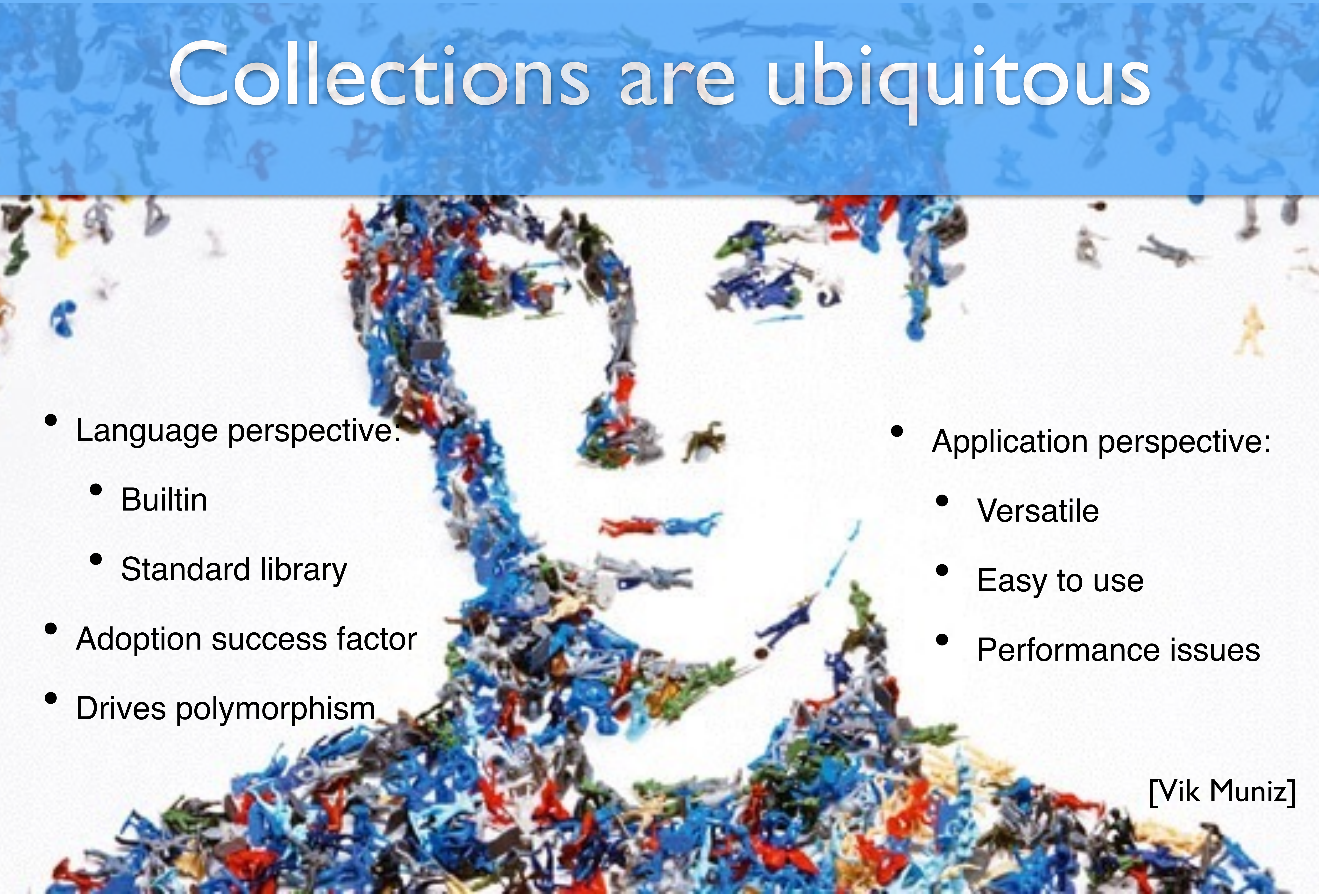
Optimizing Hash-tries for Fast and Lean Immutable Collection Libraries

IFIP WG 2.4 Software Implementation Technology
Stellenbosch, November 2014

Michael Steindorfer
Jurgen J. Vinju
Centrum Wiskunde & Informatica
INRIA Lille & TU Eindhoven

Collections are ubiquitous



- Language perspective:
  - Builtin
  - Standard library
- Adoption success factor
- Drives polymorphism

- Application perspective:
  - Versatile
  - Easy to use
  - Performance issues

[Vik Muniz]

Immutable Collections

- Immutability implies safety
  - sharing with referential integrity
  - equational reasoning
  - co-variant sub-typing
- Overhead
  - Copying
  - More encoding and traversal
  - Unused data
- Special opportunities for optimization
  - Structural equality
  - Hash-consing/maximal sharing
  - Persistence (differencing)

[Michelangelo di Lodovico Buonarroti Simoni]

PhD Challenge

- Design and implement fastest & leanest collections
  - on the JVM
  - sets, maps, relations, vectors, etc.
  - staged [im]mutability
  - “versatile”
  - equals, *insert, delete*, lookup, union, intersection, diff, *iteration*
- For under-the-hood of Rascal MPL

Variability

- For experimentation & comparison
  - simulate published data-structures
  - scala simulation
  - closure simulation
- For versatility
  - builtin data-types
  - hard, soft, weak references
  - ordered/unordered
  - sets vs. maps
  - staged/immediate immutability

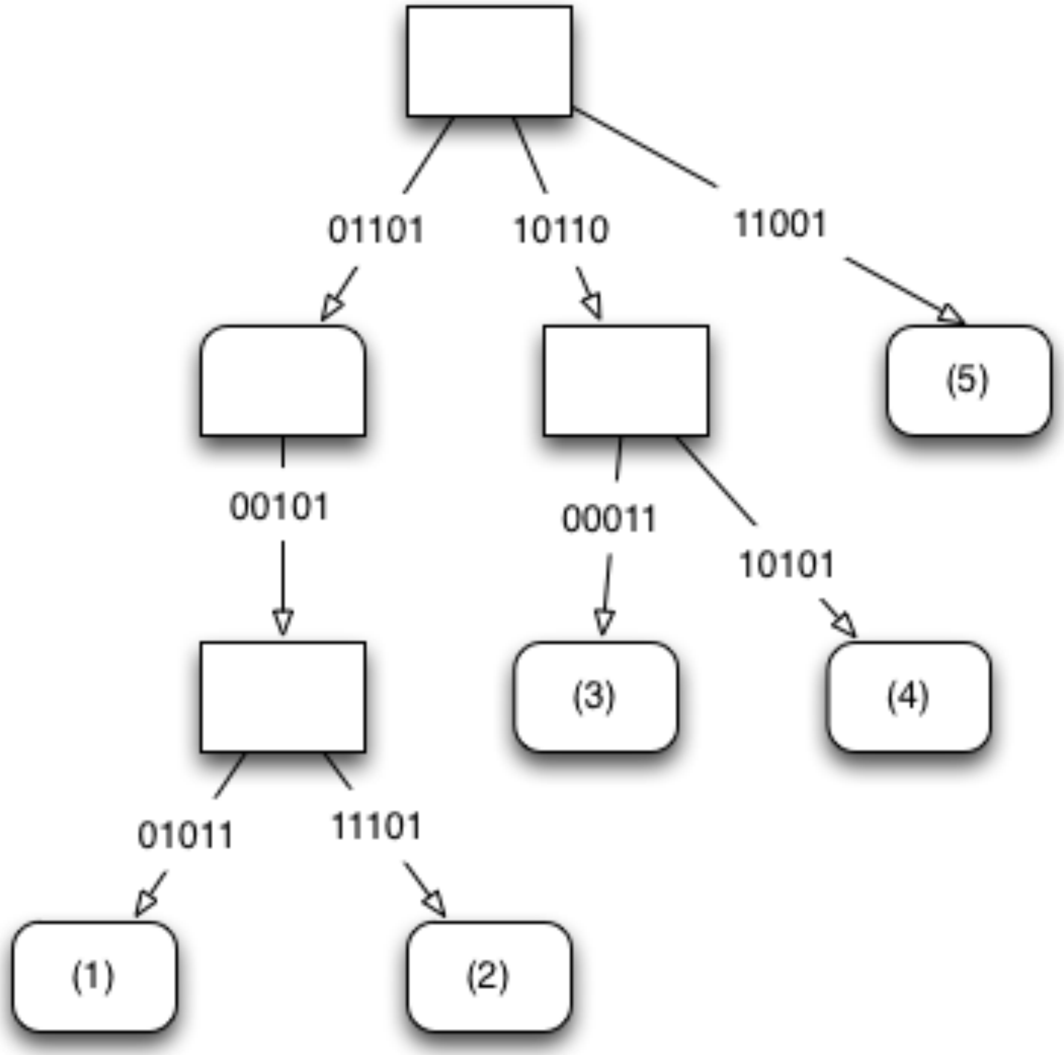
Solution:
Generative Programming

(and you really don't want to (re)write this code)

Results

- Measuring and profiling [submitted] *(not today)*
  - “Object Redundancy and Equals-Call Profiling”
  - Precisely modeling JVM object footprints and alignment
- Leaner [GPCE 2014, ongoing work]
  - “Code Specialization for Memory Efficient Hash Tries”
- Faster [ongoing work]

Hash-array Mapped Tries



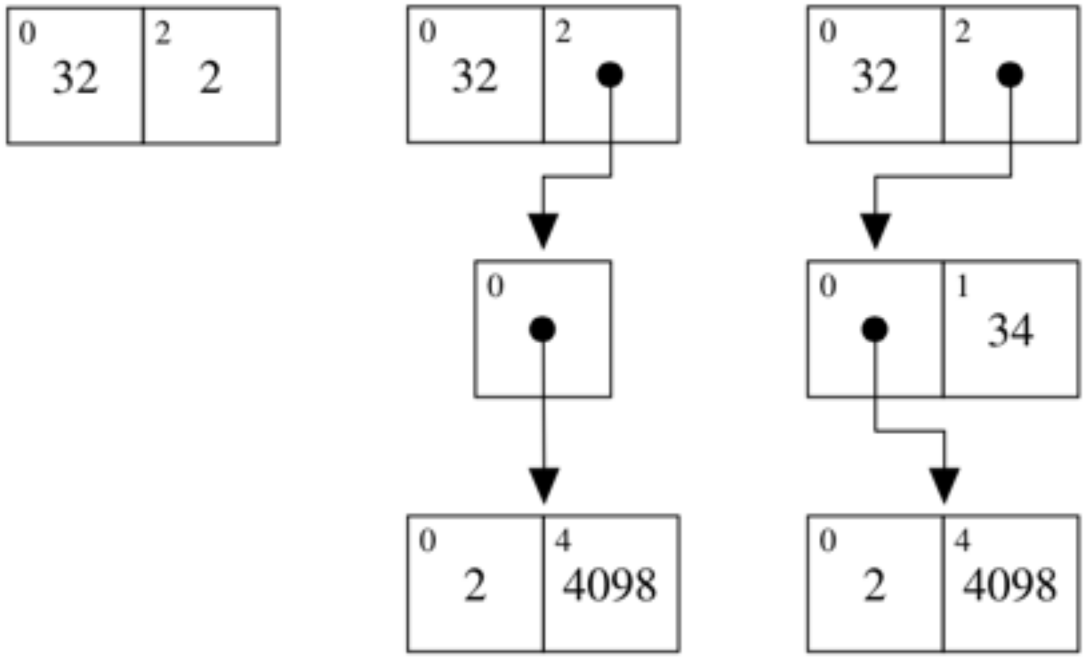
- [Bagwell 2001], Scala, Clojure
- What is a HAMT?
  - Radix tree with hashes
  - Prefix/postfix tree
  - DFA without cycles
- Only expand if prefix overlaps
- Keys are encoded, step-by-step, inside
- Keys are ordered explicitly

Canonical Code

```
class TrieSet implements java.util.Set
{
  TrieNode root;
  int size;

  class TrieNode {
    int bitmap; // 32 bits
    Object[] contentAndSubTries;
    …
  }
}
```

inserting 32, 2, 4098, 34



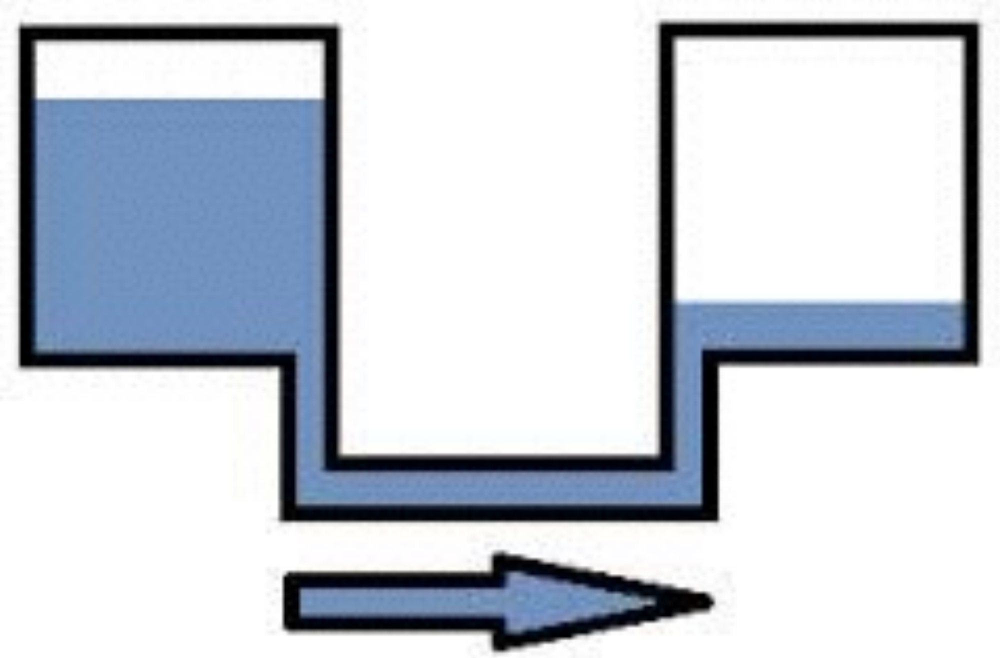
Insert does this:

1. take 5 bits from hash
2. check position
3. store value or recurse

Memory of HAMT

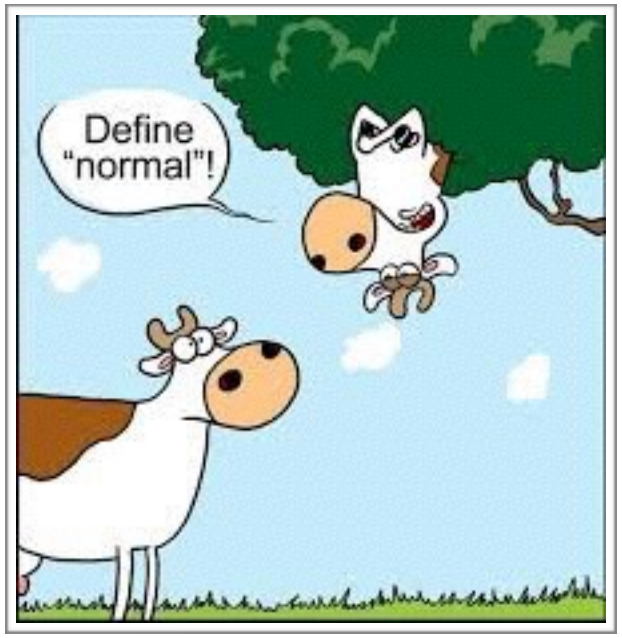
- Compared to hash-tables, hamts have:
  - fewer null array elements
  - possible persistence
  - no resizing
- Compared to dense arrays, hamts have:
  - Bitmaps (on every level)
  - Arrays (on every level)
- Compared to a flat object, hamts have:
  - Extra array
  - Extra bitmap

Speed of HAMT



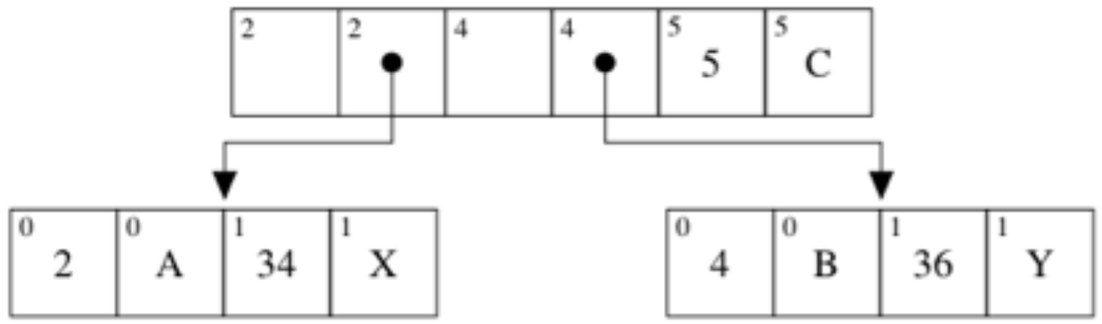
- Reasonable cache locality
- Bit-level operations
- hashCode() and equals()
- Sub-optimal shape of the tree
- *Fixed maximal depth* = 7

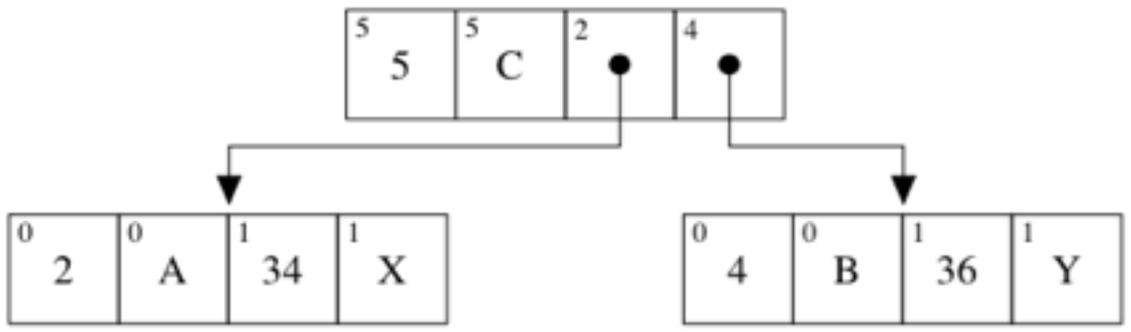
Normalize on delete



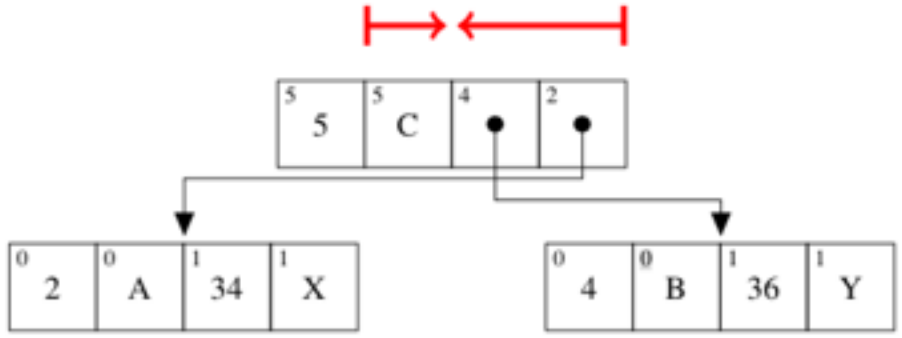
- Removes unnecessary overhead
- Improves locality
- Can assume canonical form
  - allows short-circuiting *equals* more often
- Faster and leaner

Different ordering





- Sets and maps do not need all this ordering
- Much better locality for generators/iteration
- Things to mitigate now:
    - storing the boundary
    - more bit operations
    - moving pointers acros the boundaries



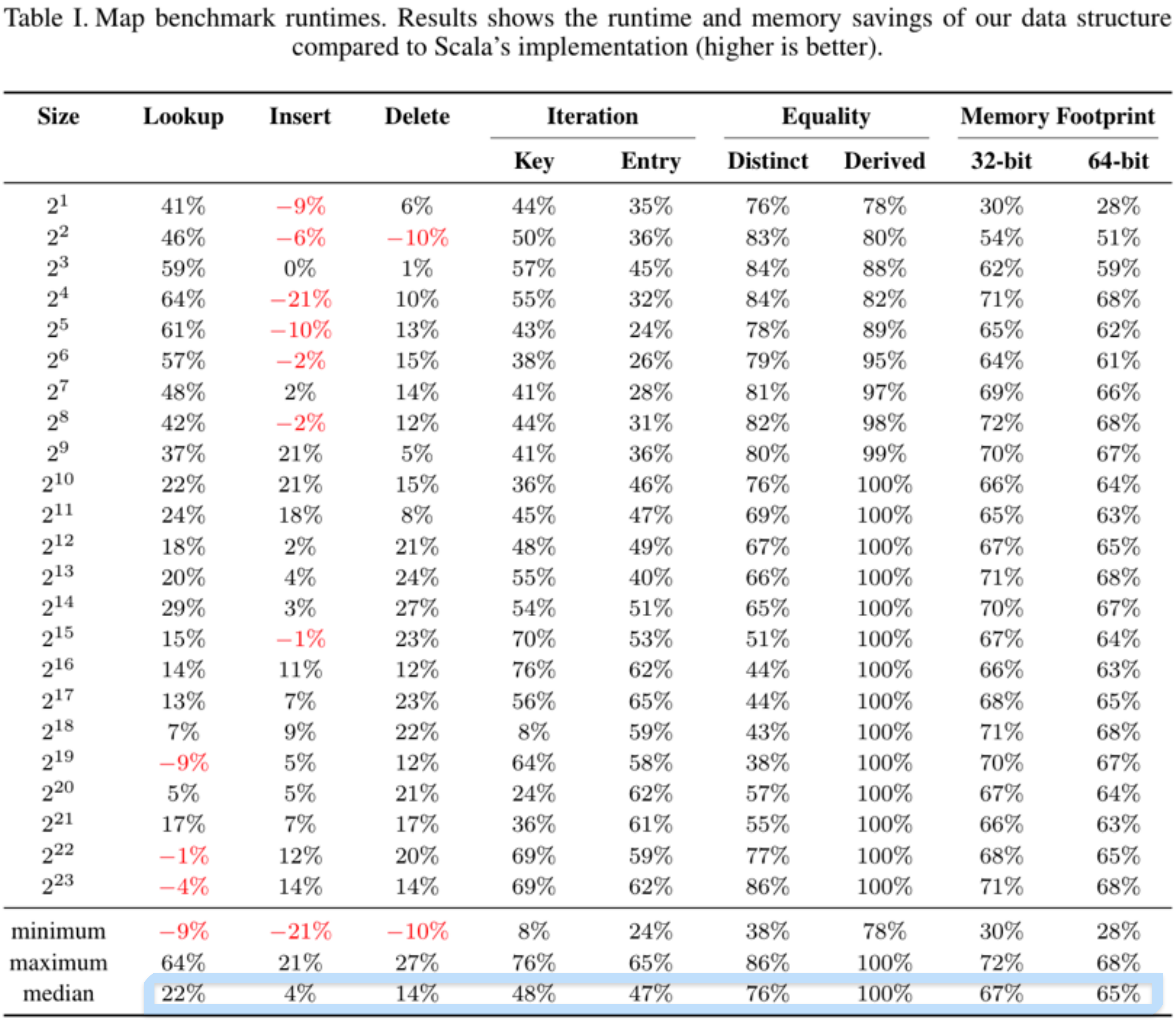


Table I. Map benchmark runtimes. Results shows the runtime and memory savings of our data structure compared to Scala's implementation (higher is better).

Size	Lookup	Insert	Delete	Iteration		Equality		Memory Footprint	
				Key	Entry	Distinct	Derived	32-bit	64-bit
2^{1}	41%	−9%	6%	44%	35%	76%	78%	30%	28%
2^{2}	46%	−6%	−10%	50%	36%	83%	80%	54%	51%
2^{3}	59%	0%	1%	57%	45%	84%	88%	62%	59%
2^{4}	64%	−21%	10%	55%	32%	84%	82%	71%	68%
2^{5}	61%	−10%	13%	43%	24%	78%	89%	65%	62%
2^{6}	57%	−2%	15%	38%	26%	79%	95%	64%	61%
2^{7}	48%	2%	14%	41%	28%	81%	97%	69%	66%
2^{8}	42%	−2%	12%	44%	31%	82%	98%	72%	68%
2^{9}	37%	21%	5%	41%	36%	80%	99%	70%	67%
2^{10}	22%	21%	15%	36%	46%	76%	100%	66%	64%
2^{11}	24%	18%	8%	45%	47%	69%	100%	65%	63%
2^{12}	18%	2%	21%	48%	49%	67%	100%	67%	65%
2^{13}	20%	4%	24%	55%	40%	66%	100%	71%	68%
2^{14}	29%	3%	27%	54%	51%	65%	100%	70%	67%
2^{15}	15%	−1%	23%	70%	53%	51%	100%	67%	64%
2^{16}	14%	11%	12%	76%	62%	44%	100%	66%	63%
2^{17}	13%	7%	23%	56%	65%	44%	100%	68%	65%
2^{18}	7%	9%	22%	8%	59%	43%	100%	71%	68%
2^{19}	−9%	5%	12%	64%	58%	38%	100%	70%	67%
2^{20}	5%	5%	21%	24%	62%	57%	100%	67%	64%
2^{21}	17%	7%	17%	36%	61%	55%	100%	66%	63%
2^{22}	−1%	12%	20%	69%	59%	77%	100%	68%	65%
2^{23}	−4%	14%	14%	69%	62%	86%	100%	71%	68%
minimum	−9%	−21%	−10%	8%	24%	38%	78%	30%	28%
maximum	64%	21%	27%	76%	65%	86%	100%	72%	68%
median	22%	4%	14%	48%	47%	76%	100%	67%	65%

Squeezing space

- The HAMT overhead is
  - bitmap
  - array
- For both the sparsity is defined by node arity:
  - distribution of the input integers/hash-code
  - details like chunk size
- Hypothesis: we can specialize for *node arity*

Specializing Node Arity

- For the ordered version: exponential amount
  - infeasible due to memory, cache, code size
- For the re-ordered version: polynomial amount
  - but we pay in bit-level operations
- For which sizes do we specialize?

Specialized code

```
class TrieSet implements java.util.Set {
   TrieNode root; int size;
   interface TrieNode { ... }
   ...
   class NodeNode extends TrieNode {
     byte pos1; TrieNode nodeAtPos1;
     byte pos2; TrieNode nodeAtPos2;
     …
   }
   class ElementNode extends TrieNode {
     byte pos1; Object key;
     byte pos2; TrieNode node;
     …
   }
   class ElementElement extends TrieNode {
     byte pos1; Object key1;
     byte pos2; Object key2;
     …
   }
   class GenericNode implements TrieNode {
      …
   }
}
```

- code to switch between specialized and generic code
- lookup, insert, delete are more complex
- miminize code generation by having a fragile base class

Experiment

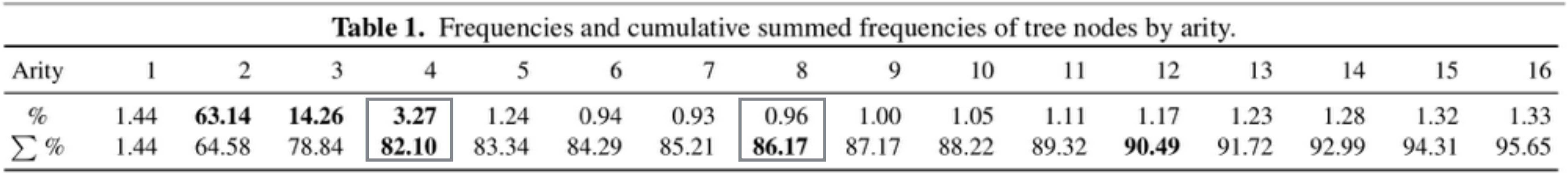


Table 1. Frequencies and cumulative summed frequencies of tree nodes by arity.

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
∑ %	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

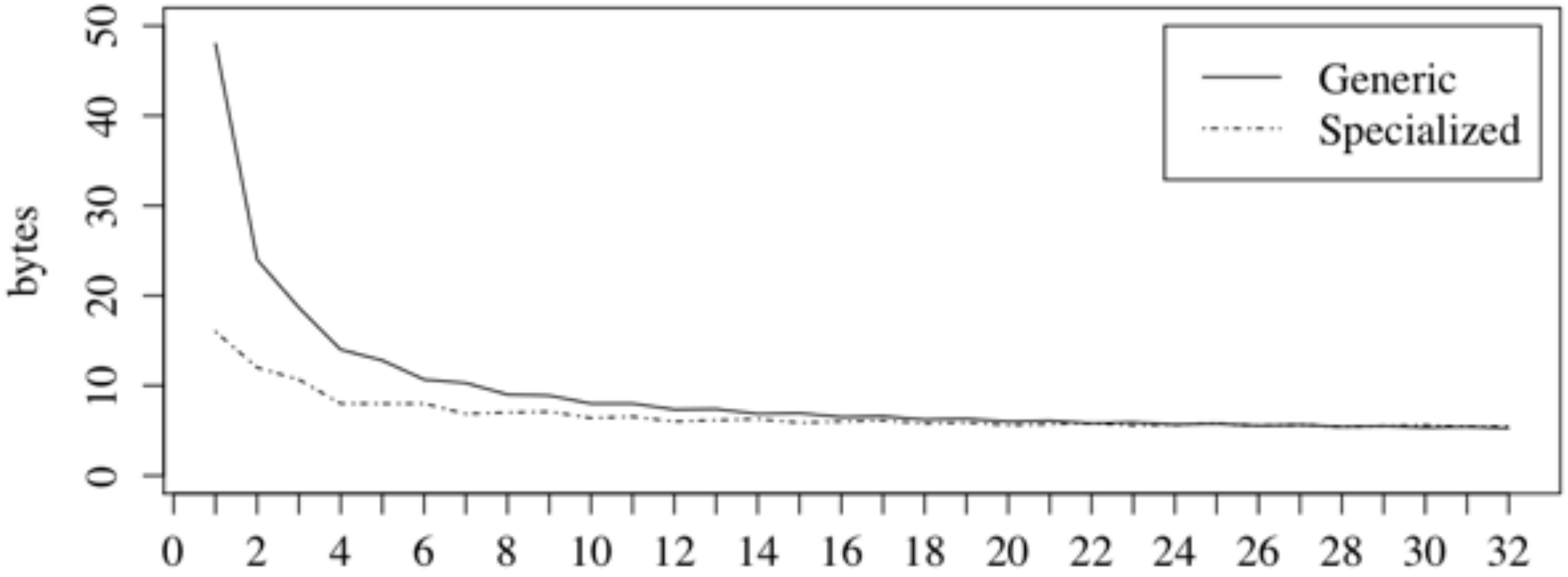


Figure 2. Memory overhead per node arity in 32-bit mode.

Random integers simulating good hash codes

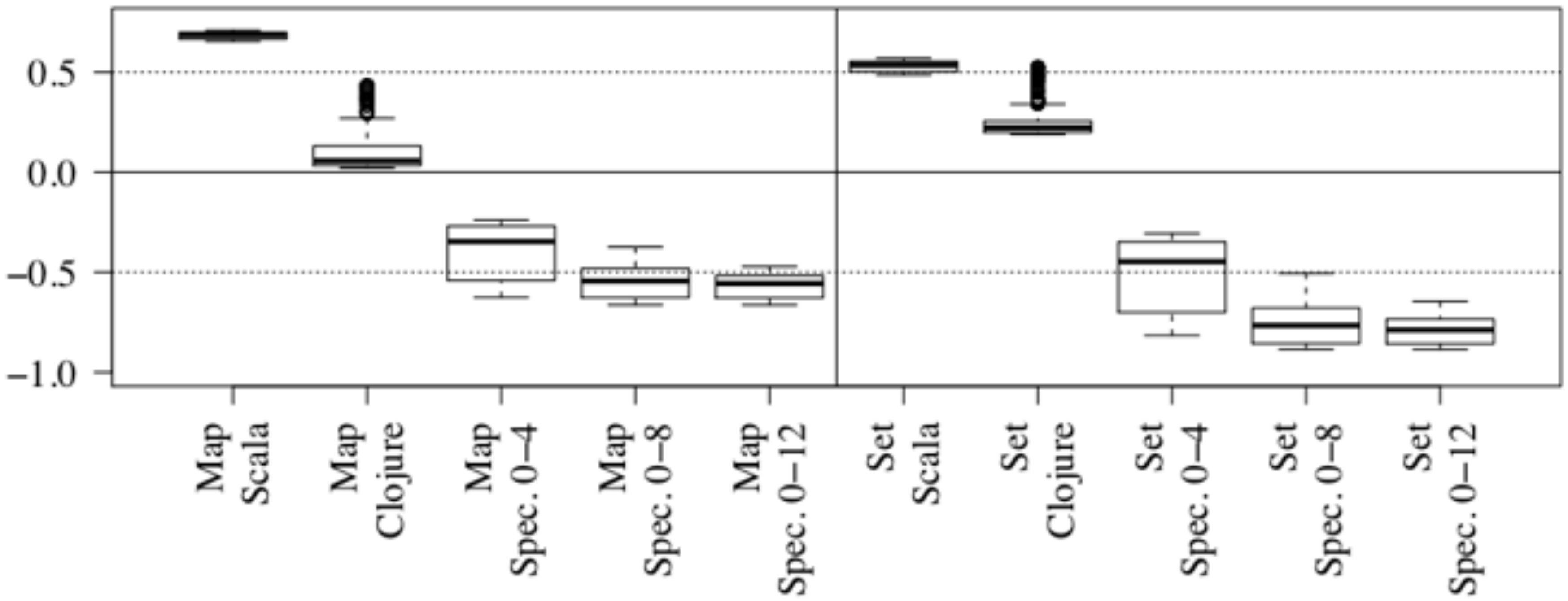


Figure 3. Relative footprints of 32-bit sets and maps compared against our generic implementation (i.e., the zero line).

a lot leaner

but not much slower

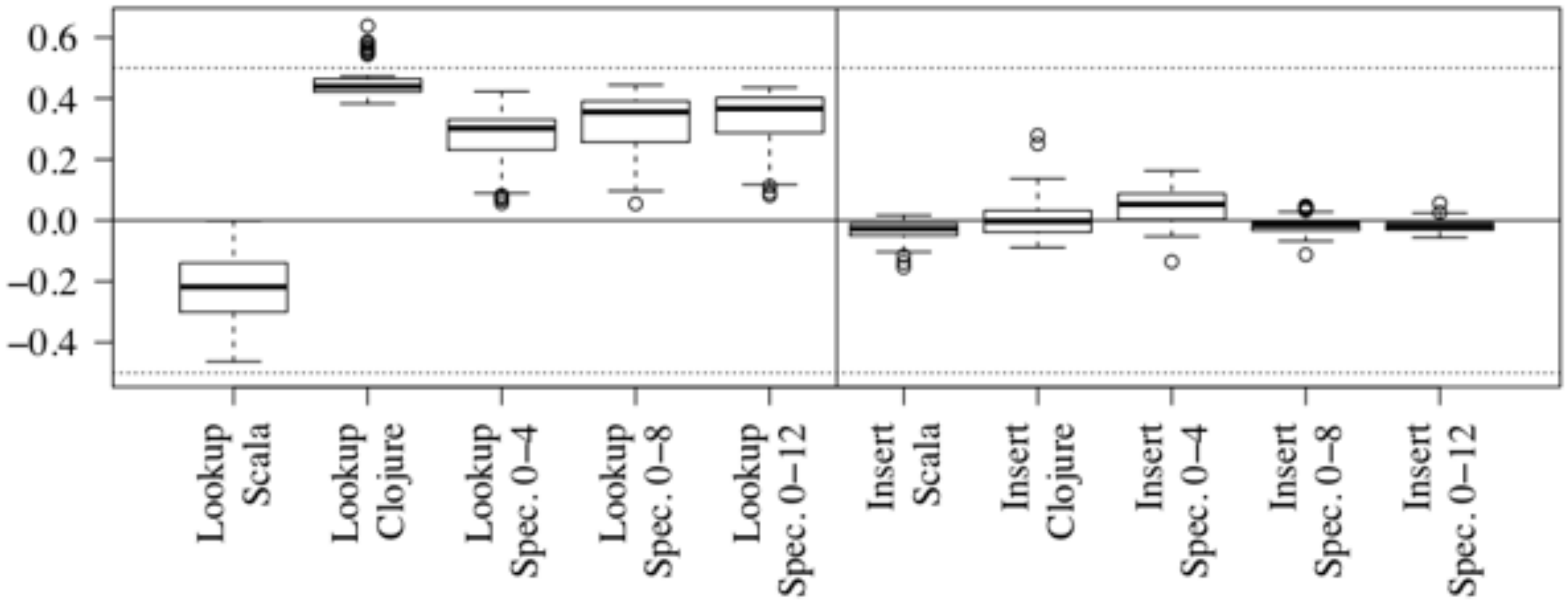


Figure 4. Relative run-times for `lookup` and `insert` in maps compared against our generic implementation (i.e., the zero line).

Summary

- Currently we get, compared to the state-of-the-art
  - 50%-100% speedups
  - 50%-80% memory savings
- Generated Java code
  - very low level, intrinsic complexity
  - many variants for features, few specializations for optimization
- Current work:
  - Experimental evaluation on real code
  - Integrating different optimizations
  - Squeezing more out of iteration
  - Squeezing more out of incrementality and staged immutability