cannot be differentiated. Similar to a hashtable, a chained bucket can then be created at the bottom of the tree to hold both elements (or more). To be able to see why short-circuiting on unequal bitmaps is still possible, even though the structure of the buckets depends on dynamic insertion and deletion order, consider that we short-circuit only on hash codes prefixes being unequal and not on the actual values. This means that if the Java hash code contract —unequal hash codes imply unequal values— has been implemented correctly the short-circuiting is safe. Listings 5 and 6 show the source code of `equals` methods (regular node and hash collisions node) of a CHAMP-based set implementation.

Canonical representations enable equality checks to benefit from the persistent nature of immutable HAMTs. Like lookup, insert, and delete already did, equality checks can exploit short-circuiting due to incremental updates and sharing of intermediate nodes between different trie instances.

4.4 Summary

Compacting trie nodes on delete and only lazily expanding them on insert makes sure they are always in a canonical and compact state. The algorithm satisfies an invariant for trie nodes and needs to be implemented with care to mitigate the overhead. We save memory and gain locality, but we increase CPU overhead with more complex bitmap compaction. The evaluation in Section 6 analyses the true impact of the proposed trade-offs.

5. Further Factors of Efficiency

Preliminary to the evaluation we discuss two other factors of efficiency that are orthogonal to CHAMP's design, but are key design decisions that are influential in the evaluation.

5.1 Implementing Fast Iterators

To support Java's `Iterable<T>` and `Iterator<T>` interfaces, our code is layered as in Listing 1. The outer (collection) layer provides key, value and entry iterators and a node iterator for the entire trie. The inner (trie node) layer provides separate iterators for the internal values and for the child nodes of each internal node. Thus, the internal iterators essentially reflect the logical partitioning.

We implemented the node iterator using a stack interface. Since we statically know the maximal trie depth, the stack can be implemented in a pre-allocated cache-friendly array.

The idea of using pre-allocated stack iterators is not novel. Scala's HAMT implementations are already leveraging such iterators. Our own implementation is more elaborate to achieve the worst case iteration complexity reduction from $\mathcal{O}(m+n)$ to $\mathcal{O}(n)$ as discussed in earlier in Section 3.1.

[4] Note that an alternative implementation of equals —e.g., such as implemented in Clojure— is to iterate over one tree while looking up each key in the other until the first key which is not found. This operation is especially expensive on large HAMTs as it performs in $\mathcal{O}(n\ log_{32}(n))$.

```
boolean equals(Object other) {
  if (other == this) return true;
  if (other == null) return false;
  if (getClass() != other.getClass())
    return false;

  ChampNode<?> that = (ChampNode<?>) other;

  if (datamap != that.datamap)
    return false;
  if (nodemap != that.nodemap)
    return false;

  if (!Arrays.equals(nodes, that.nodes))
    return false;

  return true;
}
```

Listing 5. `equals` method of a regular CHAMP node.

```
boolean equals(Object other) {
  if (other == this) return true;
  if (other == null) return false;
  if (getClass() != other.getClass())
    return false;

  HashCollisionNode<?> that =
    (HashCollisionNode<?>) other;

  if (hash != that.hash)
    return false;

  for (K key : keys)
    if (!that.contains(key))
      return false;

  return true;
}
```

Listing 6. `equals` method of a set's hash collision node.

5.2 Memoization and Hash Codes

In the following we discuss design and performance implications of memoization of hash codes on two different levels: on element basis, and on collection level. For the latter case we discuss implications of incrementally updating them.

Memoizing Collection Hash Codes. A key design element of CHAMP is to use the outer wrapper objects to cache collection hash codes and to incrementally update these hash codes as elements are added or removed. This requires insertion-order independent and reversible hash code computations. Due to the JVM's 8-byte memory alignment, adding the hash code field to CHAMP does not increase its memory footprint.[5]

In contrast, memoization of collection hash codes is not an option for Scala, because in their class hierarchy every node is also a collection. Adding another field to each node would increase the memory footprint of inner nodes by 8 bytes.

Memoizing Element Hash Codes. While Scala's HAMT implementations memoize the full 32-bit hash codes of keys in leaf nodes, neither Bagwell's HAMT design nor Clojure's implementations consider memoization. Adding memoization is trading a higher memory footprint against improved worst case runtime performance of lookup, insertion, and deletion.

Adding Memoization of Element Hash Codes to CHAMP. We strive for a flexible design that can be used with or without memoization. To add memoization to CHAMP, we use a technique called *field consolidation* [14]. Instead of storing cached hash codes in a large number of leaf nodes as Scala does, we consolidate the hashes of all elements of a node, to store them in a single integer array per node.

With MEMCHAMP we will refer throughout the text to the variant of CHAMP that adds memoized element hash codes, but drops incremental collection hash codes.

5.2.1 Performance Implications of Memoization

Table 1 summarizes the worst case number of `hashCode` and `equals` calls per lookup or update operation. It shows the cross product of all combinations from hash-set and hash-map data types paired with feature options of memoizing and incrementally updating the collection hash code, and memoizing the hash codes of the keys.

As in general with hash-based data structures, each operation needs to calculate the hash code of the key. Non-memoized HAMTs call in worst case `hashCode` once more with Insert $_{\neg\text{Contained}}$: if a prefix collision occurs the `hashCode` of the already stored element must be recalculated to extend the prefix tree. Otherwise, only incremental map operations cause additional `hashCode` calls, because suddenly the hash codes of the values are eagerly evaluated. Furthermore, equality is checked at most once for keys. Memoization avoids some `equals` calls due to fast-failing on equal hashes. Like Clojure and Scala, also other JVM collection libraries differ in their design choices for hash code memoization. E.g., Google's Guava[6] collections do not memoize key hash codes, whereas Java Development Kit (JDK) collections do.

We conclude, based on Table 1, that for sets incrementally calculating collection hash codes comes essentially for free: the number of `hashCode` and `equals` calls stay the same. Maps in contrast pay for the eager and incremental evaluation of value hash codes, as compared to lazy evaluation. We suggest considering incremental collection hash codes for maps, if nesting of maps into other hash-based collections is a frequent pattern in a program, library or language.

[5] This is valid for JVM instances with less than 32GB heaps with the *CompressedOops* option enabled (default). `https://wikis.oracle.com/display/HotSpotInternals/CompressedOops`.

[6] `https://github.com/google/guava`

Table 1. Worst case number of invocations of `hashCode`/`equals` per HAMT operation. The numbers exclude full hash collisions, but assume distinct hash codes with matching prefixes. For each operations we distinguish between the succeeding and the failing case. The table is split by the following HAMT features: memoizing and incrementally updating the collection hash code, and memoizing the hash codes of the keys.

Operation	`hashCode` / `equals` Calls per Data Structure and Feature Set							
	HAMT Map				HAMT Set			
	¬Incremental		Incremental		¬Incremental		Incremental	
	¬Memo	Memo	¬Memo	Memo	¬Memo	Memo	¬Memo	Memo
Lookup (¬Contained)	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0
Lookup (Contained)	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Insert (¬Contained)	2 / 1	1 / 1	3 / 1	2 / 0	2 / 1	1 / 0	2 / 1	1 / 0
Insert (Contained)	1 / 1	1 / 0	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Replace (Contained)	1 / 1	1 / 1	3 / 1	3 / 1	– / –	– / –	– / –	– / –
Delete (¬Contained)	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0	1 / 1	1 / 0
Delete (Contained)	1 / 1	1 / 1	2 / 1	2 / 1	1 / 1	1 / 1	1 / 1	1 / 1
Total	**8 / 7**	**7 / 4**	**12 / 7**	**11 / 4**	**7 / 6**	**6 / 3**	**7 / 6**	**6 / 3**

6. Assessing Performance Characteristics

We further evaluate the performance characteristics of CHAMP and MEMCHAMP. While the former is most comparable to Clojure's HAMTs, the latter is most comparable to Scala's implementations. Specifically, we compare to Clojure's `PersistentHash{Set,Map}` and Scala's immutable `Hash{Set,Map}` implementations. We used latest stable versions available of Scala (2.11.6) and Clojure (1.6.0) at the time of evaluation.

The evaluated CHAMP data structures are used daily in the runtime environment of the Rascal[7] programming language, and are currently being tested for inclusion into the Object Storage Model [32] of the Truffle language implementation framework. We report on this to assert that our implementations have been well tested and used by users other than ourselves, mitigating threats to the internal validity of the following evaluation.

Assumptions. We evaluate the pure overhead of operations on the data structures, without considering cost functions for `hashCode` and `equals` methods. Our performance assessment is supposed to reveal the overhead of maintaining bitmaps, incremental hash codes and hash code memoization.

Hypotheses. We expect CHAMP's runtime performance of lookup, deletion, and insertion to equal Clojure's and Scala's runtime performance, albeit canonicalizing and computing over two bitmaps. Runtimes should not degrade below a certain threshold —say 20 % for median values and 40 % for maximum values would just be acceptable— (Hypothesis 1). In general we are betting on significantly better cache behavior and therefore expect to see speedups in iteration (Hypothesis 2). We further expect structural equality checking of two equal collections that do not share reference equal objects to yield at least the same —and likely even higher improvement— as iteration (Hypothesis 3). We also expect that equality checks on derived sets and maps, i.e., measuring `coll.insert(x).delete(x).equals(coll)`, to be orders of magnitude faster (Hypothesis 4).

Despite the addition of a second bitmap, we expect CHAMP maps to use less memory than Clojure's maps because we avoid empty slots (Hypothesis 5). We expect to save a significant amount of memory in big HAMTs because many nodes should have more than two empty slots.

The following hypothesis (Hypothesis 6) formulates our expectations with respect of memory savings of the two HAMT designs: internal values versus leaf nodes. To get a precise expectation we modeled and computed the growth of HAMT-based sets and maps on the Java code skeleton from Figure 1. We assume the absence of empty cells in the `contentArray` and consider the JVM's 8-byte memory alignment. Based on our model [29], we expect for optimal implementations that HAMTs with internal values require roughly half the memory of HAMTs with explicit leaf nodes, both in 32-bit and 64-bit.

With our final hypothesis (Hypothesis 7) we expect the memory footprints of MEMCHAMP to remain close to Clojure's footprints and still vastly improve over Scala's footprints. With field consolidation MEMCHAMP avoids memory gaps due to object alignment, while minimizing the number of worst case `hashCode` and `equals` methods invocations to the same level that Scala's implementations do.

[7] `http://www.rascal-mpl.org`

Benchmark Selection. We assess the performance characteristics of CHAMP with microbenchmarks, focusing on the usage of sets and maps to store and retrieve data and to manipulate them via iterations over the entire collection. We deliberately chose to not use existing CPU benchmark suites or randomly selected applications. The reasons are:

- The pressure on the collection API is quite different per selected real-world application. We have no background theory available to randomly select representative applications which use immutable collections without likely introducing a selection bias.
- It is possible to accurately isolate the use of collections and their operations easily in our microbenchmark setup to confirm or deny our hypotheses. Using these results the designers of standard libraries for programming languages as well as their users will be able to make an informed designed choice for the set and map data structures, irrespective of the other important parts of their designs.

So, on the one hand the importance of these optimizations is different for every application and this remains a threat to external validity. On the other hand the results in the following experiments are very general since they hold for every immutable set or map implementation that uses the proposed optimizations.

6.1 Experiment Setup

We use a machine with Linux Fedora 20 (kernel 3.17) and 16 GB RAM. It has an Intel Core i7-2600 CPU, with 3.40 GHz, and an 8 MB Last-Level Cache (LLC) with 64-byte cache lines. Frequency scaling was disabled.

We used Oracle's JVM (JDK 8u25) configured with a fixed heap size of 4 GB. We measure the exact memory footprints of data structures with Google's memory-measurer library.[8] Running times of operations are measured with the Java Microbenchmarking Harness (JMH), a framework to overcome the pitfalls of microbenchmarking.[9] 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 Garbage Collector (GC) between measurement iterations to reduce a possible confounding effect of the GC on time measurements.

Because each evaluated library comes with its own API, we implemented facades to uniformly access them. 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 [3]. 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 structures.[10] 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 map, yielding tuples of type `Map.Entry`.

Equality (Distinct): Comparing two structurally equal data structures. The two object graphs are distinct from each other (i.e., they contain no reference equal elements).

Equality (Derived): Comparing two structurally equal data structures. The second structure is derived from the first by applying two operations: inserting a new element and then removing it again.

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 it also mitigates a threat to external validity: the shape of a tree depends on the hash codes and hash code transformations which may vary between implementations or applications. E.g., Scala's HAMTs apply a bit-spreading transformation to every hash code —similar to `java.util.HashMap`— to counter badly implemented hash functions.

Evaluating HAMT-based sets and maps 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 library— 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.

We first discuss how CHAMP compares against Clojure and Scala (Sections 6.2 and 6.3), before we focus on the performance implications of adding memoization (Section 6.4).

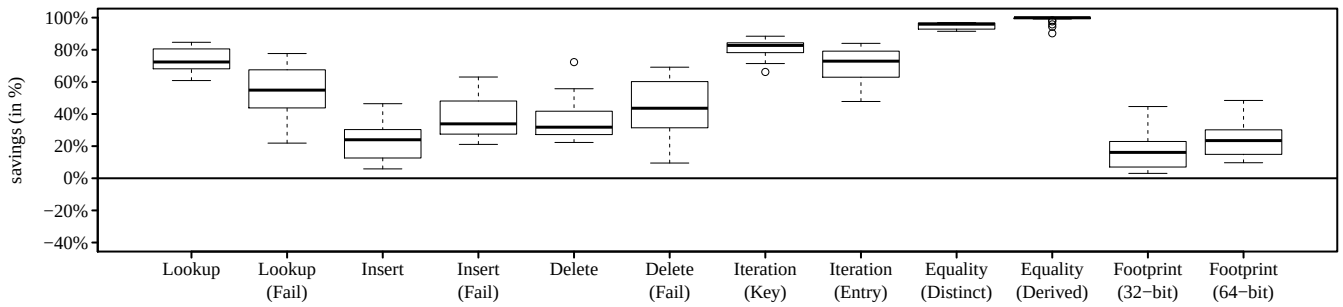
6.2 Runtime Speedup Results

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 0–4.9 % and a median error of 0 %.

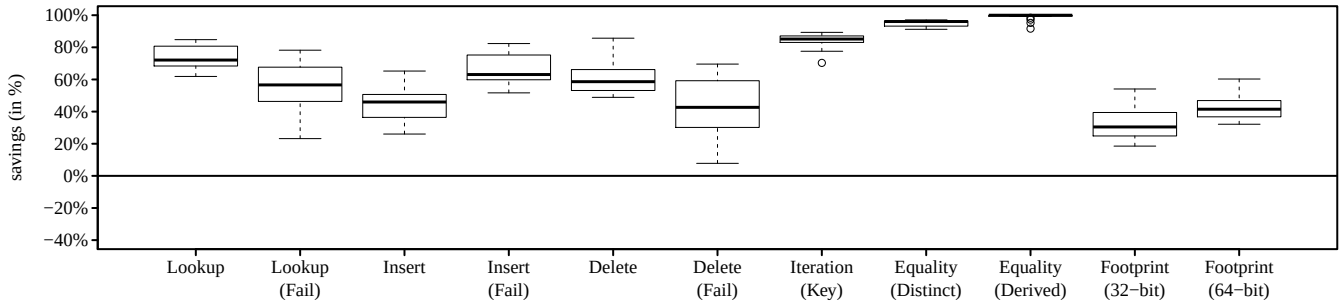
[8] `https://github.com/DimitrisAndreou/memory-measurer`

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

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

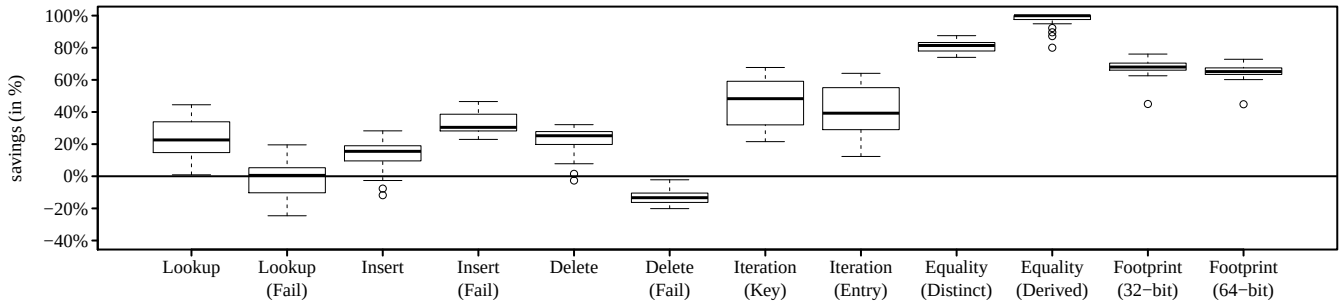


(a) CHAMP versus Clojure's `PersistentHashMap`

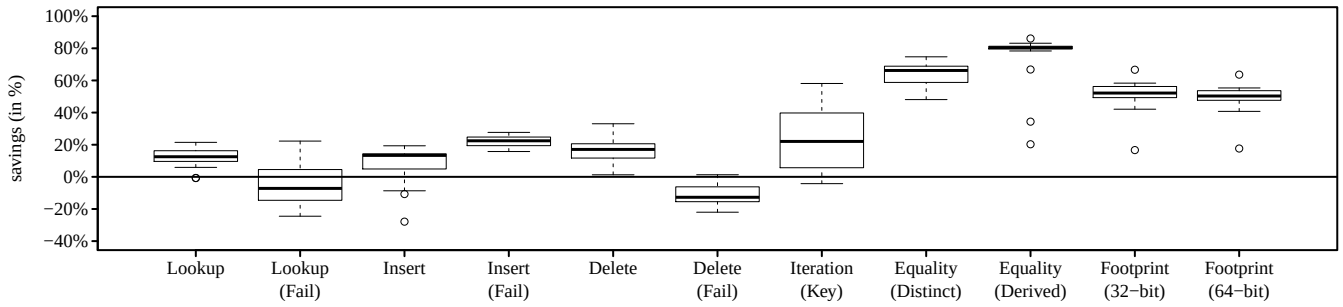


(b) CHAMP versus Clojure's `PersistentHashSet`

Figure 4. Runtime and memory savings of CHAMP compared to Clojure's `PersistentHash{Map,Set}`.



(a) CHAMP versus Scala's `immutable.HashMap`



(b) CHAMP versus Scala's `immutable.HashSet`

Figure 5. Runtime and memory savings of CHAMP compared to Scala's `immutable.Hash{Map,Set}`.

We summarize the data points of the runs with the five different trees with their medians. Then Figure 4a, 4b, 5a, and 5b report for each benchmark the ranges of runtime improvements and memory footprint reductions. Each boxplot visualizes the measurements for the whole range of input size parameters. Because the input sizes are scaled exponentially we report savings in percentages using the following formula, normalizing the size factor: $(1 - \text{measurement}_{\text{CHAMP}} \,/\, \text{measurement}_{\text{Other}}) * 100$.

Speedups Compared to Clojure's Maps. In every runtime measurement CHAMP is better than Clojure. CHAMP improves by a median 72 % for Lookup, 24 % for Insert, and 32 % for Delete. At iteration and equality checking, CHAMP significantly outperforms Clojure. Iteration (Key) improves by a median 83 %, and Iteration (Entry) by 73 %. Further, CHAMP improves on Equality (Distinct) by a median 96 %, and scores several magnitudes better at Equality (Derived).

Speedups Compared to Clojure's Sets. The speedups of CHAMP for sets are similar to maps across the board, with exception of insertion and deletion where it scores even better.

Speedups Compared to Scala's Maps. Lookup performance improves by a median 23 %, especially at small collections until 128 entries (34–45 %). For bigger sizes the advantage is less pronounced. Given that in practice most collections are small [21] and that a usual workload performs more queries than updates, these improvements look promising. For Lookup (Fail) both implementations are neck-and-neck, however CHAMP performs better on the smaller half of the data set and Scala on the bigger half.

Insertion improves by a median 16 %, however with a different pattern from lookup. Insertion performs up to 12 % worse at small collections until 2^5 entries, and then improves again (9–28 %). At Insert (Fail) CHAMP improves across the size range (23–47 %).

Deletion performs with a median runtime reduction of 25 % better than Scala, despite of the compaction overhead. Only for size 2^4 CHAMP is 3 % slower. For Delete (Fail) however, CHAMP is behind Scala all-over (13 % median).

For iteration and equality checking, CHAMP clearly improves upon Scala's immutable hash maps. Iteration (Key) improves by 48 % and Iteration (Entry) by 39 %.

Equality (Distinct) improves by a median 81 %. At Equality (Derived), unsurprisingly, we perform 100 % better because the operation is in a different complexity class.

Speedups Compared to Scala's Sets. The results for sets exhibit similar patterns as the results for maps, however as expected the runtime and memory savings are across the board are slightly lower than for maps. Median improvements for Lookup (13 %), Insert (13 %), and Delete (17 %) highlight that the performance of those operations are similar to Scala. In contrast, CHAMP performs worse in benchmarks Lookup (Fail) and Delete (Fail). In case of the former, CHAMP lags 7–24 % behind Scala from sizes 2^7 upwards, and up to 22 % across the whole size range in case of the latter.

For all other benchmarks we have a handful of measurements where Scala does slightly better. Lookup performed 1 % worse at sizes 2^{20} and 2^{21}. Insert performed up to 11 % slower until 2^4 elements and 28 % slower at size 2^5.

Iteration (Key) improves by a median 22 % and increases up to 58 % for sets bigger than a thousand elements. However, falsifying our hypothesis, iteration performed 4 % worse at size 2^1 and at three sizes (smaller than 128) 1 % slower. The median improvements of 66 % for Equality (Distinct) and 80 % for Equality (Derived) are less pronounced than for maps, but still substantial. Scala implements set equality with a structural `subsetOf` operation that exploits the HAMT encoding and therefore performs well. Scala's subset operation is conceptually a unidirectional version of structural equality.

6.3 Memory Improvements

CHAMP reduces the footprint of Clojure's maps by a median 16 % (32-bit) and 23 % (64-bit), ranging from 3 % to 48 %. The median reductions for sets are 30 % (32-bit) and 41 % (64-bit) respectively, ranging from 19 % to 60 %.

Compared to Scala's maps, CHAMP saves a median 68 % (32-bit) and a median 65 % (64-bit) memory. Savings range from 45 % to 76 %. Note that CHAMP supports a transient representation for efficient batch updates, like Clojure does, and therefore uses one more reference per node. For completeness' sake we also tested a version of CHAMP without support for transients —similar to Scala in terms of functionality— that yields median savings of 71 % (32-bit) and 67 % (64-bit).

CHAMP reduces memory footprints over Scala's sets by a median 52 % (32-bit) and 50 % (64-bit). Once again, a CHAMP variant without support for transients yields slightly higher savings of 57 % (32-bit) and 53 % (64-bit) respectively.

6.4 Performance Implications of Adding Memoization

Adding memoization of the full 32-bit hash codes improves worst case performance of lookup, insertion and deletion in practice (cf. Table 1). In our microbenchmarks this effect is not observable because we evaluate the performance overhead that incurs with aforementioned operations. However because we measure overhead, we can evaluate how the performance characteristics of operations change with MEMCHAMP.

Figures 6a, 6b, 7a, and 7b show the results of comparing MEMCHAMP against Clojure's and Scala's data structures. All-over the memory footprint advantage lessens, due to the additional (consolidated) array of hash codes per node. As a consequence, MEMCHAMP consumes a median 15 % (and maximally 35 %) more memory than Clojure's maps. MEMCHAMP still outperforms Clojure in all operations, with exception of single data points at Insert for maps.

Comparing to Scala, MEMCHAMP retains memory footprint reductions of at least 49 % for maps at sizes bigger than 2^2, and reductions of 15–51 % for equally sized sets. The only outliers here are measurements for size 2^1 where the addi-

tional array has a negative impact. The runtimes of lookup and update operations show a similar profile —but worse net runtimes— than CHAMP. MEMCHAMP's allover performance declines, although median runtimes of aforementioned operations are still close to Scala, with exception of Delete (Fail). The reasons for this are twofold. First, locality decreases because hash codes are stored apart from the elements in a separate array. Second, copying, modifying, and maintaining a second array increases the runtimes of operations.

6.5 Summary

Hypothesis 1 has to be answered case-by-case. With respect to Clojure, it is confirmed. Both variants of CHAMP outperform Clojure's implementations of lookup, insert, and delete.

When compared to Scala, the two inherently different designs reveal varying performance profiles. Hypothesis 1 is confirmed for CHAMP, because it performs mostly faster and not often a bit slower than Scala. The exception is when calling delete with a key that is not present in the data structure. Finally, for MEMCHAMP the hypothesis is falsified, because several data points violate our thresholds of acceptable loss of runtime performance. With matching characteristics in terms of `hashCode` and `equals` calls, MEMCHAMP loses runtime performance over Scala, to gain significant memory savings (cf. Hypothesis 7).

Hypothesis 2 is confirmed as well. Over all implementations, the median speedups for iteration range from 22–85 %. However, counter to our expectations CHAMP performs up to 4 % worse on some small sets when compared to Scala.

Hypothesis 3 is confirmed. Structural equality checking of two collections that do not share reference equal objects improves by 96 %, 96 %, 81 % and 66 % (medians) over the competing HAMT implementations.

Hypothesis 4 is confirmed. Structural equality checking of two derived collections improves by median 80–100 %, with speedups up to 34 x.

Hypothesis 5 is confirmed. Despite increasing the memory footprint per node, CHAMP-based maps decrease overall memory footprints by up to 46 % compared to Clojure's maps. We conclude that savings due to more efficient compaction outweigh the overhead of the addition of a second bitmap.

Hypothesis 6 is confirmed. When compared to Scala's design with leaf nodes, CHAMP reduces memory footprints by median 65 % for maps and 50 % for sets.

Hypothesis 7 is confirmed as well. MEMCHAMP adds little memory overhead over Clojure's implementation, for a great part also due to our savings from Hypothesis 6. The median savings over Scala's HAMTs still range from 27 % to 56 %.

To conclude, despite its more complex encoding, CHAMP achieves excellent runtimes across all tested operations. MEMCHAMP does add overhead over CHAMP, nevertheless runtimes of lookup, insertion, and deletion and memory footprints remain competitive. The significant improvements of the runtime performance of iteration and structural equality checking are observable for CHAMP and MEMCHAMP likewise.

7. Realistic Case: CFG Dominators

Next to microbenchmarks which isolate important effects experimentally, we also need to evaluate the new design on a realistic case to be able to observe its relevance in relation to other (unexpected) factors. In contrast to the microbenchmarks that exclude a cost model for `hashCode` and `equals` methods, the realistic case has costs attached to those methods. We chose to use a classic algorithm from program analysis which is used in optimizing compilers, static analysis, reverse engineering tools and refactoring tools: computing the control flow dominators [1].

Instead of implementing an optimized data structure specifically for the purpose of computing dominators on a Control-Flow Graph (CFG) [8] we picked a most direct implementation finding a maximal solution to the two dominator equations with Scala's and Clojure's HAMTs, and CHAMP:

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

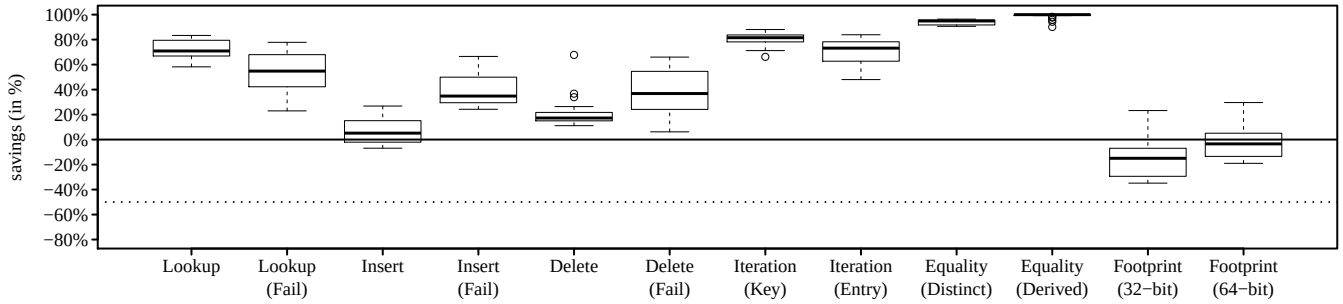
The benchmark source code[11] uses only classical set and relational calculus in a fixed-point loop: Dom and preds are implemented as maps, the big intersection is generated by first producing a set of sets for the predecessors and then folding intersection over it. The goal is to show that such a direct implementation is viable when implemented on top of an optimized representation of immutable sets and maps. In program analysis pipelines simple high-level code, immutability and persistency are beneficial especially for intermediate data structures like dominator sets. It avoids any bugs caused by unnecessary control and data dependencies.

For our experiment, we obtained all the ± 5000 control flow graphs for all units of code (function, method and script) of Wordpress,[12] one of the most popular open-source Content Management Systems written in PHP, using the PHP AiR framework [16]. We then applied the aforementioned dominator implementations to measure CPU time. We used JMH to measure dominator calculations on a randomly-sampled subset of all CFGs. Sample sizes ranged from 128 to 4096 in exponential steps; we omitted smaller samples due to the expected long tail distribution of the CFGs.

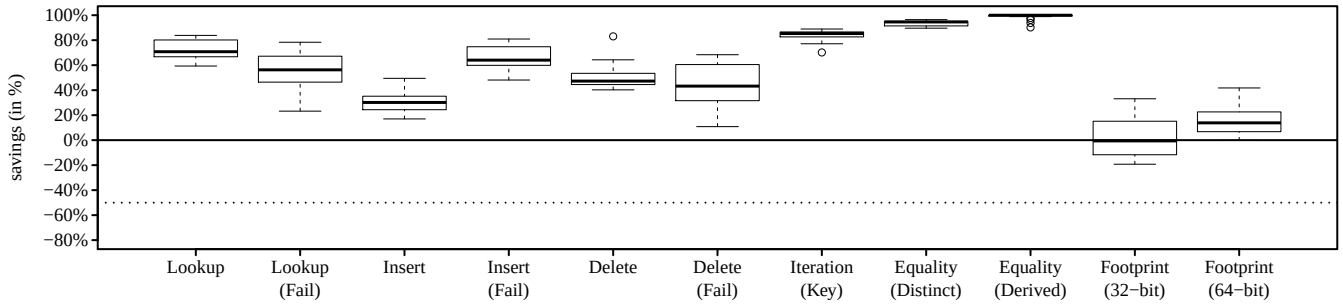
This experiment is not trivial since the effect depends on the shape of the real data and the hard-to-predict dynamic progression of the algorithm as it incrementally solves the equations. The nodes in the CFGs we use are the Abstract Syntax Trees of blocks inside PHP units which are arbitrarily complex; equality checks on these sub-structures could overshadow the computations. The hypothesis is that this case should highlight our optimizations showing a significant benefit; if not it will invalidate the relevance of our contributions.

[11] `http://michael.steindorfer.name/papers/oopsla15-artifact`

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

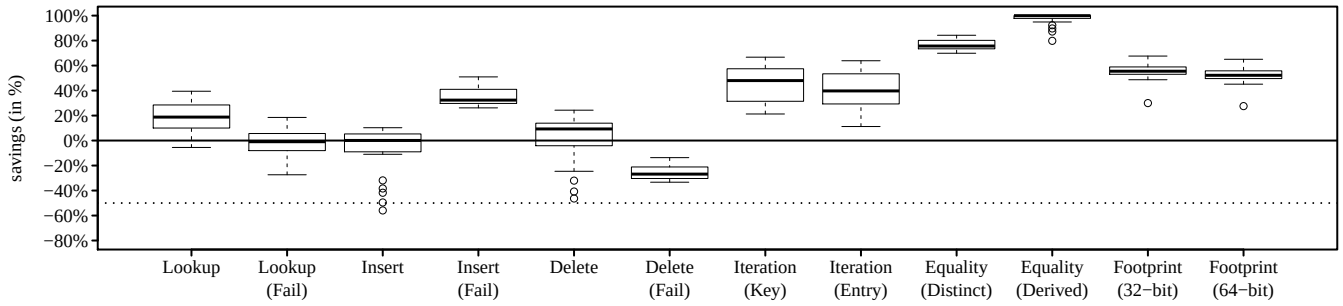


(a) MEMCHAMP versus Clojure's `PersistentHashMap`

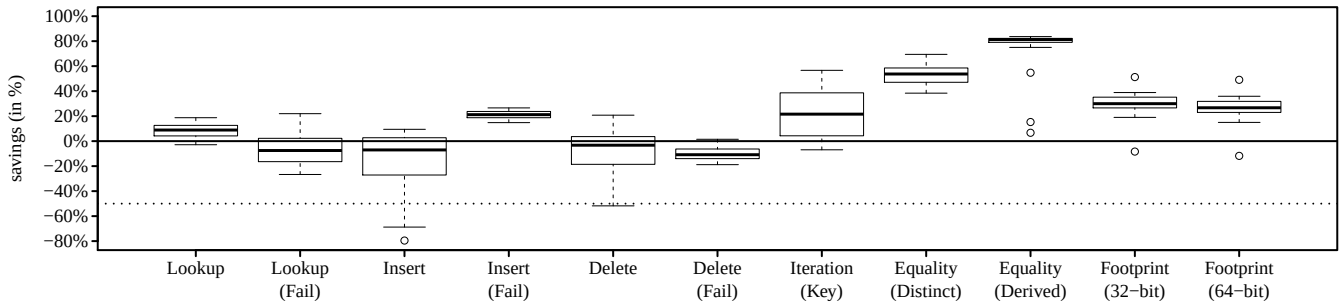


(b) MEMCHAMP versus Clojure's `PersistentHashSet`

Figure 6. Runtime and memory savings of MEMCHAMP compared to Clojure's `PersistentHash{Map,Set}`.



(a) MEMCHAMP versus Scala's `immutable.HashMap`



(b) MEMCHAMP versus Scala's `immutable.HashSet`

Figure 7. Runtime and memory savings of MEMCHAMP compared to Scala's `immutable.Hash{Map,Set}`.

Table 2. Runtimes of Clojure, Scala, and CHAMP for CFG dominators experiment per CFG count. All libraries are unmodified.

#CFG	Clojure	Scala	CHAMP	Speedup w.r.t.	
				Clojure	Scala
4096	1686 s	2654 s	170 s	9.9 x	15.6 x
2048	834 s	1387 s	81 s	10.2 x	17.0 x
1024	699 s	1215 s	61 s	11.4 x	19.8 x
512	457 s	469 s	27 s	16.7 x	17.1 x
256	403 s	418 s	18 s	22.3 x	23.1 x
128	390 s	368 s	14 s	28.1 x	26.5 x

Table 3. Runtimes of Clojure, Scala, and CHAMP for CFG dominators experiment per CFG count. Scala and CHAMP were modified to calculate hash codes lazily, such as Clojure does.

#CFG	Clojure	Scala	CHAMP	Speedup w.r.t.	
				Clojure	Scala
4096	1686 s	1022 s	565 s	3.0 x	1.8 x
2048	834 s	535 s	289 s	2.9 x	1.8 x
1024	699 s	461 s	243 s	2.9 x	1.9 x
512	457 s	277 s	153 s	3.0 x	1.8 x
256	403 s	241 s	132 s	3.1 x	1.8 x
128	390 s	228 s	123 s	3.2 x	1.8 x

7.1 Results

Table 2 shows the mean runtimes of 10 benchmark executions. Measurement errors for Clojure and CHAMP are smaller than 1 s, however Scala's measurements varied by 8–18 s.

CHAMP's runtimes range from 14–170 s, Clojure ranges from 390–1686 s, and Scala from 368–2654 s. To summarize, CHAMP computed the biggest CFG sample of size 4096 more than two times faster than Clojure and Scala could compute the smallest sample of size 128. Overall, the speedups range from minimal 9.9 x to 28.1 x. The highest speedups were achieved at smaller sample sizes.

Instrumented Libraries to Normalize Results. By profiling and code inspection we identified differences in how the libraries implement the hash code operation. Instead of caching, Scala always recomputes the hash code for collections. This necessarily follows from its design choice where every internal trie node implements the full container API: caching hash codes on every level would incur a major memory overhead. In contrast, Clojure computes collection hash codes lazily, whereas CHAMP incrementally updates them.

These differences do not influence the microbenchmarks since they do not invoke hash code calculations at all. Nevertheless, from the dominators case we conclude that caching hash codes is a critical design element of immutable nested collections. To remove possible bias from hash code calculations, we manually modified the source code of Scala and CHAMP to calculate hash codes lazily such as Clojure does. Subsequently, we ran the dominator experiment again; the results are illustrated in Table 3. CHAMP improves over Clojure by median 3 x, and over Scala by 1.8 x. These speedups are directly accountable to improved iteration and equality checking performance in this realistic case.

7.2 Conclusion

Although the dominators case was selected to demonstrate a positive effect of our optimizations, it is real and the evaluation could have produced a contra-indication of the relevance of the new CHAMP data structure. It is not true that every algorithm will benefit from CHAMP, but this case does provide a strong indication that if you start from the design decision of immutable intermediate data structures (functional programming), then CHAMP is bound to be faster than the traditional renderings of a HAMT on the JVM.

8. Analysis and Threats to Validity

The experimental results in the previous sections answer our hypotheses. From this we learn that the optimizations work but not exactly why they work. The three evaluated HAMT implementations, and their map and set implementations do not only differ from each other, but also from Bagwell's original. In this section we dig deeper to find confirmation of the hypothesis that indeed better locality is the cause of the improvement and we discuss which other factors may be at play to threaten the validity of this claim.

8.1 Differences with Clojure's Implementation

Firstly, Clojure uses a lazy sequence abstraction for their iterators. This extra indirection might cause a performance issue or higher number of cache misses as well. However, we did isolate the effect of our optimizations comparing versions of CHAMP itself (cf. Section 8.4), mitigating this threat to validity.

Secondly, Clojure's `PersistentHashSet` internally wraps a `PersistentHashMap` instead of specializing for the lack of a value reference. This explains why memory savings are bigger for sets than for maps compared to CHAMP, but has no effect on our conclusions otherwise.

Finally, Clojure uses utility functions for calculating hash codes that dispatch (with two `instanceof` checks) on specific interfaces, to apply specialized hash functions. In our case, Clojure delegated to the standard `hashCode` method. However, these utility functions are called in lookup, insert, and delete and may have a small negative effect on performance.

Table 4. Preliminary measurements of Last-Level Cache misses for data structures of size 2^{23}. The number in brackets illustrate how much CHAMP reduces cache misses over the other implementations.

Operation	Last-Level Cache Misses for Maps			Last-Level Cache Misses for Sets		
	CHAMP	Scala	Clojure	CHAMP	Scala	Clojure
Equality (Distinct)	112 682	364 452 (3.2 x)	157 240 (1.4 x)	100 576	205 710 (2.0 x)	203 176 (2.0 x)
Equality (Derived)	82 744	351 063 (4.2 x)	146 735 (1.8 x)	71 348	171 872 (2.4 x)	206 862 (2.9 x)
Iteration (Key)	110 397	333 985 (3.0 x)	152 210 (1.4 x)	99 268	205 177 (2.1 x)	160 346 (1.6 x)
Iteration (Entry)	109 979	341 010 (3.1 x)	147 221 (1.3 x)	–	–	–

8.2 Influence of Internal Values versus Leaf Values

The most fundamental difference between CHAMP and Scala's implementation is how they store the content. Conceptually, Scala's leaf nodes mirror Java's `Map.Entry` instances, and therefore do not require boxing of tuples when iterating over entries. With leaf nodes, Scala's HAMT design stores the content elements exactly one tree level deeper than other HAMTs. Whereas the differences in memory footprints can be directly attributed to this difference in design, the runtime differences for lookup, insertion, and deletion can not. The median differences between CHAMP and Scala (up to 30 %) could be either due to the additional level of memory indirections or because of implementation details.

8.3 Modeling Costs of Hash Codes and Equality

The microbenchmarks deliberately focused on the overhead of operations and excluded explicit costs for `hashCode` and `equals` methods. We mitigated this concern by providing a realistic benchmark that has costs attached, and by additionally microbenchmarking MEMCHAMP, a CHAMP variant that exactly matches Scala's memoized design in numbers of `hashCode` and `equals` invocations. Furthermore, if for a particular workload the expected costs for `hashCode` and `equals` are known, one could establish a worst case cost calculation based on Table 1 by weighting the operation runtimes.

8.4 Isolating the Iteration and Equality Contributions

We internally validated that the speedups for structural equality checking and iteration are due to our contributions by selectively switching on/off code parts. We are using code generators to produce all different variants of CHAMP implementations to mitigate human error.

Thus, we compared our final CHAMP design with a number of variants. E.g., for equality checking we removed the overriding `equals` implementations (cf. Listing 5 and Listing 6) to fallback to the equality behavior of `java.util.AbstractSet` and `AbstractMap` implementations. These mini-experiments do confirm that the compaction and canonical forms are the main source of performance improvement.

8.5 Observing Cache Misses

Most of our hypotheses in the evaluation are based on a claim that CHAMP has better cache behavior, but measuring this on a JVM is not so easy. We used Linux *perf* tool to observe low-level cache statistics.[13] In particular, we measured the CPU's hardware events for Last-Level Cache (LLC) misses (i.e., how often a data request could not be served any cache) for the experiments from Section 6. We used JMH's built-in bridge to *perf* for our cache measurements.

In a preliminary setup we applied *perf* to measure selective data points at a sampling rate of 1000 Hz. Because sampling does not report the exact amounts of LLC misses, we restricted our observations to largest input size of 2^{23} to the following benchmarks: Iteration (Key), Iteration (Entry), Equality (Distinct), and Equality (Derived). We expect from the large input size to see the effects of data locality more accentuated. Table 4 shows the results of these experiments. For sets and maps a pronounced effect is observable in terms of cache misses. CHAMP always has fewer cache misses, explaining (at least for a large part) the observed performance benefits. A future more fine-grained analysis uncovering different cache levels may reveal more detail.

8.6 Trie Compression and Canonicalization

On insertion Clojure and Scala compress paths, if and only if the full 32-bit hash codes of two or more elements are equal, by directly allocating a hash collision node. Canonicalization, as presented in Section 4, does not implement this form of compaction currently.

9. Related Work

Trie data structures have been studied since 1959: they were originally invented by Briandais [10] and named a year later by Fredkin [13]. Bagwell [2] and Olsson and Nilsson [23] give an elaborate overview of trie and hash-trie variants and their performance characteristics.

[13] `https://perf.wiki.kernel.org/index.php/Main_Page`

HAMTs and Persistent Data Structures. In 2001 Bagwell [3] described the HAMT, a space-efficient trie variant that encodes the hash code prefixes of elements, while preserving an upper bound in $\mathcal{O}(log_{32}(n))$ on lookup, insertion, and deletion. Bagwell described a mutable hash-map implementation, but his HAMT design was picked up to implement efficient persistent data structures [11, 22]. Persistency refers to purely functional, immutable data structures that are incrementally constructed by referencing its previous states. The first persistent HAMT implementation can be attributed to Rich Hickey, lead-developer of Clojure.

CHAMP builds directly on the foundations of HAMTs, improving iteration and equality checking runtime performance while still reducing memory footprints.

Functional Lists and Vectors Inspired by HAMTs. After the introduction of HAMTs, Bagwell published about functional list implementations [4] that were evaluated in the context of Common Lisp and OCaml runtimes. A decade later Bagwell and Rompf [5] published a techreport about efficient immutable vectors that feature logarithmic runtimes of split and merge operations. Stucki et al. [30] improved upon the latter and performed a broader scale evaluation.

These immutable vectors are Array Mapped Tries (AMTs) and not HAMTs, because they build prefix trees from the indices of densely filled lists. Nevertheless, the implementation of such vectors take many design cues from HAMTs.

Concurrent HAMTs. Prokopec et al. [24] worked on mutable concurrent HAMTs that feature iterators with snapshot semantics, which preserve enumeration of all elements that were present when the iterator was created.

In contrast to Prokopec et al., CHAMP improves a sequential data structure design. However, similar to Clojure's implementation, CHAMP supports edits on a thread-local copy that together with Software Transactional Memory [28] can be used to resolve concurrent modifications.

Generative Programming Techniques. In other work, we applied generative programming techniques [7, 9, 20] to specialize HAMT nodes [29]. We discussed how to reduce the number of specializations from a large exponential number to a small subset while still maximizing memory savings. With this techniques we achieved a median decrease of 55 % in memory footprint for maps and 78 % for sets compared to a non-specialized version, but at the cost of 20–40 % runtime overhead of the lookup operation.

Orthogonal to our previous work, Ureche et al. [31] presented a specialization transformation technique called *miniboxing*. Miniboxing adds specializations for primitive JVM data types to Scala, while reducing the generated bytecode.

In contrast to the generative approach, CHAMP achieves memory reductions and speedups without the need for specialization. However, generative programming techniques could be applied to CHAMP to save even more space.

Cache-Aware and Cache-Oblivious Data Structures [19]. Cache-aware data structures store elements in close space proximity if they are supposed to be used together in close time proximity. Cache-aware data structures exploit knowledge about the memory block sizes, cache lines, etc. Thus, research about cache-aware data structures is mostly concerned with (low-level) system programming languages where the engineer has precise control over memory layout. Java does not offer this: everything beside primitive numeric data types are objects that live on the heap.

In contrast, cache-oblivious data structures try to store elements that are accessed closely in time close to each other, without considering details of cache and memory hierarchies. In that sense, CHAMP can be seen as a cache-oblivious HAMT with respect to iteration and equality checking.

Memory Bloat and Improving Mutable Collections. On the side of mutable collections, Gil et al. [14] identified sources of memory inefficiencies and proposed memory compaction techniques [14] to counter them. They improved the memory efficiency of Java's mutable `Hash{Map,Set}` and `Tree{Map,Set}` data structures by 20–77 % while keeping the runtime characteristics mostly unchanged. In contrast, CHAMP improves runtime performance of immutable collections while also slightly improving memory performance.

10. Conclusion

We proposed CHAMP, a new design for Hash-Array Mapped Tries on the JVM which improves locality and makes sure the trees remain in a canonical and compact representation.

The evaluation of the new data structure design shows that it yields smaller memory footprints and that runtimes can be expected to perform at least the same but usually faster on all operations (with the exception of a slight disadvantage at unsuccessful deletes when compared to Scala). We highlight equality checking and iteration in particular which improves by order of magnitude in a number of cases.

CHAMP's design can be used with or without memoization of hash codes and thus reconciles the benefits of Clojure's and Scala's implementations such as small footprints and improved worst case runtime performance.

We further showed that memoizing and incrementally updating collection hash codes is a critical design element of nested collections, which can be added to sets without a cost.

The proposed data structures are currently used within the runtime data structures of the Rascal programming language. We expect CHAMP to be relevant for standard libraries of other JVM programming languages as well.

Acknowledgments

We thank Mark Hills for providing PHP control flow graphs, and our colleagues and the anonymous referees for providing feedback on earlier drafts of this paper.

References

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

[2] P. Bagwell. Fast And Space Efficient Trie Searches. Technical Report LAMP-REPORT-2000-001, Ecole polytechnique fédérale de Lausanne, 2000.

[3] P. Bagwell. Ideal Hash Trees. Technical Report LAMP-REPORT-2001-001, Ecole polytechnique fédérale de Lausanne, 2001.

[4] P. Bagwell. Fast Functional Lists, Hash-Lists, Deques, and Variable Length Arrays. Technical Report LAMP-REPORT-2002-003, Ecole polytechnique fédérale de Lausanne, 2002.

[5] P. Bagwell and T. Rompf. RRB-Trees: Efficient Immutable Vectors. Technical Report EPFL-REPORT-169879, Ecole polytechnique fédérale de Lausanne, 2011.

[6] 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.

[7] T. J. Biggerstaff. A perspective of generative reuse. *Annals of Software Engineering*, 5(1):169–226, 1998. ISSN 1022-7091.

[8] K. D. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. Technical Report TR-06-33870, Rice University, 2006.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press, 2000.

[10] 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. ACM Press, 1959.

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the 18th annual ACM symposium on Theory of computing*. ACM, 1986.

[12] J. Ebert, D. Bildhauer, H. Schwarz, and V. Riediger. Using difference information to reuse software cases. *Softwaretechnik-Trends*, 2007.

[13] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9): 490–499, 1960.

[14] 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.

[15] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer-Verlag, 2006.

[16] 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.

[17] 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.

[18] 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.

[19] R. E. Ladner, R. Fortna, and B.-H. Nguyen. A Comparison of Cache Aware and Cache Oblivious Static Search Trees Using Program Instrumentation. In *Experimental Algorithmics*. Springer-Verlag, 2002.

[20] D. McIlroy. Mass-Produced Software Components. In P. Naur and B. Randell, editors, *Proceedings of NATO Software Engineering Conference*, pages 138–155. NATO Scientific Affairs Division, 1968.

[21] 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.

[22] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[23] R. Olsson and S. Nilsson. TRASH A dynamic LC-trie and hash data structure. In *High Performance Switching and Routing, 2007. HPSR '07. Workshop on*. IEEE, 2007.

[24] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. In *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. ACM, 2012.

[25] P. Rademaker. Binary relational querying for structural source code analysis. Universiteit Utrecht. Master's thesis, 2008.

[26] I. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of the 12th International Conference on Logic Programming*. Elsevier, 1995.

[27] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7): 669–679, 1986.

[28] N. Shavit and D. Touitou. *Software transactional memory*. ACM Press, 1995.

[29] M. J. Steindorfer and J. J. Vinju. Code Specialization for Memory Efficient Hash Tries (Short Paper). In *GPCE '14: Proceedings of Generative Programming Concepts & Experiences*. ACM, 2014.

[30] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell. RRB Vector: A Practical General Purpose Immutable Sequence. In *ICFP '15: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

[31] V. Ureche, C. Talau, and M. Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *OOPSLA '13: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*. ACM, 2013.

[32] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An object storage model for the truffle language implementation framework. In *PPPJ '14: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform*. ACM, 2014.