Faster persistent data structures through hashing

Johan Tibell
johan.tibell@gmail.com

2011-09-23
Motivating problem: Twitter data analysis

I'm computing a communication graph from Twitter data and then scan it daily to allocate social capital to nodes behaving in a good karmic manner. The graph is culled from 100 million tweets and has about 3 million nodes.

We need a data structure that is

- fast when used with string keys, and
- doesn't use too much memory.
Persistent maps in Haskell

- `Data.Map` is the most commonly used map type.
- It's implemented using size balanced trees and is representative of the performance of other binary tree implementations.
- Keys can be of any type, as long as values of the type can be ordered.
Real world performance of Data.Map

- Good in theory: no more than $O(\log n)$ comparisons.
- Not great in practice: up to $O(\log n)$ comparisons!
- Many common types are expensive to compare e.g. String, ByteString, and Text.
- Given a string of length $k$, we need $O(k \times \log n)$ comparisons to look up an entry.
Hash tables perform well with string keys: $O(k)$ amortized lookup time for strings of length $k$.

However, we want persistent maps, not mutable hash tables.
Milan Straka's idea: IntMaps as arrays

- We can use hashing without using hash tables!
- `Data.IntMap` implements a persistent array and is much faster than `Data.Map`.
- Use hashing to derive an `Int` from an arbitrary key.

```haskell
class Hashable a where
  hash :: a -> Int
```
Collisions are easy to deal with

- **IntMap** implements a sparse, persistent array of size $2^{32}$ (or $2^{64}$).
- Hashing using this many buckets makes collisions rare: for $2^{24}$ entries we expect about 32,000 single collisions.
- Implication: We can use any old collision handling strategy (e.g. chaining using linked lists).
HashMap implemented using an IntMap

Naive implementation:

```
newtype HashMap k v = HashMap (IntMap [(k, v)])
```

By inlining (``unpacking'') the list and pair constructors we can save 2 words of memory per key/value pair.
## Benchmark: Map vs HashMap

Keys: $2^{12}$ random 8-byte *ByteStrings*

<table>
<thead>
<tr>
<th></th>
<th>Runtime (μs)</th>
<th>Runtime % increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Map</td>
<td>HashMap</td>
</tr>
<tr>
<td>lookup</td>
<td>1956</td>
<td>916</td>
</tr>
<tr>
<td>insert</td>
<td>3543</td>
<td>1855</td>
</tr>
<tr>
<td>delete</td>
<td>3791</td>
<td>1838</td>
</tr>
</tbody>
</table>
Can we do better?

- Imperative hash tables still perform better, perhaps there's room for improvement.
- We still need to perform $O(\min(W, \log n))$ Int comparisons, where $W$ is the number of bits in a word.
- The memory overhead per key/value pair is still high, about 9 words per key/value pair.
Borrowing from our neighbours

- Clojure uses a *hash-array mapped trie* (HAMT) data structure to implement persistent maps.
- Described in the paper `Ideal Hash Trees` by Bagwell (2001).
- Originally a mutable data structure implemented in C++.
- Clojure's persistent version was created by Rich Hickey.
Hash-array mapped tries

- Shallow tree with high branching factor.
- Each node, except the leaf nodes, contains an array of up to 32 elements.
- 5 bits of the hash are used to index the array at each level.
- A clever trick, using bit population count, is used to represent sparse arrays.
The Haskell definition of a HAMT

```haskell
data HashMap k v
    = Empty
    | BitmapIndexed !Bitmap !(Array (HashMap k v))
    | Leaf !Hash !k v
    | Full !(Array (HashMap k v))
    | Collision !Hash !(Array (Leaf k v))

type Bitmap = Word
type Hash = Int
data Array a = Array (Array# a)
```
High performance Haskell programming

Optimized implementation using standard techniques:

- constructor unpacking,
- GHC's new `INLINABLE` pragma, and
- paying careful attention to strictness.

`insert` performance still bad (e.g. compare to hash tables).
Optimizing insertion

- Most time in `insert` is spent copying small arrays.
- Array copying is implemented in Haskell and GHC doesn't apply enough loop optimizations to make it run fast.
- When allocating arrays GHC fills the array with dummy elements, which are immediately overwritten.
Optimizing insertion: copy less

- Bagwell's original formulation used a fanout of 32.
- A fanout of 16 seems to provide a better trade-off between lookup and insert performance in our setting.
- Improved performance by 14%
Optimizing insertion: copy faster

- Daniel Peebles and I have implemented a set of new primops for copying arrays in GHC.
- The implementation generates straight-line code for copies of statically known small size, and uses a fast `memcpy` otherwise.
- Improved performance by 20%
In many cases maps are created in one go from a sequence of key/value pairs.

We can optimize for this case by repeatedly mutating the HAMT and freezing it when we're done.

Keys: $2^{12}$ random 8-byte `ByteStrings`

<table>
<thead>
<tr>
<th>Method</th>
<th>Runtime (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fromList/pure</td>
<td>100</td>
</tr>
<tr>
<td>fromList/mutating</td>
<td>50</td>
</tr>
</tbody>
</table>
Optimizing lookup: Faster population count

- Tried several bit population count implementations.
- Best speed/memory-use trade-off is a lookup table based approach.
- Using the `POPCNT` SSE 4.2 instructions improves the performance of `lookup` by 12%.
Benchmark: IntMap-based vs HAMT

Keys: $2^{12}$ random 8-byte **ByteStrings**

<table>
<thead>
<tr>
<th></th>
<th>Runtime (μs)</th>
<th>Runetime % increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IntMap</td>
<td>HAMT</td>
</tr>
<tr>
<td>lookup</td>
<td>916</td>
<td>477</td>
</tr>
<tr>
<td>insert</td>
<td>1855</td>
<td>1998</td>
</tr>
<tr>
<td>delete</td>
<td>1838</td>
<td>2303</td>
</tr>
</tbody>
</table>

The benchmarks don't include the **POPCNT** optimization, due to it not being available on many architectures.
Memory usage: IntMap-based

Total: 96 MB, tree: 66MB ($2^{20}$ Int entries)
Memory usage: HAMT

Total: 71MB, tree: 41MB ($2^{20}$ Int entries)
### Summary

Keys: $2^{12}$ random 8-byte **ByteStrings**

<table>
<thead>
<tr>
<th></th>
<th>Runtime (µs)</th>
<th>Runtime % increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Map</td>
<td>HAMT</td>
</tr>
<tr>
<td>lookup</td>
<td>1956</td>
<td>477</td>
</tr>
<tr>
<td>insert</td>
<td>3543</td>
<td>1998</td>
</tr>
<tr>
<td>delete</td>
<td>3791</td>
<td>2303</td>
</tr>
</tbody>
</table>