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:
  - Built-in
  - 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)
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
```java
class TrieSet implements java.util.Set {
    TrieNode root;
    int size;

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

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 across 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).

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

**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

```java
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
- minimize code generation by having a fragile base class
Experiment

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

<table>
<thead>
<tr>
<th>Arity</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>1.44</td>
<td>63.14</td>
<td>14.26</td>
<td>3.27</td>
<td>1.24</td>
<td>0.94</td>
<td>0.93</td>
<td>0.96</td>
<td>1.00</td>
<td>1.05</td>
<td>1.11</td>
<td>1.17</td>
<td>1.23</td>
<td>1.28</td>
<td>1.32</td>
<td>1.33</td>
</tr>
<tr>
<td>Σ %</td>
<td>1.44</td>
<td>64.58</td>
<td>78.84</td>
<td>82.10</td>
<td>83.34</td>
<td>84.29</td>
<td>85.21</td>
<td>86.17</td>
<td>87.17</td>
<td>88.22</td>
<td>89.32</td>
<td>90.49</td>
<td>91.72</td>
<td>92.99</td>
<td>94.31</td>
<td>95.65</td>
</tr>
</tbody>
</table>

Random integers simulating good hash codes

Figure 2. Memory overhead per node arity in 32-bit mode.
Figure 3. Relative footprints of 32-bit sets and maps compared against our generic implementation (i.e., the zero line).

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