The GHC Runtime System

Edward Z. Yang
Last time

Haskell → Core → STG → C-- → Assembly
Today

Haskell

Core

STG

C--

Assembly + RTS

Runnable executable
Why learn about the RTS?
In investigating some weird benchmarking results in a library, I stumbled upon some behavior I don't understand, though it might be really obvious. It seems that the time taken for many operations (creating a new `MutableArray`, reading or modifying an `IORef`) increases in proportion to the number of arrays in memory.

Here's the first example:

```haskell
module Main
  where

import Control.Monad
import qualified Data.Primitive as P
import Control.Concurrent
import Data.IORef
import Criterion.Main
import Control.Monad.Primitive(PrimState)
```
Computer Programming:  Edit

Why are Haskell 'green threads' more efficient/performant than native threads?  Edit

Related to this paper: Page on Yale  (Mio: A High-Performance Multicore IO Manager for GHC)

Specifically quoting the introduction:

A naive implementation, using one native thread (i.e. OS thread) per request would lead to the use of a large number of native threads, which would substantially degrade performance due to the relatively high cost of OS context switches [22]. In contrast, Haskell threads are lightweight threads, which can be context switched without incurring an OS context switch and with much lower overhead.

I've heard the anecdote that Ruby threading was so slow because Ruby used "green threads" instead of native threads e.g. like Java. So what makes Haskell "green threads" different from Ruby "green threads"?
In a nutshell...

- Storage Manager (Garbage Collection)
- Scheduler
- Bytecode Interpreter (GHCi)
- Dynamic Linker
- Software Transactional Memory
- Profiling

[and more...]
In a nutshell...

→ Storage Manager (Garbage Collection)

→ Scheduler

→ Bytecode Interpreter (GHCi)

→ Dynamic Linker

→ Software Transactional Memory

→ Profiling

[and more...]
→ **Storage Manager**
  - Generational Copying GC
  - Write barriers & promotion
  - Parallel GC (briefly)

→ **Scheduler**
  - Threads
  - HECs
  - Load balancing
  - Bound threads
  - MVars
Garbage Collection
Garbage Collection: Brief Review

Reference Counting
- Can’t handle cycles
- PHP, Perl, Python*

Mark and Sweep
- Fragmentation
- Needs to sweep entire heap
- GoLang, Ruby
Generational Copying Collector

JVM, v8, GHC

“Most objects die young”

—the Generational Hypothesis
Generational Copying Collector

JVM, v8, GHC

“Most objects die young”
especially in functional languages!

—the Generational Hypothesis
root set

A B C

from space

Scavenge pointer

to space

EVACUATING
Evacuating

From space:

Scavenge pointer

To space:
SCAVENGING

from space

Scavenge pointer

to space
EVACUATING

from space

Scavenge pointer

to space
SCAVENGING

from space

Scavenge pointer

to space
EVACUATING

from space

Scavenge pointer

to space
SCAVENGING

from space

Scavenge pointer

to space
to space from
Evacuate → Scavenge
Nursery

Generation 1

Minor GC
Generational Copying Collector

- The more garbage you have, the faster it runs
- Free memory is contiguous
mk_exit()
  entry:
    Hp = Hp + 16;
    if (Hp > HpLim) goto gc;

    v::I64 = I64[R1] + 1;

    I64[Hp - 8] = GHC_Types_I_con_info;
    I64[Hp + 0] = v::I64;

    R1 = Hp;
    Sp = Sp + 8;
    jump (I64[Sp + 0]) ();

    gc: HpAlloc = 16;
    jump stg_gc_enter_1 ();
What about Purity?

→ Write Barriers

→ Parallel Garbage Collection
Minor GC
Why is generational GC hard?
Why is generational GC hard in Java?

This.
Purity to the rescue

- Mutation is rare
- IORefs are slow anyway
- Laziness is a special kind of mutation
Nursery

Generation 1 thunk
Parallel GC

Idea: Split heap into blocks, and parallelize the scavenging process.
Needs synchronization
If A is immutable...

...observationally indistinguishable!
Purity = Flexibility
In investigating some weird benchmarking results in a library, I stumbled upon some behavior I don't understand, though it might be really obvious. It seems that the time taken for many operations (creating a new `MutableArray`, reading or modifying an `IORef`) increases in proportion to the number of arrays in memory.

Here's the first example:

```haskell
module Main
  where

import Control.Monad
import qualified Data.Primitive as P
import Control.Concurrent
import Data.IORef
import Criterion.Main
import Control.Monad.Primitive(PrimState)
```
Scheduler
Haskell threads

• Haskell implements user-level threads in `Control.Concurrent`
  - Threads are lightweight (in both time and space)
  - Use threads where in other languages would use cheaper constructs
  - Runtime emulates blocking OS calls in terms of non-blocking ones
  - Thread-switch can happen any time GC could be invoked

• `forkIO` call creates a new thread:

```haskell
forkIO :: IO () -> IO ThreadId -- creates a new thread
```

• A few other very useful thread functions:

```haskell
throwTo :: Exception e => ThreadId -> e -> IO ()
killThread :: ThreadId -> IO () -- = flip throwTo ThreadKilled
threadDelay :: Int -> IO () -- sleeps for # of μsec
myThreadId :: IO ThreadId
```
mk_exit()
entry:
   Hp = Hp + 16;
   if (Hp > HpLim) goto gc;

   v::I64 = I64[R1] + 1;

   I64[Hp - 8] = GHC_Types_I_con_info;
   I64[Hp + 0] = v::I64;

   R1 = Hp;
   Sp = Sp + 8;
   jump (I64[Sp + 0]) ();

   gc: HpAlloc = 16;
   jump stg_gc_enter_1 ();
}
Anatomy of a thread

(heap allocated)
Single-threaded operation

TSO

Thread queue

StgRun

GC

Y

heap overflow?

N

timer

Scheduler Loop

root set!
Multi-threaded operation

Scheduler loops (HEC)

OS threads

-N3
HECs are locks
GC takes all locks
Work imbalance
Work imbalance
Work imbalance
Work imbalanced

Throughput First!
Bound threads
Bound threads

Diagram:
- A flowchart with a decision point labeled 'TSo'
- A blue square with 'A'
- A green square with 'A'
- A lock symbol pointing to the green square
Bound threads
Bound threads
MVars

Stg_MVAR_*.info
- head
- tail
- value

TSO_QUEUE
- tso
- link

TSO

(value)
MVars

Fun fact: If the MVar becomes garbage, the threads in its queue die too.
Scheduler in a nutshell

Everything lives on the heap

Small initial stack segments
  = cheap green threads

Purity = most code threadsafe by default
GHC Commentary: The Runtime System

GHC's runtime system is a slightly scary beast: 50,000 lines of C and C++ seems at first glance to be completely obscure. What on earth does the highlights:

- It includes all the bits required to execute Haskell code that aren't itself. For example, the RTS contains the code that knows how to recall `error`, code to allocate `Array#` objects, and code to implement

- It includes a sophisticated storage manager, including a multi-generational copying and compacting strategies.

- It includes a user-space scheduler for Haskell threads, together with Haskell threads across multiple CPUs, and allowing Haskell threads separate OS threads.

- There's a byte-code interpreter for GHCi, and a dynamic linker for a GHCi session.

- Heap-profiling (of various kinds), time-profiling and code coverage included.
Related Work


