Performance

From HaskellWiki

Haskell Performance Resource

Constructs:
Data Types - Functions
Overloading - FFI - Arrays
Strings - Integers - I/O
Floating point - Concurrency
Modules - Monads

Techniques:
Strictness - Laziness
Avoiding space leaks
Accumulating parameter

Implementation-Specific:
GHC - nhc98 - Hugs
Yhc - JHC

Welcome to the **Haskell Performance Resource**, the collected wisdom on how to make your Haskell programs go faster.

Contents

1 Introduction

One question that often comes up is along the general lines of "Can I write this program in Haskell so that it performs as well as, or better than, the same program written in some other language?"

This is a difficult question to answer in general because Haskell is a language, not an implementation. Performance can only be measured relative to a specific language implementation.

Moreover, it's often not clear if two programs which supposedly have the same functionality really do the same thing. Different languages sometimes require very different ways of expressing the same intent. Certain types of bug are rare in typical Haskell programs that are more common in other languages and vice versa, due to strong typing, automatic memory management and lazy evaluation.

Nonetheless, it is usually possible to write a Haskell program that performs as well as, or better than, the same program written in any other language. The main caveat is that you may have to modify your code significantly in order to improve its performance. Compilers such as GHC are good at eliminating layers of abstraction, but they aren't perfect, and often need some help.

There are many non-invasive techniques: compiler options, for example. Then there are techniques that require adding some small amounts of performance cruft to your program: strictness annotations, for example. If you still don't get the best performance, though, it might be necessary to resort to larger refactorings.

Sometimes the code tweaks required to get the best performance are non-portable, perhaps because they require language extensions that aren't implemented in all compilers (e.g. unboxing), or because they require using platform-specific features or libraries. This might not be acceptable in your setting.

If the worst comes to the worst, you can always write your critical code in C and use the FFI to call it. Beware of the boundaries though - marshaling data across the FFI can be expensive, and multi-language memory management can be complex and error-prone. It's usually better to stick to Haskell if possible.

2 Basic techniques

The key tool to use in making your Haskell program run faster is *profiling*. Profiling is provided by GHC and nhc98. There is *no substitute* for finding where your program's time/space is *really* going, as opposed to where you imagine it is going.

Another point to bear in mind: By far the best way to improve a program's performance *dramatically* is to use better algorithms. Once profiling has thrown the spotlight on the guilty time-consumer(s), it may be better to re-think your program than to try all the tweaks listed below.

Another extremely efficient way to make your program snappy is to use library code that has been Seriously Tuned By Someone Else. You *might* be able to write a better sorting function than the one in `Data.List`, but it will take you much longer than typing `import Data.List`.

We have chosen to organise the rest of this resource first by Haskell construct (data types, pattern matching, integers), and then within each category to describe techniques that apply across implementations, and also techniques that are specific to a certain Haskell implementation (e.g. GHC). There are some implementation-specific techniques that apply in general - those are linked from the General Implementation-Specific Techniques section below.

3 Haskell constructs

- Data Types
- Functions
- Overloading
- FFI
- Arrays
- Strings
- Integers
- I/O
- Floating Point
- Concurrency
- Modules
- Monads

4 General techniques

The most important thing for a beginner to keep in mind is that Haskell programs are evaluated using lazy evaluation, which has many advantages, but also drawbacks when it comes to performance. The rule of thumb is that if you really want to have explicit control over the evaluation, then you should try to avoid it.

- Strictness
- Laziness
- Avoiding space leaks
- Accumulating parameters
- Avoiding stack overflow

5 Benchmarking libraries

The Criterion library is used often to generate detailed benchmark results.

- Introductory blog post (http://www.serpentine.com/blog/2009/09/29/criterion-a-new-benchmarking-library-for-haskell/)
- Simple example of using Criterion (http://www.stackbuilders.com/news/obverse-versus-reverse-benchmarking-in-haskell-with-criterion)

6 Compiler specific techniques

- GHC
- nhc98
- Hugs
- Yhc
- JHC

7 More information

- Haskell as fast as C: working at a high altitude for low level performance (https://donsbot.wordpress.com/2008/06/04/haskell-as-fast-as-c-working-at-a-high-altitude-for-low-level-performance/)

- High-Performance Haskell (http://www.randomhacks.net/2007/01/22/high-performance-haskell/) (Slide deck (http://www.slideshare.net/tibbe/highperformance-haskell))

- Haskell as fast as C: A case study (http://lambda.jstolarek.com/2013/04/haskell-as-fast-as-c-a-case-study/)

- Dan Doel - Introduction to Low Level Haskell Optimization (https://www.youtube.com/watch?v=McFNkLPTOSY&feature=youtu.be) ; a video of Dan Doel's talk at the Boston Haskell Meetup, Sept 17, 2014 (slides (http://hub.darcs.net/dolio/optimization-talk-demo/browse/slides/slides.md)).

- Don Stewart's Haskell performance overview (http://stackoverflow.com/questions/3276240/tools-for-analyzing-performance-of-a-haskell-program/3276557#3276557) on StackOverflow (2013)

- There are plenty of good examples of Haskell code written for performance in the The Computer Language Benchmarks Game (http://benchmarksgame.alioth.debian.org/)

- And many alternatives, with discussion, on this wiki: Benchmarks Game and on the old Haskell wiki (http://web.archive.org/web/20060209215702/http://haskell.org/hawiki/ShootoutEntry) (in the Web Archive)

- There are ~100 slides on High-Performance Haskell (http://blog.johantibell.com/2010/09/slides-from-my-high-performance-haskell.html) from the 2010 CUFP tutorial on that topic.

8 Specific comparisons of data structures

8.1 Ropes

For modifying very long strings, a rope data structure (https://en.wikipedia.org/wiki/Rope_(data_structure)) is very efficient. The rope (https://hackage.haskell.org/package/rope) package provides these.

8.2 Data.Sequence vs. lists

Data.Sequence has complexity O(log(min(i,n-i))) for access, insertion and update to position i of a sequence of length n.

List has complexity O(i).

List is a non-trivial constant-factor faster for operations at the head (cons and head), making it a more efficient choice for stack-like and stream-like access patterns. Data.Sequence is faster for every other access pattern, such as queue and random access.

See the following program for proof:

```haskell
import Data.Sequence

insert_million 0 sequence = sequence
insert_million n sequence = insert_million (n - 1)(sequence |> n)

main = print (Data.Sequence.length (insert_million 1000000 empty))
```

```
 $ ghc -O2 --make InsertMillionElements.hs && time ./InsertMillionElements +RTS -K100M
1000000
real 0m7.238s
user 0m6.804s
sys 0m0.228s
```

```haskell
insert_million 0 list = reverse list
insert_million n list = insert_million (n -1) (n:list)

main = print (length (insert_million 1000000 []))
```

```
 $ ghc -O2 --make InsertMillionElements.hs && time ./InsertMillionElementsList +RTS -K100M
1000000
real 0m0.588s
user 0m0.528s
sys 0m0.052s
```

Lists are substantially faster on this micro-benchmark.

A sequence uses between 5/6 and 4/3 times as much space as the equivalent list (assuming an overhead of one word per node, as in GHC). If only deque operations are used, the space usage will be near the lower end of the range, because all internal nodes will be ternary. Heavy use of split and append will result in sequences using approximately the same space as lists. In detail:

- a list of length *n* consists of *n* cons nodes, each occupying 3 words.
- a sequence of length *n* has approximately *n*/(*k*-1) nodes, where *k* is the average arity of the internal nodes (each 2 or 3). There is a pointer, a size and overhead for each node, plus a pointer for each element, i.e. *n*(3/(*k*-1) + 1) words.

9 Additional Tips

- Use strict returns (return $! ...) unless you absolutely need them lazy.
- Profile, profile, profile - understand who is allocated the memory on your heap (+RTS -hc) and how it's being used (+RTS -hb).
- Use +RTS -p to understand who's doing all the allocations and where your time is being spent.
- Approach profiling like a science experiment - make one change, observe if anything is different, rollback and make another change - observe the change. Keep notes!
- Use ThreadScope to visualize GHC eventlog traces.

Retrieved from "https://wiki.haskell.org/index.php?title=Performance&oldid=59952"
Categories:

- Idioms
- Language
- Performance
