Code Specialization for Memory Efficient
Hash Tries (Short Paper)

Michael J. Steindorfer

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

Jurgen J. Vinju

Centrum Wiskunde & Informatica, The Netherlands
Jurgen.Vinju@cwi.nl

Abstract

The hash trie data structure is a common part in standard collection libraries of JVM programming languages such as Clojure and Scala. It enables fast immutable implementations of maps, sets, and vectors, but it requires considerably more memory than an equivalent array-based data structure. This hinders the scalability of functional programs and the further adoption of this otherwise attractive style of programming.

In this paper we present a product family of hash tries. We generate Java source code to specialize them using knowledge of JVM object memory layout. The number of possible specializations is exponential. The optimization challenge is thus to find a minimal set of variants which lead to a maximal loss in memory footprint on any given data. Using a set of experiments we measured the distribution of internal tree node sizes in hash tries. We used the results as a guidance to decide which variants of the family to generate and which variants should be left to the generic implementation.

A preliminary validating experiment on the implementation of sets and maps shows that this technique leads to a median decrease of 55% in memory footprint for maps (and 78% for sets), while still maintaining comparable performance. Our combination of data analysis and code specialization proved to be effective.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization; E.1 [*Data Structures*]: Trees

General Terms Experimentation, Measurement, Performance

Keywords Code generation, Specialization, Hash trie, Persistent data structure, Immutability, Performance, Memory optimization

1. Introduction

Trie data structures have been studied since more than 55 years, yet major performance improvements in memory usage are still possible using generative programming. Tries are used to implement efficient persistent set and map data structures [5, 9]. They were originally invented by Briandais [4] and named a year later by Fredkin [6]. Persistency —in this context— means a functional, immutable data structure that is incrementally built by referencing

its previous states; the previous state is what is persistent. In 2001 Bagwell [1] described a Hash Array Mapped Trie (HAMT), a space-efficient trie that encodes common hash code prefixes of elements, while preserving an upper bound in O $(log_{32}(n))$ on lookup, insert, and delete operations. Bagwell's contribution is a corner stone for immutable collection libraries of modern programming languages that run on the Java Virtual Machine, such as Clojure and Scala.

The hash trie design space. Firstly, the current versions of the aforementioned collections libraries can be considered to be quite optimized, yet we need better memory behavior from a HAMT implementation for the sake of scalability. This is one reason why we explored generative programming for specializing the implementation of a HAMT's internal nodes. Secondly there exist a number of very similar uses of HAMT implementation strategies for different kinds of data structures which cannot be modeled using Java generic programming techniques without loss of efficiency.

Hash tries exist in many variations in standard collection libraries of programming languages. These are the variation points:

- **Update semantics:** Hash tries can have *immutable* semantics, *mutable* semantics, or *staged mutability*.
- **Processing semantics:** *sequentially*, *concurrent*, or *in parallel*.
- **Data type semantics:** sets ($element \mapsto boolean$), maps ($key \mapsto value$), and vectors ($index \mapsto element$).
- **Shape of internal nodes:** Hash trie nodes are n-ary, where n commonly defaults to 32.

The Scala collection library is split by the following dimensions: *mutable/immutable* and *sequential/parallel/concurrent*. Within these categories there exist distinct data types for *set/map/vector* semantics. The Clojure library contains one implementation for hash trie-based maps and one for hash trie-based vectors, while sets are implemented as wrappers for maps ($key \mapsto boolean$). Wrapping enables reuse, but at the cost of memory efficiency. These choices illustrate the (common) problem with the family: any manual decomposition of one dimension will make variation in the other dimension impractical or even infeasible.

In other words, the hash trie data structure seems like an ideal case for generative programming in the traditional sense [2, 3, 8]. We expect to both specialize for better efficiency and to factor the common code for ease-of-maintenance. On the other hand, especially the shape of the internal nodes makes the size of the product family very large. Generating the code, loading it, "jitting" it and keeping it in the CPU's caches would all be hard and lead to performance penalty. Instead, we should find a way to limit the number of specializations necessary to achieve better memory behavior without too much losing run-time performance. This is the technical challenge of this work; code generation is an enabler here, but it requires careful design to benefit from it.
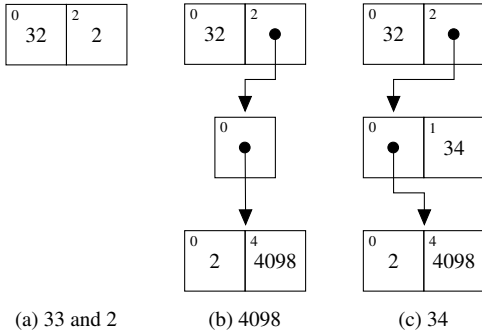
(a) 33 and 2   (b) 4098   (c) 34

Figure 1. Inserting numbers into a trie (array indices in top-left).

Contributions

- Statistical analysis of the shape and distribution of hash trie nodes in practice. This evidence guides the selection of product family members to specialize.

- A hash trie specialization layout that reduces the amount of possible specializations from exponential to quadratic. Incorporating results from our analyses, we show how to further restrict the amount of useful specializations.

- Early experimental evidence that the above techniques do decrease memory usage of immutable sets by 78% and maps by 55%, while maintaining comparable performance.

The resulting hash trie family that we can generate appears to significantly outperform the current state-of-the-art implementations in terms of memory consumption.

2. Introduction into Hash Tries

A general trie is a lookup structure for finite strings that looks and acts like a finite automaton (DFA) without any loops: the transitions are the elements of the strings, the internal nodes encode prefix sharing, and the accept nodes may point to values associated with the strings. In a hash trie, the strings are the bits of a hash-code of the elements stored in the trie. We use lazily created hash tries, in the sense that internal nodes are only created when two hash-code prefixes would collide, leading to internal nodes with values and references to further "states" stored along.

For example, we sequentially insert objects with the following 32-bit hash-codes into a set: 32, 2, 4098, 34. Figure 1 visualizes the (intermediate) hash trie states as these values are inserted. A hash trie distinguishes elements by their (binary) hash-code prefixes[1]:

$$32 : \ldots 00000\ 00001\ 00000_2 = \ldots 0\ 1\ 0_{32}$$
$$2 : \ldots 00000\ 00000\ 00010_2 = \ldots 0\ 0\ 2_{32}$$
$$4098 : \ldots 00100\ 00000\ 00010_2 = \ldots 4\ 0\ 2_{32}$$
$$34 : \ldots 00000\ 00001\ 00010_2 = \ldots 0\ 1\ 2_{32}$$

We have hash tries with a maximal arity of 32 (n-ary trees, with $n = 32$). To select the path sequence that indicates where a value is inserted, we first separate a hash code in chunks of 5 bits (values ranging between 0 and 31).

We expand the tree structure until every prefix can be unambiguously stored. In our example: number 32 is inserted at the root node; number 2 as well (because they do not share a common prefix). Number 4098 shares the prefix path $\rightarrow 2 \rightarrow 0$ with number 2, consequently it is placed unambiguously on level 3. Number 32 shares the

---

[1] We use "prefix" here for consistency with the literature, but we are looking at the least significant bits first so on a little endian PC its the postfix.

```
1  abstract class TrieSet implements java.util.Set {
2    TrieNode root; int size;
3    class TrieNode {
4      int bitmap; Object[] contentAndSubTries;
5    }
6  }
```

Listing 1. Skeleton of a hash trie-based set data structure in Java.

```
1  abstract class TrieSet implements java.util.Set {
2    TrieNode root; int size;
3
4    interface TrieNode { ... }
5    ...
6    class NodeNode extends TrieNode {
7      byte pos1; TrieNode nodeAtPos1;
8      byte pos2; TrieNode nodeAtPos2;
9    }
10   class ElementNode extends TrieNode {
11     byte pos1; Object keyAtPos1;
12     byte pos2; TrieNode nodeAtPos2;
13   }
14   class NodeElement extends TrieNode {
15     byte pos1; TrieNode nodeAtPos1;
16     byte pos2; Object keyAtPos2;
17   }
18   class ElementElement extends TrieNode {
19     byte pos1; Object keyAtPos1;
20     byte pos2; Object keyAtPos2;
21   }
22   ...
23 }
```

Listing 2. Skeleton of a specialized hash trie-based set in Java.

prefix path $\rightarrow 2$ with numbers 2 and 4098, but can be differentiated from both on level 2.

Listing 1 shows a class skeleton of a hash trie implementation for a set data structure, where the container class, the `TrieSet` contains `size` information and a reference to the `root` node of the trie. The nested `TrieNode` class encodes the possible $n = 32$ sub-tries as a compacted sparse array; the 32-bit integer `bitmap` signals which of the branches are used (for value or child nodes). The size of the array is equal to the number of 1's in the bitmap.

The `contentAndSubTries` array is of type `Object` to either store set elements or references to sub-tries. This compaction technique obviates extra leaf nodes by pulling them up one level. Listing 1 closely resembles Clojure's hash trie implementations.

Both the bitmap and the array are candidates for specialization because they introduce overhead. Here, all possible specializations would lead to thousands of classes. Which classes do we generate for maximum effect on footprint with minimal loss on efficiency?

3. Hash Trie Node Arity Frequency

To answer that question we measure the distribution of node arities. Specialization should focus on the most frequent node arities. The distribution of arities is governed only by the hash-codes. We can assume a uniform hash distribution, because if the hash-codes in a real application do not approximate a uniform distribution, the trie's efficiency would degenerate anyway due to collisions, and our optimizations would be less relevant.

To generate representative data sets, we use the Java pseudo-random number generator. First, we generated 4096 integers between 0–8M, which we used as target sizes for set data structures.[2] For each size we then generated a random sequence of integers to be inserted. Note that in Java the hash-code for an integer equals

---

[2] Bagwell's data set for evaluation included maps up to 8M. For the sake of comparability, we benchmarked the same data points [1].

Table 1. Frequencies and cumulative summed frequencies of tree nodes by arity.

Arity	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
%	1.44	**63.14**	**14.26**	**3.27**	1.24	0.94	0.93	0.96	1.00	1.05	1.11	1.17	1.23	1.28	1.32	1.33
$\sum$ %	1.44	64.58	78.84	**82.10**	83.34	84.29	85.21	**86.17**	87.17	88.22	89.32	**90.49**	91.72	92.99	94.31	95.65

Arity	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
%	1.28	1.09	0.75	0.40	0.15	0.04	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.61
$\sum$ %	96.93	98.01	98.76	99.16	99.32	99.36	99.37	99.37	99.37	99.37	99.37	99.37	99.37	99.38	99.39	100

its value. The results in Table 1 show how a uniform distribution of hash-codes does lead to a non-uniform distribution in node arities. Similar distributions appear every time we vary the sizes of the sets.

It can be seen that the smaller arities (2–4) account for more than 80% of all nodes. In sum, all node combinations with arities 0–4 account for 82% of all nodes, arities 0–8 for 86%, and arities 0–12 for 90.5%. After this there is a long tail of sizes with a an almost uniform distribution to make up for the last 9.5%.

This data suggests the number of specializations can be reduced to 31, while still achieving an impact on 82% of the nodes. Yet we still need to answer two questions. First, how do the impact numbers translate to real memory savings? Second, should we go beyond arities 0–4 with specializing?

4. Modeling and Measuring Memory

Measuring JVM memory consumption precisely takes quite some time and energy. To optimize our experiments, instead we accurately model the memory consumption of hash tries, calibrate the model once, and then continue to optimize using the predictive model. We model these two properties:

1. the footprint of trie nodes, following JVM's memory alignment.

2. the overhead of trie nodes compared to real data stored in nodes.

In Java every object is 8-byte memory aligned, allowing for memory compaction techniques [7]. Consequently, if an object's header together with the size of all fields do not sum to a multiple of eight, your object will be aligned to the nearest 8-byte boundary and consume more memory than strictly necessary. Note that Oracle's HotSpot JVM uses 12-byte object headers in 32-bit mode and 16-byte headers in 64-bit mode. References consume four bytes in a 32-bit JVM, and eight bytes in a 64-bit JVM. Based on this knowledge we model the footprint (fp) of hash trie nodes in formulas.

$$\text{fp}_{32}(n) = \lceil (12 + 4 + 4)/8 \rceil * 8 + \lceil (12 + 4 + 4 * n)/8 \rceil * 8$$
$$\text{fp}_{64}(n) = \lceil (16 + 4 + 8)/8 \rceil * 8 + \lceil (16 + 4 + 8 * n)/8 \rceil * 8$$

The parameter n is the arity of the node. The first part of each formula calculates the footprint of a tree node which consists of the class header, the size of the integer bitmap and the reference to the array. The second part of the formula describes the layout of an array, containing an integer length field with value n, and n slots for references.

We validated the correctness of both formulas on JVM 1.7.0u55, on OS X 10.9.3. To measure the exact footprints of internal trie nodes, we use *memory-measurer*.[3]

In order to put the numbers obtained from the formulas into perspective, we put them into relation to the theoretical minimum amount of data that has to be stored per node, i.e. the number of references to data/sub-tries. Figure 2 shows the overhead per reference a node stores, for each possible arity in 32-bits.[4] It is visible
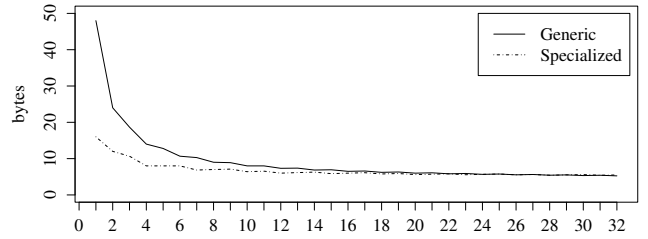

Figure 2. Memory overhead per node arity in 32-bit mode.

that the sparse-array implementation (cf. `TreeNode` in Listing 1) has a significant overhead at small arities and negligible overhead for the larger arities.

This analysis illustrates how trie nodes usually have low arity with a high overhead. Due to the uniformity of hash-codes, there are low chances of sharing prefixes if there are few elements. When the trie fills up, more and more prefixes are shared, lowering the overhead per element. This is another strong argument in favor of only specializing the classes in the lower ranges.[5]

Our goal is to achieve a worst-case per-reference overhead of 16 bytes in 32-bit mode and 24 bytes in 64-bit mode, even for small numbers of elements. For tries, where nodes are represented as objects, this seems a reasonable target since an object with a single reference to it would consume $12 + 4 = 16$ or $16 + 8 = 24$ bytes in 32-bit and 64-bit mode, respectively.

5. Specialized Low Arity Trie Nodes

Listing 2 contains a Java source code skeleton that shows all specializations for arity 2.[6] Instead of a single bitmap we now store the 5-bit chunk of the hash-code for each branch in a byte and the reference to either a child or value node in a field.

The following formula models the footprints for these class layouts, yielding our target 24 bytes for size 2 (which was 48 before):

$$\text{fp}_{32}^s(n) = \lceil (12 + 1 * n + 4 * n)/8 \rceil * 8$$
$$\text{fp}_{64}^s(n) = \lceil (16 + 1 * n + 8 * n)/8 \rceil * 8$$

Given the previous analysis of arity distribution, we save 50% of the memory on at least 80% of the nodes. Figure 2 shows how the overhead drops radically for the smaller factors.

The generated code contains specialized implementations of insert and delete methods, such that we scale incrementally to different specialized classes and eventually escalate into the generic implementation. This is the reason that the fields are ordered and we generated, for example, both `ElementNode` and `NodeElement`. The number of specializations necessary for each arity is the number of permutations: 31 for 4, 511 for 8 and 8191 for 12.

---

[3] `https://code.google.com/p/memory-measurer/`

[4] The picture is comparable for 64-bit mode.

[5] *An optimal, uniform hash distribution results in the worst memory performance of hash tries, on the JVM.*

[6] We used Rascal's auto-indenting recursive templates for code generation.
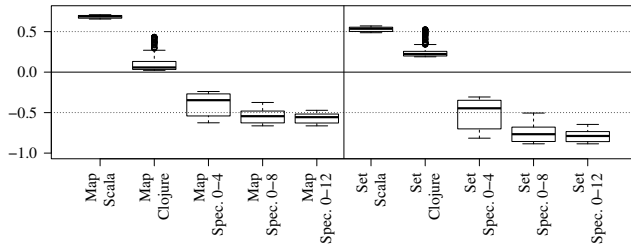
Figure 3. Relative footprints of 32-bit sets and maps compared against our generic implementation (i.e., the zero line).

Figure 4. Relative run-times for `lookup` and `insert` in maps compared against our generic implementation (i.e., the zero line).

6. Not Generating Permutations

We know we may save about half of the memory, but at the cost of generating too many classes. A quick experiment showed that this generates too much of an efficiency overhead. Many of the permutation classes we generate have the exact same types and numbers of fields. If we only generate classes for unique combinations of values and internal nodes, then the number of classes would go down to quadratic, or more precisely to $\sum_{i=1}^{n} \binom{2+i-1}{i}$ classes.

We only store how for a certain arity the node splits into content elements and sub-tries. This yields 15 classes (0–4), 45 classes (0–8), and 91 classes (0–12), and all possible specializations would yield 561 classes (0–32). For example, in Listing 2 classes `ElementNode` and `NodeElement` would collapse to a single class.

We achieve this complexity reduction by dropping the total ordering of elements within in a specialized node. The cost of this optimization is that we need to dynamically sort the entries when we escalate from a specialized representation to the generic trie node. This only happens for nodes with low numbers of elements, necessarily, so the run-time overhead is expected to be very low, especially given that it only happens at the boundary.

7. Evaluation

Here we evaluate memory and run-time efficiency. The goal is to position the memory behavior of the resulting tries to the current Clojure and Scala collections, and we report the effect of specialization. The costs of specialization in terms of run-time overhead are also measured.

We used the following versions: *clojure-1.6.0.jar*, *scala-library-2.10.4.jar*, and a research branch of *pdb.values-0.4.1.jar*, our current library. We compared Clojure's `PersistentHash{Set,Map}`, Scala's immutable `Hash{Set,Map}` and our generic set and map against specialized versions with arities up to 4, 8 and 12.

Memory We used the same experimental setup for evaluating the space savings, as we used in Section 3 for obtaining frequency statistics. We used the first 256 pseudo-randomly generated target sizes and performed a single experiment for every version. Figure 3 shows the results obtained for 32-bit. Our generic trie-map uses less memory than the Scala and Clojure implementations, but the specialized versions still improve on it up to 55% for maps and 78% for sets. The 64-bit measurements are the same, always 5% below the 32-bit results, but exhibit the same magnitude of savings.

Run-time Experiments were run with a 64-bit JVM, version 1.7.0u55, running on an Intel Core i7 3720QM CPU under Mac OS X 10.9.3. We used *Caliper*[7] to run all measurements. We report the median of the 15 last repetitions of each microbenchmark, after Caliper has warmed up the JVM.

For microbenchmarking we reused the earlier setup to generate random sets and maps. For lookup and insert we tested with a
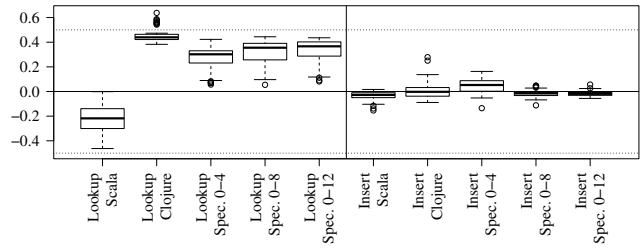
---

[7] https://code.google.com/p/caliper/

sequence of 8 randomly generated examples. The results in Figure 4 show how insertion time is not affected much by the postponed sorting and that our implementation performs the same as the Scala and Clojure implementations. For lookup the Scala code is 20% better than our generic implementation, and our specializations cost between 20% and 40% run-time overhead due to the unordered storage of node elements. We still perform faster lookups than the Clojure implementation though.

Specialization range 0–8 yields best performance characteristics while keeping the number of classes necessary low. It performs better than range 0–4 and equally well as range 0–12.

8. Conclusion

We reduced the memory footprint of hash trie-based set and map implementations by 55–78% by (one-time) generating specializations of hash trie nodes at the cost of a 40% slow-down in lookup efficiency and no loss in efficiency of insertion.

A side-effect was the generation of a product family for the entire multi-dimensional design space for hash trie implementations of collections. The presented library is currently used in the run-time of the Rascal meta-programming language.

As future work we intend to study and evaluate a number of additional optimizations to the current design, mainly focusing on run-time efficiency. This would also include a more in-depth evaluation based on wider and more realistic benchmarks.

References

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

[2] T. J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5(1):169–226, Jan. 1998.

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

[4] R. De La Briandais. File Searching Using Variable Length Keys. In *IRE-AIEE-ACM '59 (Western): Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*. ACM, Mar. 1959.

[5] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. In *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. ACM, Nov. 1986.

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

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

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

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