Rewriting Haskell Strings

Duncan Coutts Don Stewart Roman Leshchinskiy

Programming Tools Group
Oxford University

Programming Languages and Systems
University of New South Wales

SAPLING 2006

IO IN HASKELL

Lazy Haskell IO can be beautiful:

$$return \cdot foldl' \; k \; 5381 \cdot map \; toLower \cdot filter \; isAlpha \; =\!\ll \; readFile \; f$$
$$\textbf{where} \; k \; h \; c \; = \; h \; * \; 33 \; + \; ord \; c$$

IO IN HASKELL

Lazy Haskell IO can be beautiful:

$$return \cdot foldl'\ k\ 5381 \cdot map\ toLower \cdot filter\ isAlpha\ =\ll\ readFile\ f$$
$$\textbf{where}\ k\ h\ c\ =\ h\ *\ 33\ +\ ord\ c$$

But naive code can be slow:

```
$ time ./a.out
bf805325
./a.out 23.04s total
```

AND ENTIRELY UNLIKE C

An equivalent naive C implementation:

```
int c;
long h = 5381;
FILE *fp = fopen(f, "r");
while ((c = fgetc(fp)) != EOF)
  if (isalpha(c))
     h = h * 33 + tolower(c);
fclose(fp);
return h;
```

AND ENTIRELY UNLIKE C

An equivalent naive C implementation:

```
int c;
long h = 5381;
FILE *fp = fopen(f, "r");
while ((c = fgetc(fp)) != EOF)
  if (isalpha(c))
     h = h * 33 + tolower(c);
fclose(fp);
return h;
```

```
$ time ./a.out
bf805325
./a.out 3.93s total
```

LET'S FIX IT!

Use ByteStrings with stream fusion

```
import Data.ByteString.Lazy
return . foldl' k 5381 . map toLower . filter isAlpha =<< readFile f
  where k h c = h * 33 + ord c
```

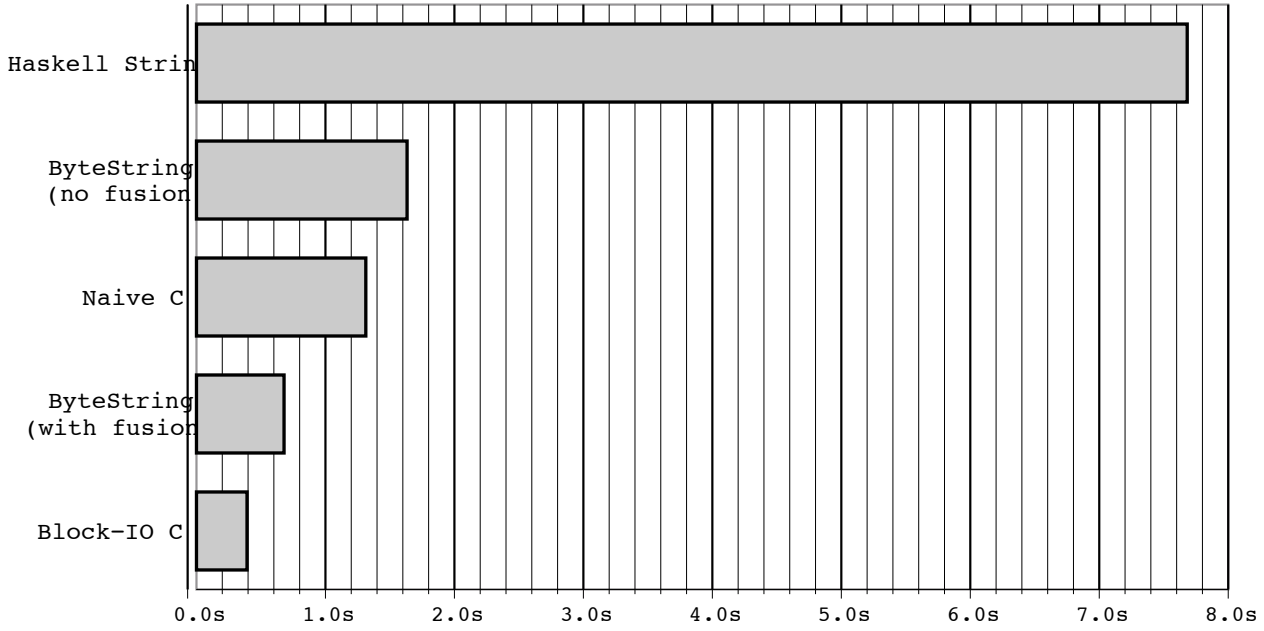
LET'S FIX IT!

Use ByteStrings with stream fusion

```
import Data.ByteString.Lazy
return . foldl' k 5381 . map toLower . filter isAlpha =<< readFile f
  where k h c = h * 33 + ord c
```

```
$ time ./a.out
bf805325
./a.out  2.04s total
```

COMPARING STRINGS, BYTESTRINGS AND C



You have to rewrite the C version to be block oriented to win.

Representing Strings

The Lovely, Delicate [Char]

```
data [] a = [] | a : [a]
type String = [Char]
```

- ➜ Strings are represented as linked lists of pointers to chars.
- ➜ Every node requires an indirection, and another to retrieve character
- ➜ Bad for the cache, bad for gcc, bad for performance.

Representing Strings

The Lovely, Delicate [Char]

```
data [] a = [] | a : [a]
type String = [Char]
```

- ➜ Strings are represented as linked lists of pointers to chars.
- ➜ Every node requires an indirection, and another to retrieve character
- ➜ Bad for the cache, bad for gcc, bad for performance.

The Ruthless, Cut-throat ByteString

```
data ByteString = BS !(ForeignPtr Word8) !Int !Int
```

Listen to your hardware!

Changing the type is a big win, but we can play better games yet ...

BOTTLENECK: TEMPORARY DATA STRUCTURES

```
print . minimum . filter (‘isPrefixOf‘ x) . map toUpper
```

BOTTLENECK: TEMPORARY DATA STRUCTURES

```
print . minimum . filter (‘isPrefixOf‘ x) . map toUpper
```

And this is compiled to:

```
case map toUpper s                 of s’   ->
case filter (‘isPrefixOf‘ x) s’ of s’’  ->
case minimum s’’                   of s’’’ -> print s’’’
```

Lots of temporary arrays are being allocated and discarded!

THE BIG IDEA: DEFORESTATION AND FUSION

Ideally, we'd like to make just one pass over the input data

We'd write:

```
map f . map g
```

THE BIG IDEA: DEFORESTATION AND FUSION

Ideally, we'd like to make just one pass over the input data

We'd write:

```
map f . map g
```

and the compiler would emit:

```
map (f . g)
```

THE BIG IDEA: DEFORESTATION AND FUSION

Ideally, we'd like to make just one pass over the input data

We'd write:

```
map f . map g
```

and the compiler would emit:

```
map (f . g)
```

We can teach the compiler how to do this:

$$\langle \mathbf{map/map\ fusion} \rangle \quad map\ f \cdot map\ g \mapsto map\ (f \cdot g)$$

FUSION SYSTEMS

A variety of general purpose fusion systems exist:

$\langle \mathbf{foldr/build\ fusion} \rangle \ \forall\ g\ k\ z\ .$
$\quad \mathit{foldr}\ k\ z\ (\mathit{build}\ g)\ \mapsto\ g\ k\ z$

FUSION SYSTEMS

A variety of general purpose fusion systems exist:

$\langle$**foldr/build fusion**$\rangle$ $\forall\ g\ k\ z\ .$
foldr $k\ z$ (*build* g) $\mapsto\ g\ k\ z$

$\langle$**destroy/unfoldr fusion**$\rangle$ $\forall\ g\ f\ e\ .$
destroy g (*unfoldr* $f\ e$) $\mapsto\ g\ f\ e$

FUSION SYSTEMS

A variety of general purpose fusion systems exist:

$\langle$**foldr/build fusion**$\rangle$ $\forall\ g\ k\ z\ .$
foldr $k\ z$ (*build* g) $\mapsto\ g\ k\ z$

$\langle$**destroy/unfoldr fusion**$\rangle$ $\forall\ g\ f\ e\ .$
destroy g (*unfoldr* $f\ e$) $\mapsto\ g\ f\ e$

$\langle$**function array fusion**$\rangle$ $\forall\ f\ g\ s\ t\ .$
loop $f\ s \cdot$ *fst* $\cdot$ *loop* $g\ t\ \mapsto$ *loop* (*fuse* $f\ g$) (s, t)

FUSION SYSTEMS

A variety of general purpose fusion systems exist:

$\langle \textbf{foldr/build fusion} \rangle \; \forall \; g \; k \; z \; .$
$\quad \textit{foldr} \; k \; z \; (\textit{build} \; g) \; \mapsto \; g \; k \; z$

$\langle \textbf{destroy/unfoldr fusion} \rangle \; \forall \; g \; f \; e \; .$
$\quad \textit{destroy} \; g \; (\textit{unfoldr} \; f \; e) \; \mapsto \; g \; f \; e$

$\langle \textbf{function array fusion} \rangle \; \forall \; f \; g \; s \; t \; .$
$\quad \textit{loop} \; f \; s \cdot \textit{fst} \; \cdot \textit{loop} \; g \; t \; \mapsto \; \textit{loop} \; (\textit{fuse} \; f \; g) \; (s, t)$

But none support all of: a wide range of operations, flat non-inductive structures, *and* compile to fast code in GHC.

STREAM FUSION

To support *array* access patterns in $\mathcal{O}(1)$, we need to factor out the three phases:

- read the array into a stream of elements
- process the stream elements
- write the resulting stream into a new array

The Stream Type

We need an efficient stream abstraction:

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream} = \exists s.\ \mathit{Stream}\ \ (s \to \mathit{Step}\ s)\ \ s\ \ \mathit{Int}\\
&\textbf{data}\ \mathit{Step}\ s\ \ = \mathit{Done}\\
&\qquad\qquad\quad \mid\ \mathit{Yield}\ \ \mathit{Word8}\ \ s\\
&\qquad\qquad\quad \mid\ \mathit{Skip}\ \ s
\end{aligned}$$

An existentially wrapped seed, and stepper function, generating one of 3 results for each element of the stream:

- ➜ Done Finished processing the stream
- ➜ Yield Produce a transformed element
- ➜ Skip Filter this element out

Also carry a hint about the resulting size, to guide reallocations.

Building a stream from an array

Arrays to Streams:

$$
\begin{aligned}
&read \;\; :: \; ByteString \rightarrow Stream \\
&read \; s = \; Stream \; next \; 0 \; n \\
&\quad \textbf{where} \\
&\qquad n \qquad\qquad\qquad\quad = \; length \; s \\
&\qquad next \; i \mid i \; < \; n \quad = \; Yield \; (index \; s \; i) \; (i + 1) \\
&\qquad\qquad\quad \mid otherwise = \; Done
\end{aligned}
$$

Writing streams back out to array is also fairly straight forward.

REMOVING INTERMEDIATE STREAMS

Pairs of read and write are just the identity function on streams, so they can be removed, yielding our fusion rule:

$$\langle \mathbf{read/write\ fusion} \rangle \quad \mathit{read} \cdot \mathit{write} \quad \mapsto \quad id$$

- ➜ Whenever we see these pairs in the user's code, it is safe for the compiler to remove them.
- ➜ The library author supplies rewrite rules specific to the library, extending the compiler's optimisation range.

STREAM TRANSFORMERS

What about actually transforming the data?

Build a stream, transform it, and write it back:

$$map :: (Word8 \to Word8) \to ByteString \to ByteString$$
$$map\ f = write \cdot mapS\ f \cdot read$$

STREAM TRANSFORMERS

What about actually transforming the data?

Build a stream, transform it, and write it back:

$$
\begin{aligned}
&map \;::\; (Word8 \to Word8) \to ByteString \to ByteString \\
&map\, f \;=\; write \;\cdot\; mapS\, f \;\cdot\; read
\end{aligned}
$$

And applying *f* to each element:

$$
\begin{aligned}
&mapS \;::\; (Word8 \to Word8) \to Stream \to Stream \\
&mapS\, f\, (Stream\; next\; s\; n) \;=\; Stream\; next'\; s\; n \\
&\quad \textbf{where} \\
&\qquad next'\; s = \; \textbf{case}\; next\; s\; \textbf{of} \\
&\qquad\qquad \begin{aligned}
Done \quad\;\; &\to Done \\
Yield\; x\; s' &\to Yield\; (f\; x)\; s' \\
Skip\; s' \quad\;\; &\to Skip\; s'
\end{aligned}
\end{aligned}
$$

FUSING MAP

$$map\ f\ \cdot\ map\ g$$

FUSING MAP

$$map\ f\ \cdot\ map\ g$$

$$= write\ \cdot\ mapS\ f\ \cdot\ read\ \cdot \quad \{\text{inline } \textit{map}\ \times 2\}$$
$$\quad write\ \cdot\ mapS\ g\ \cdot\ read$$

FUSING MAP

$$map\ f\ \cdot\ map\ g$$

$$= write\ \cdot\ mapS\ f\ \cdot\ read\ \cdot \quad \{\text{inline } \textit{map}\ \times 2\}$$
$$\quad write\ \cdot\ mapS\ g\ \cdot\ read$$

$$= write\ \cdot\ mapS\ f\ \cdot\ mapS\ g\ \cdot\ read \quad \{read/write \text{ fusion}\}$$

FUSING MAP

$$map\ f\ \cdot\ map\ g$$

$$= write\ \cdot\ mapS\ f\ \cdot\ read\ \cdot \quad \{\text{inline } \textit{map}\ \times 2\}$$
$$\quad write\ \cdot\ mapS\ g\ \cdot\ read$$

$$= write\ \cdot\ mapS\ f\ \cdot\ mapS\ g\ \cdot\ read \quad \{read/write \text{ fusion}\}$$

$$= write\ \cdot\ mapS\ (f\ \cdot\ g)\ \cdot\ read \quad \{map/map \text{ fusion}\}$$

FILTER

$$
\begin{aligned}
&\mathit{filterS} :: (\mathit{Word8} \to \mathit{Bool}) \to \mathit{Stream} \to \mathit{Stream} \\
&\mathit{filterS}\ p\ (\mathit{Stream}\ \mathit{next}\ s\ n) = \mathit{Stream}\ \mathit{next}'\ s\ n \\
&\quad \textbf{where} \\
&\qquad \mathit{next}'\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad \begin{array}{lll}
\mathit{Done} & & \to\ \mathit{Done} \\
\mathit{Yield}\ x\ s' & \mid p\ x & \to\ \mathit{Yield}\ x\ s' \\
& \mid \mathit{otherwise} & \to\ \mathit{Skip}\ s' \\
\mathit{Skip}\ s' & & \to\ \mathit{Skip}\ s'
\end{array}
\end{aligned}
$$

FOLD

$$
\begin{aligned}
&\mathit{foldlS'} :: (a \to \mathit{Word8} \to a) \to a \to \mathit{Stream} \to a \\
&\mathit{foldlS'}\ f\ z\ (\mathit{Stream}\ \mathit{next}\ s\ n) = \mathit{loop}\ z\ s \\
&\quad \textbf{where} \\
&\qquad \mathit{loop}\ z\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad \begin{aligned}
\mathit{Done} &\to z \\
\mathit{Yield}\ x\ s' &\to \mathit{loop}\ (f\ z\ x)\ s' \\
\mathit{Skip}\ s' &\to \mathit{loop}\ z\ s'
\end{aligned}
\end{aligned}
$$

FIND/SHORT-CIRCUITING

$$
\begin{aligned}
&\mathit{findS} :: (\mathit{Word8} \to \mathit{Bool}) \to \mathit{Stream} \to \mathit{Maybe}\ \mathit{Word8} \\
&\mathit{findS}\ p\ (\mathit{Stream}\ \mathit{next}\ s\ n) = \mathit{loop}\ s \\
&\quad \textbf{where} \\
&\qquad \mathit{loop}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\quad \begin{array}{lll}
\mathit{Done} & & \to\ \mathit{Nothing} \\
\mathit{Yield}\ x\ s' & \mid p\ x & \to\ \mathit{Just}\ x \\
& \mid \mathit{otherwise} & \to\ \mathit{loop}\ s' \\
\mathit{Skip}\ s' & & \to\ \mathit{loop}\ s'
\end{array}
\end{aligned}
$$

FUSING WITH STREAMS

$$foldl'\ f\ z\ \cdot\ map\ g\ \cdot\ filter\ h$$

FUSING WITH STREAMS

$$foldl'\ f\ z\ \cdot\ map\ g\ \cdot\ filter\ h$$

$$\begin{aligned} =\ & foldlS'\ f\ z\ \cdot\ read\ \cdot\ write\ \cdot\ mapS\ g \\ & \cdot\ read\ \cdot\ write\ \cdot\ filterS\ h\ \cdot\ read \end{aligned} \qquad \begin{aligned} & \{\text{inline } foldl', map \\ & \text{and } filter\} \end{aligned}$$

FUSING WITH STREAMS

$$foldl'\ f\ z\ \cdot\ map\ g\ \cdot\ filter\ h$$

$$= foldlS'\ f\ z\ \cdot\ read\ \cdot\ write\ \cdot\ mapS\ g \qquad \{\text{inline } foldl', map$$
$$\cdot\ read\ \cdot\ write\ \cdot\ filterS\ h\ \cdot\ read \qquad \text{and } filter\}$$

$$= foldlS'\ f\ z\ \cdot\ mapS\ g\ \cdot\ filterS\ h\ \cdot\ read \quad \{read/write \text{ fusion}\}$$

FUSING WITH STREAMS

$$foldl'\ f\ z\ \cdot\ map\ g\ \cdot\ filter\ h$$

$$= foldlS'\ f\ z\ \cdot\ read\ \cdot\ write\ \cdot\ mapS\ g \qquad \{\text{inline } foldl', map$$
$$\cdot\ read\ \cdot\ write\ \cdot\ filterS\ h\ \cdot\ read \qquad \text{and } filter\}$$

$$= foldlS'\ f\ z\ \cdot\ mapS\ g\ \cdot\ filterS\ h\ \cdot\ read \quad \{read/write \text{ fusion}\}$$

- ➜ The original 3 loops, and 2 intermediate arrays, are *automatically* transformed into a single traversal.
- ➜ GHC then further inlines and combines transformers, eliminating $Step$ values.
- ➜ $2.4\times$ faster

EXTENSIONS

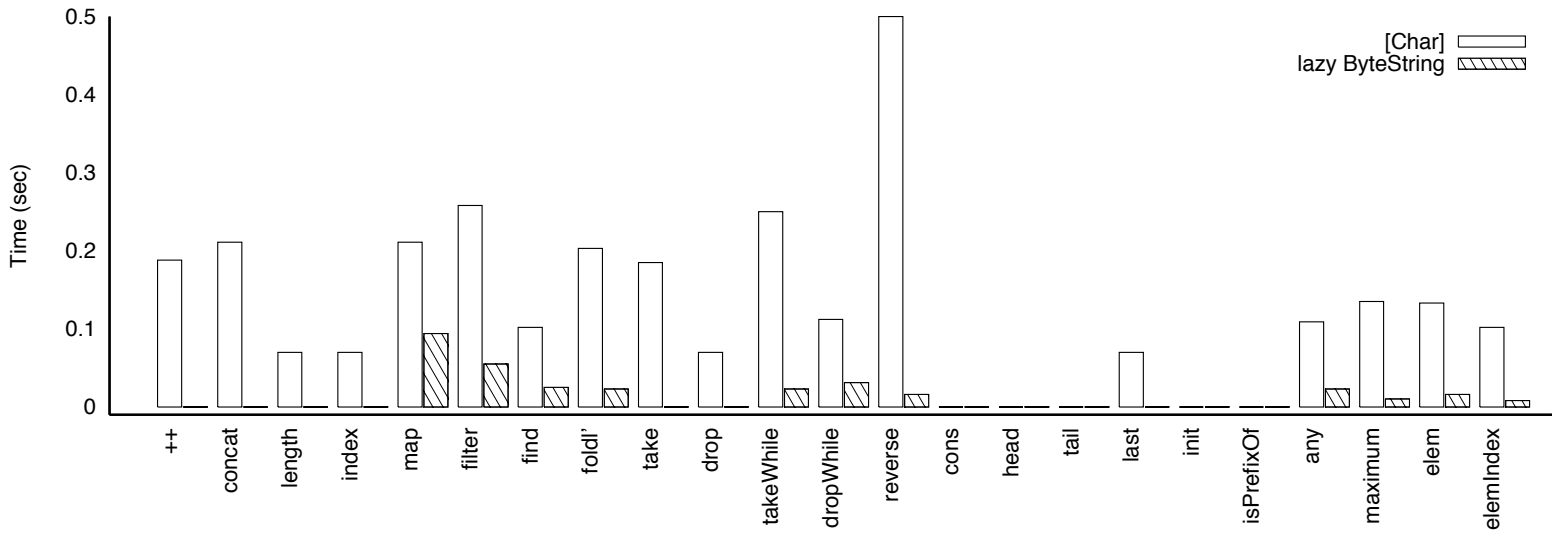
Over 40 functions in the ByteString library fuse, including:

- ➜ both up and down traversals ($foldl$, $foldr$, $scanl$, $scanr$)
- ➜ up and down loops will fuse with "bidirectional" loops like map
- ➜ *lazy* bytestrings, combining strict array chunks in the L2 cache, with a lazy spine

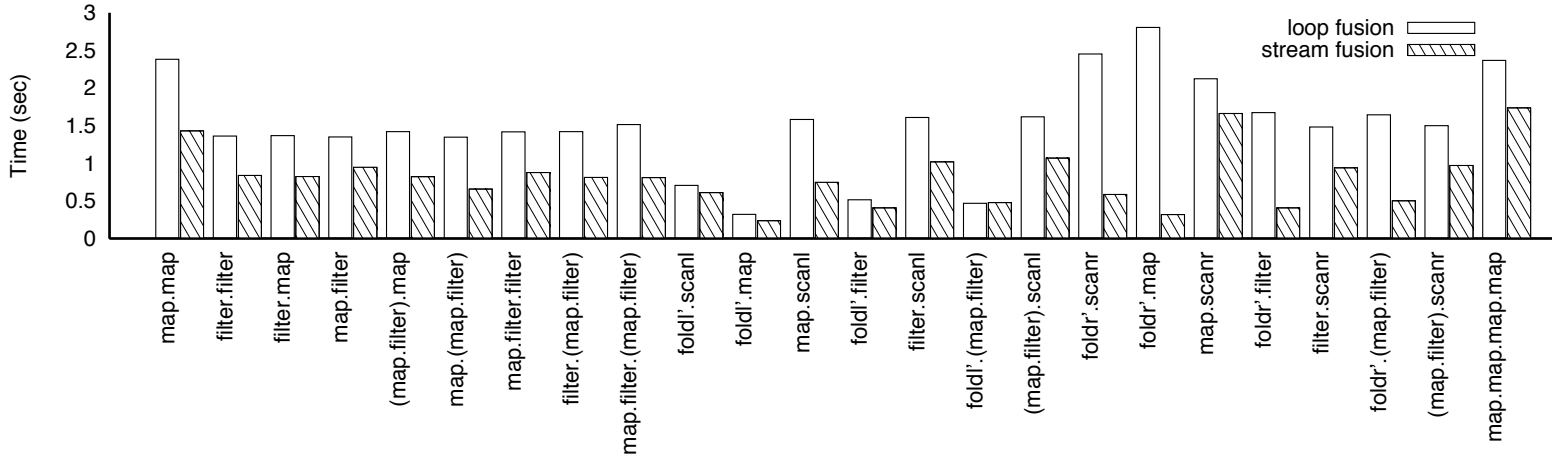
More details: see the paper.

RESULTS

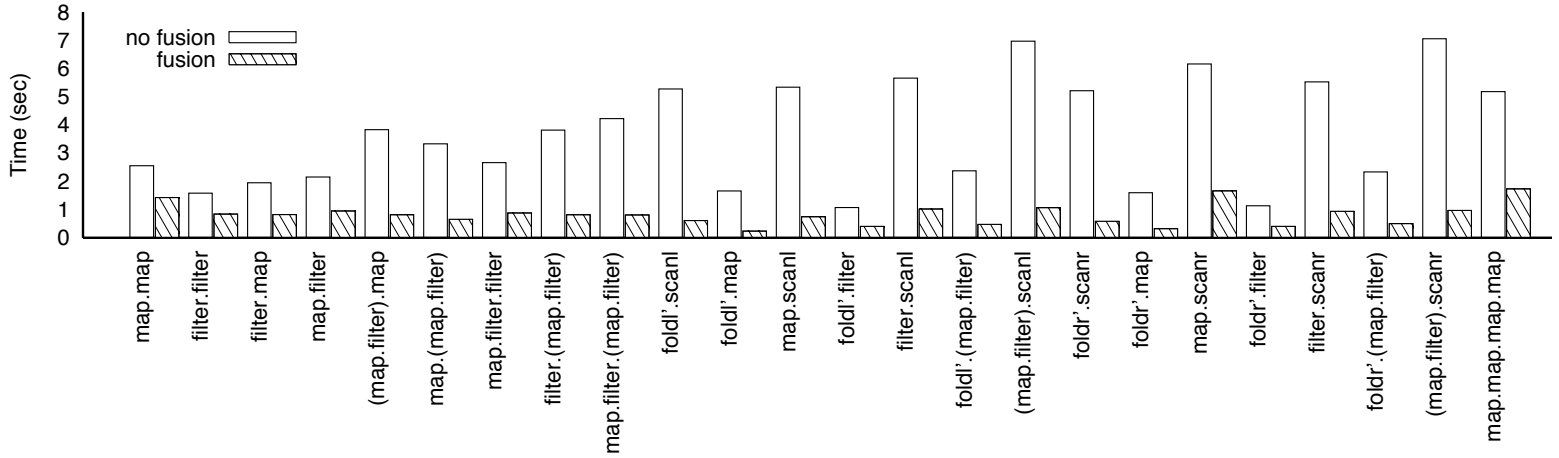
$[Char]$ AND LAZY BYTESTRING : RUNNING TIME



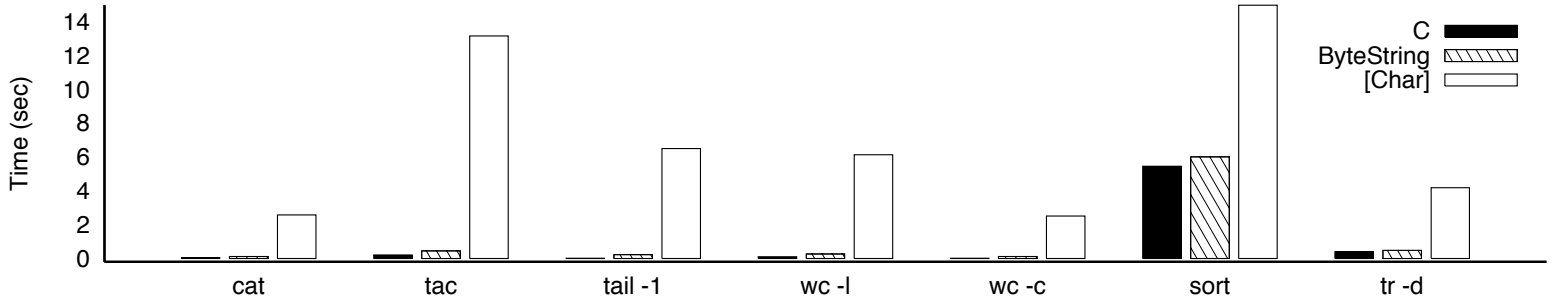
COMPARATIVE FUSION STRATEGIES



EFFECT OF FUSION: STREAMS WITH AND WITHOUT FUSION



C, ByteString and [Char] Unix tools



FUTURE

- ➜ Stream fusion for Haskell lists
- ➜ Improving the GHC backend for tight loops and branch prediction
- ➜ Efficiently fusing multiple traversals: append, zip, concatMap
- ➜ Fusible Binary serialisation
- ➜ Fusion across IO boundaries
- ➜ Establish the safety of streams in the presence of seq
- ➜ Rewrite Great Language Shootout entries, and take back 1st place! ;-)
- ➜ *Done*: Fusible $Storable\ a\ =>\ Vector\ a$ (Spencer Janssen for Summer of Code)

DATA.BYTESTRING

- Home page : `http://www.cse.unsw.edu.au/~dons/fps.html`
- Available with the current version of Hugs
- Available with GHC 6.6