

facebook

Parallel Functional Programming at Scale, at Facebook

Simon Marlow

May 2015





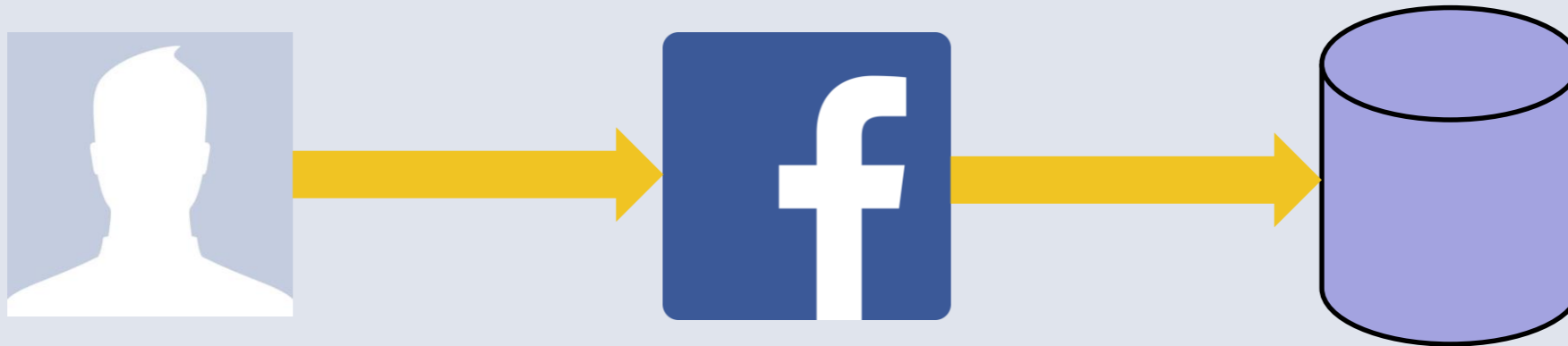
Headlines

- Right now there are thousands of multicore machines running Haskell 24/7 at Facebook
- Haskell is a key part of the anti-abuse infrastructure
- Using a novel kind of implicit parallelism (the Haxl monad)
- This talk
 1. The problem
 - Abuse detection & remediation
 - Why (and how) Haskell?
 2. Applicative do-notation
 3. Experience: tales from the trenches

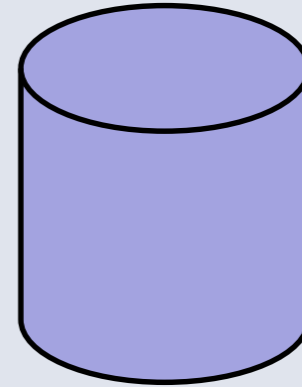
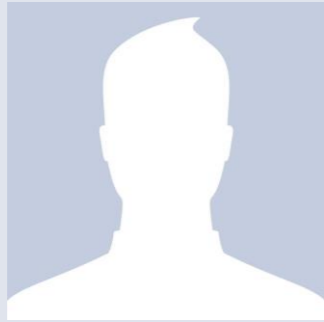
Abuse detection & remediation

The problem

- There is spam and other types of abuse
 - Malware attacks, credential stealing
 - Sites that trick users into liking/sharing things or divulging passwords
 - Fake accounts that spam people and pages
- Spammers can use automation and viral attacks
- Want to catch as much as possible in a completely automated way
- Squash attacks quickly and safely



The write pipeline



Yes!



You can't post this because it has a blocked link.

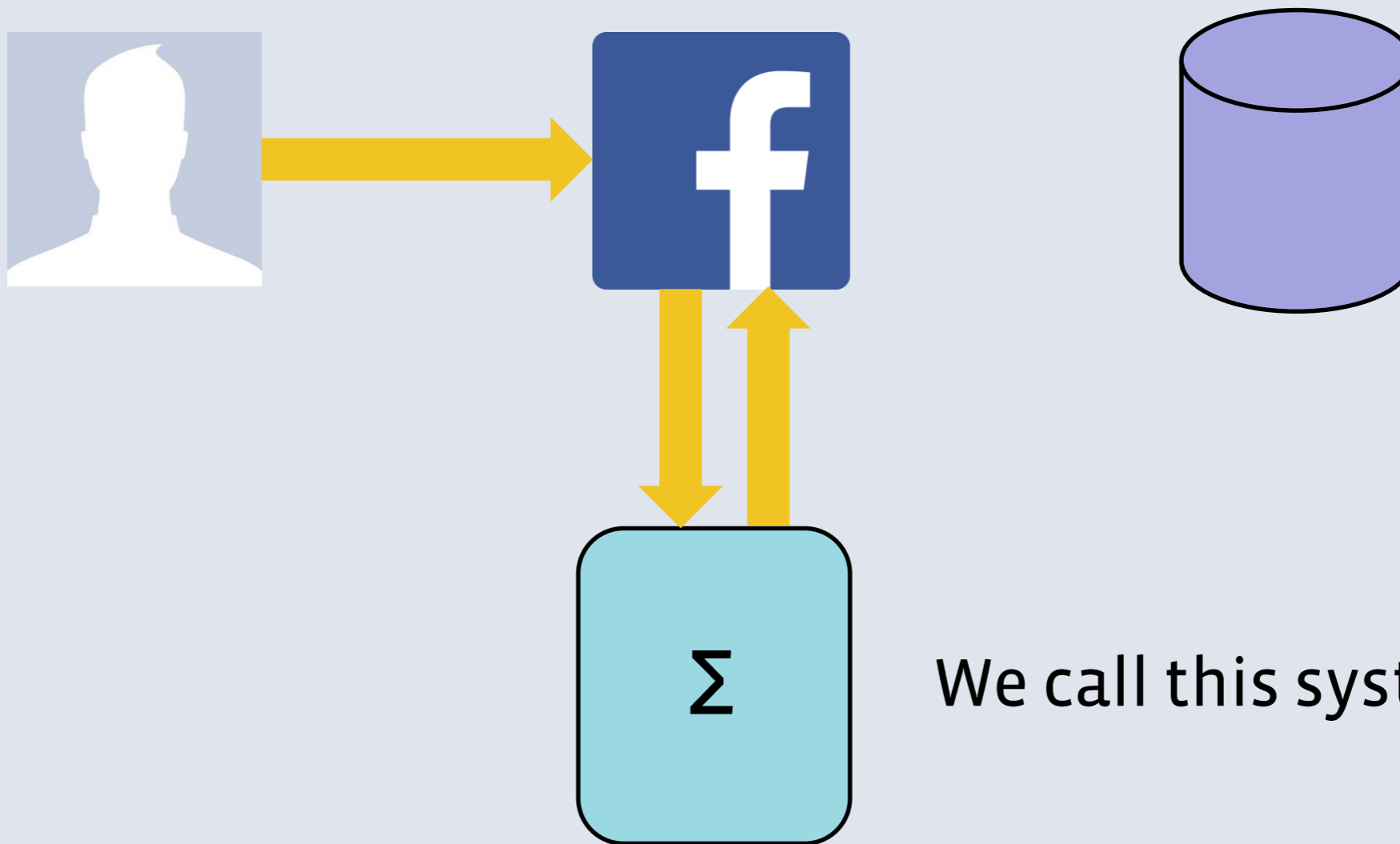
The content you're trying to share includes a link that's been blocked for being spammy or unsafe:

<http://snopes.com/images/template/snopes.gif>

For more information, visit the [Help Center](#). If you think you're seeing this by mistake, please let us know.

Okay

facebook.gif



We call this system Sigma

Sigma :: Content -> Bool

- Sigma classifies tens of billions of actions per day
 - Facebook + Instagram
- Sigma is a *rule engine*
 - For each action type, evaluate a set of rules
 - Rules can block or take other action
 - Manual + machine learned rules
 - Rules can be updated live
- Highly effective at eliminating spam, malware, malicious URLs, etc. etc.

How do we define rules?

Example

- Fanatics are spamming their friends with posts about Functional Programming!
- Let's fix it!

Example

- We want a rule that says
 - If the person is posting about Functional Programming
 - And they have >100 friends
 - And more than half of their friends like C++
 - Then block, else allow

Need info about the content

Need to fetch the friend list

Need info about each friend

Our rule, in Haskell

```
fpSpammer :: Haxl Bool  
fpSpammer =
```

- Haxl is a monad
- “Haxl Bool” is the type of a computation that may:
 - do data-fetching
 - consult input data
 - maybe throw exceptions
 - finally, return a Bool

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP
where
  talkingAboutFP =
    strContains "Functional Programming" <$> postContent
```

- postContent is part of the input (say)

```
postContent :: Haxl Text
```

Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100
where
  talkingAboutFP =
    strContains "Functional Programming" <$> postContent
```

```
(.&&) :: Haxl Bool -> Haxl Bool -> Haxl Bool
(>)  :: Ord a => Haxl a -> Haxl a -> Haxl Bool
numFriends :: Haxl Int
```


Our rule, in Haskell

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100 .&&
  friendsLikeCPlusPlus
where
  talkingAboutFP =
    strContains "Functional Programming" <$> postContent

  friendsLikeCPlusPlus = do
    friends <- getFriends
    cppFriends <- filterM likesCPlusPlus friends
    return (length cppFriends >= length friends `div` 2)
```

Observations

- Our language is Haskell + libraries
 - Embedded Domain-Specific Language (EDSL)
 - Users can pick up a Haskell book and learn about it
 - Tradeoff: not exactly the syntax we might have chosen, but we get to take advantage of existing tooling, documentation etc.
- Focus on expressing functionality concisely, avoid operational details
- “pure” semantics
 - no side effects – easy to reason about
 - scope for automatic optimisation

Efficiency

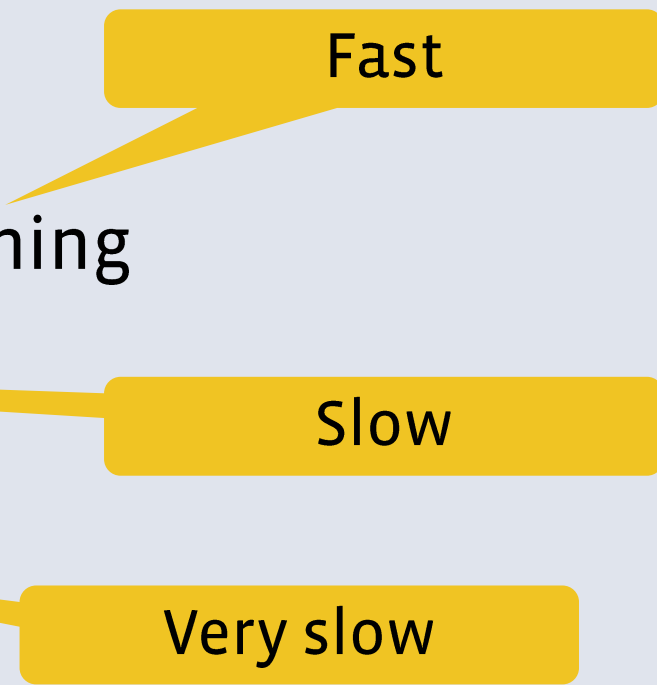
- Rules are data + computation
- Fetching remote data can be slow
- Latency is important!
 - We're on the clock: the user is waiting
- So what about efficiency?

**Fetching data efficiently
is all that matters.**

1. Fetch only the data you need to make a decision
2. Fetch data concurrently whenever possible

Let's deal with (1) first.

Example

- We want a rule that says
 - If the person is posting about Functional Programming
 - And they have >100 friends
 - And more than half of their friends like C++
 - Then block, else allow
 - Avoid slow checks if fast checks already determine the answer
- 

.&& is short-cutting

```
fpSpammer :: Haxl Bool
fpSpammer =
  talkingAboutFP .&&
  numFriends .> 100 .&&
  friendsLikeCPlusPlus
where
  talkingAboutFP =
    strContains "Functional Programming" <$> postContent

  friendsLikeCPlusPlus = do
    friends <- getFriends
    cppFriends <- filterM likesCPlusPlus friends
    return (length cppFriends >= length friends `div` 2)
```



~~(.&&)= liftA2 (&&)~~

- Programmer is responsible for getting the order right
- (tooling helps with this)

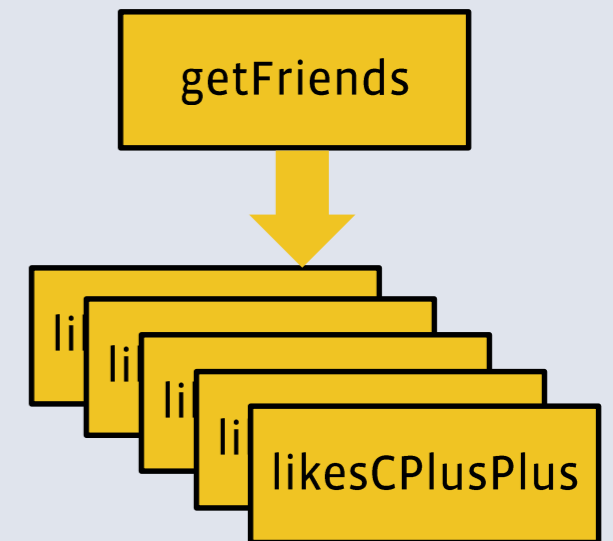
Concurrency

- Multiple independent data-fetch requests must be executed concurrently and/or batched
- Traditional languages and frameworks make the programmer deal with this
 - threads, futures/promises, async, callbacks, etc.
 - Hard to get right
 - Our users don't care
 - Clutters the code
 - Hard to refactor later

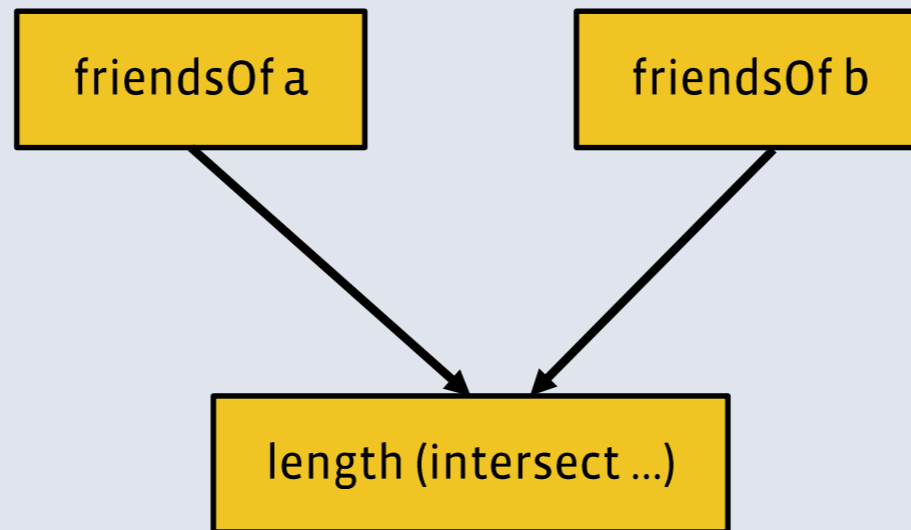
Haxl's advantage

- Because our language has no side effects, the framework can handle concurrency automatically
- We can exploit concurrency as far as data dependencies allow
- The programmer doesn't need to think about it

```
friendsLikeCplusplus = do
  friends <- getFriends
  cppFriends <- filterM likesCplusplus friends
  ...
```



```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```



How does Haxl work?

Step 1

- Haxl is a Monad
- The implementation of ($\gg=$) will allow the computation to block, waiting for data.

```
data Result a
  = Done a
  | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Done

This is the result of a computation

Blocked indicates that the computation requires this data.

Haxl may need to do IO

Monad instance

```
instance Monad Haxl where
```

```
  return a = Haxl $ return (Done a)
```

```
  Haxl m >>= k = Haxl $ do
```

```
    r <- m
```

```
  case r of
```

```
    Done a      -> unHaxl (k a)
```

```
    Blocked br c -> return (Blocked br (c >>= k))
```

If m blocks with continuation c , the continuation for $m \gg= k$ is $c \gg= k$

So far we can only block on *one* data-fetch

- Our example will block on the first friendsOf request:

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

blocks here

- How do we allow the Monad to collect multiple data-fetches, so we can execute them concurrently?

First, rewrite to use *Applicative* operators

```
numCommonFriends a b =  
  length <$> (intersect <$> friendsOf a <*> friendsOf b)
```

- Applicative is a weaker version of Monad

```
class Applicative f where  
  pure   :: a -> f a  
  (<*>) :: f (a -> b) -> f a -> f b  
  
class Monad m where  
  return :: a -> m a  
  (>>=)  :: m a -> (a -> m b) -> m b
```

- When we use Applicative, Haxl can collect multiple data fetches and execute them concurrently.

Applicative instance

```
instance Applicative Haxl where
```

```
  pure = return
```

```
Haxl f <*> Haxl x = Haxl $ do
```

```
  f' <- f
```

```
  x' <- x
```

```
  case (f',x') of
```

```
    (Done g, Done y) -> return (Done (g y))
```

```
    (Done g, Blocked br c) -> return (Blocked br (g <$> c))
```

```
    (Blocked br c, Done y) -> return (Blocked br (c <*> return y))
```

```
    (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
```

- <*> allows both arguments to block waiting for data
- <*> can be nested, letting us collect an arbitrary number of data fetches to execute concurrently

Putting it together

Here is where the actual concurrency and/or batching happens

- Given

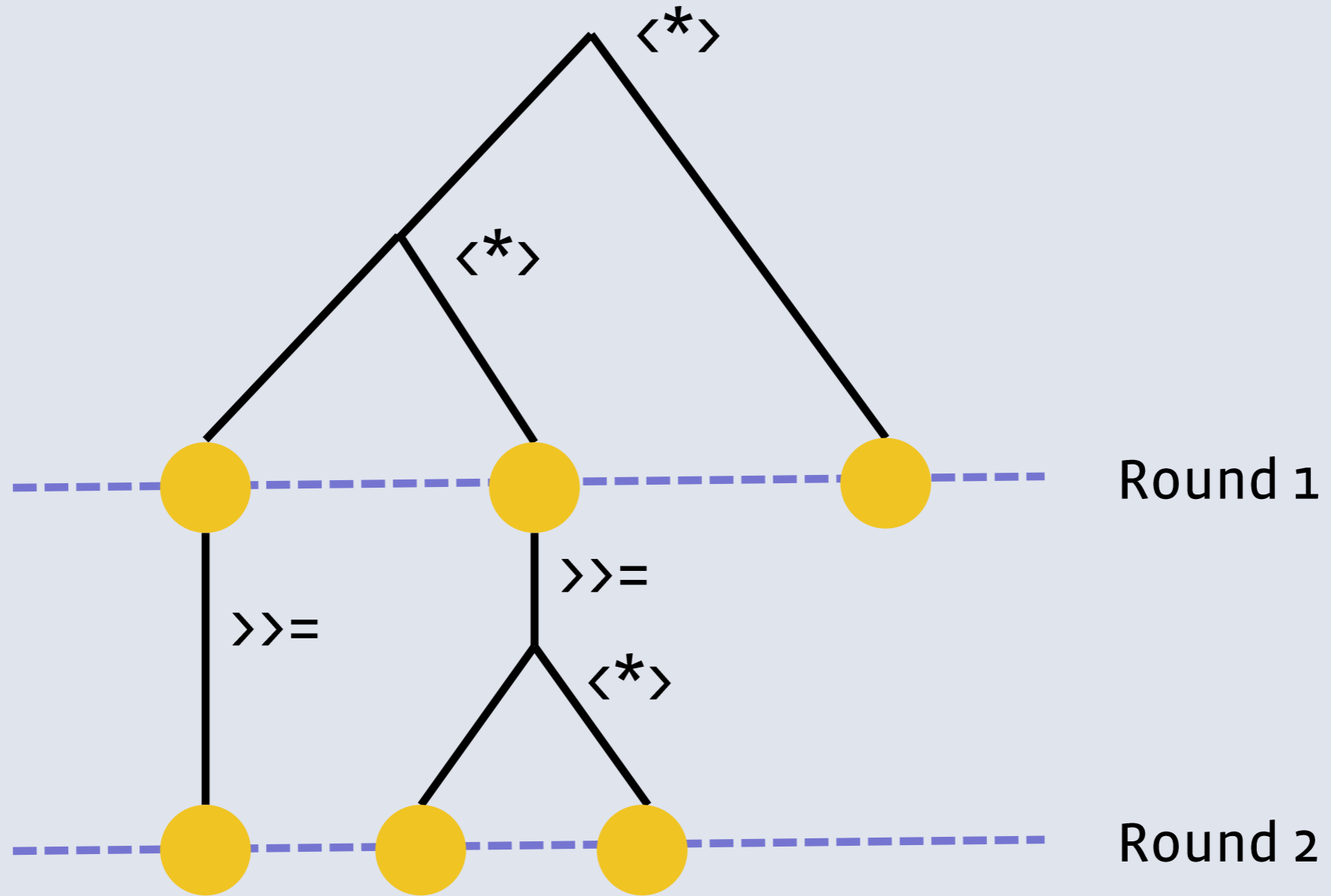
```
fetch :: [BlockedRequest] -> IO ()
```

- We can run a Haxl computation

```
runHaxl :: Haxl a -> IO a
runHaxl (Haxl h) = do
  r <- h
  case r of
    Done a -> return a
    Blocked br cont -> do
      fetch (toList br)
      runHaxl cont
```

Example

```
(intersect <$> friendsOf x) <*> friendsOf y
=
(friendsOf x >>= return . intersect) <*> friendsOf y
=
(Blocked [FriendsOf x] (get (FriendsOf x)) >>= return . intersect)
  <*> friendsOf y
=
  (<$>) = fmap
  fmap f m = m \->- return f
(B friendsOf :: UserId -> Haxl [UserId]
 friendsOf x =
= Haxl (return (Blocked [FriendsOf x] (get (FriendsOf x))
(Blocked [FriendsOf x] (get (FriendsOf x)) >>= return . intersect))
  <*> Blocked [FriendsOf y] (get (FriendsOf y))
=
Blocked [FriendsOf x, FriendsOf y]
  ((get (FriendsOf x)) >>= return . intersect) <*> get (FriendsOf y))
```



(Some) Concurrency for free

- Applicative is a standard class in Haskell
- Lots of library functions are already defined using it
- These work concurrently when used with Haxl
- e.g.

```
sequence :: Monad m => [m a] -> m [a]
mapM     :: Monad m => (a -> m b) -> m [a] -> m [b]
filterM  :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

```
friendsLikeCPlusPlus = do
  friends <- getFriends
  cppFriends <- filterM likesCPlusPlus friends
  ...
```

Haxl is a general solution

- ... to the problem of scheduling I/O
- it's useful anywhere that needs to do I/O and doesn't want to express concurrency explicitly.
- Other examples:
 - a blog engine
 - a build system

Is this *implicit* parallelism?

Granularity

- Tradeoff between granularity and parallelism
 - Lots of fine-grained parallelism, but difficult to exploit
 - hard to distinguish between fine and coarse automatically
- So we're usually happy with either
 - simple annotations
 - automatic parallelism over restricted domains (vectorisation)
 - explicit dataflow (par Monad)

Feedback Directed Implicit Parallelism

Tim Harris

Microsoft Research, Cambridge, UK
tharris@microsoft.com

Satnam Singh

Microsoft Research, Cambridge, UK
satnams@microsoft.com

Abstract

In this paper we present an automated way of using spare CPU resources within a shared memory multi-processor or multi-core machine. Our approach is (i) to profile the execution of a program, (ii) from this to identify pieces of work which are promising sources of parallelism, (iii) recompile the program with this work being performed speculatively via a work-stealing system and then (iv) to detect at run-time any attempt to perform operations that would reveal the presence of speculation.

We assess the practicality of the approach through an implementation based on GHC 6.6 along with a limit study based on the execution profiles we gathered. We support the full Concurrent Haskell language compiled with traditional optimizations and including I/O operations and synchronization as well as pure computation. We use 20 of the larger programs from the 'nofib' benchmark suite. The limit study shows that programs vary a lot in the parallelism we can identify: some have none, 16 have a potential 2x speed-up, 4 have 32x. In practice, on a 4-core processor, we get 10-80% speed-ups on 7 programs. This is mainly achieved at the

We work with programs written in Haskell (Peyton Jones et al. 1996), a pure, lazy, functional language which supports monadic I/O. In principle this language is a great fit for multi-core hardware: purity means that the compiler or run-time system can evaluate multiple parts of a program in parallel without needing to worry about data races. In practice we encounter five problems:

- Programs vary in the amount of parallelism that is actually available. As we show, some have a lot but some have very little.
- Even in programs with abundant parallelism, the work must be at a sufficiently coarse granularity that the parallel speed up compensates for the overheads introduced in managing the work.
- In languages with lazy evaluation, it is not immediately clear which pieces of computation will actually contribute to the 'real' work of the program. Performing un-needed work can harm performance – for example it can allocate a lot of memory and trigger extra garbage collections.

Remote data-fetching changes the equation

- Parallelism between data-fetches is all exploitable, because it's coarse-grained
- So we can make it as implicit as we like
- (we have to wave hands slightly about data-fetching not being strictly pure)

How implicit is Haxl?

- We had to use <*> to get parallelism:

```
blog :: Haxl Html
blog = renderPage <$> leftPane <*> mainPane
```

- When we use >>= (or the “do” syntactic sugar) we get sequential execution

```
getAllPostsInfo :: Haxl [PostInfo]
getAllPostsInfo = do
  ids <- getPostIds
  mapM getPostInfo ids
```

Haxl is semi-implicit

- You can control parallelism using Applicative vs. Monad constructs
- However, these are part of the computational fabric
 - we normally use Applicative and Monad interchangeably when both are available
 - For Haxl we want Applicative to be used “when possible”
 - e.g. `mapM` (= `traverse`), `filterM`, etc.
- Defaulting things to Applicative when possible is a kind of implicit parallelism.
- There’s a ubiquitous way we can do this...

Back to our earlier example

- These behave the same:

```
numCommonFriends a b = do
  fa <- friendsOf a
  fb <- friendsOf b
  return (length (intersect fa fb))
```

This is the
version we
want to write

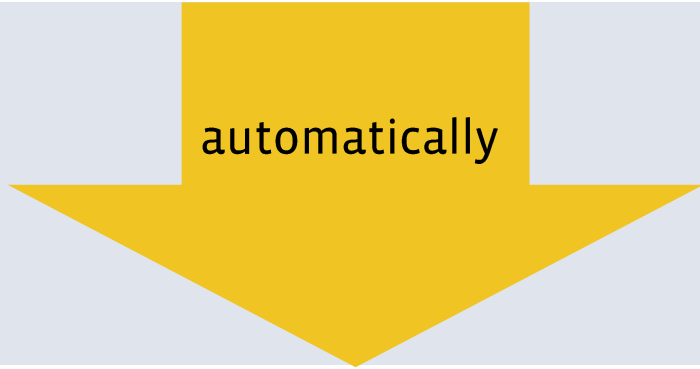
```
numCommonFriends a b =
  length <$> (intersect <$> friendsOf a <*> friendsOf b)
```

This is the
version we
want to run

- Data dependencies tell us we can translate one into the other

Translating do-syntax to Applicative

```
numCommonFriends = do
  fx <- friendsOf x
  fy <- friendsOf y
  return (length (intersect fx fy))
```



automatically

```
numCommonFriends =
  length <$> (intersect <$> friendsOf x <*> friendsOf y)
```

Desugaring Haskell's `do`-notation Into Applicative Operations

Simon Marlow

Facebook
smarlow@fb.com

Simon Peyton Jones

Microsoft Research
simonpj@microsoft.com

Edward Kmett

McGraw Hill Financial
ekmett@mhfi.com

Andrey Mokhov

Newcastle University
andrey.mokhov@ncl.ac.uk

Abstract

Monads have taken the world by storm, and are supported by `do`-notation (at least in Haskell). Programmers are increasingly waking up to the usefulness and ubiquity of `Applicative`s, but they have so far been hampered by the absence of supporting notation. In this paper we show how to re-use the very same `do`-notation to work for `Applicative`s as well, providing efficiency benefits for some types that are both `Monad` and `Applicative`, and syntactic convenience for those that are merely `Applicative`. The result is fully implemented in GHC, and is in use at Facebook to make it easy to write highly-parallel queries in a distributed system.

1. Introduction

Consider this Haskell function that calculates the number of common friends between two Facebook users:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y = do
  fx ← friendsOf x
  fy ← friendsOf y
  return (length (intersect fx fy))
```

Here `friendsOf` is an operation that makes a remote query to a database to fetch the list of friends of a user. Desugaring the monadic `do` expression according to the Haskell standard [10]

a `Monad` lies an `Applicative` [13]. To be concrete, we can rewrite `numCommonFriends` using `Applicative` combinators like this:

```
numCommonFriends :: Id → Id → Haxl Int
numCommonFriends x y =
  (λfx fy → length (intersect fx fy))
  <$> friendsOf x
  <*> friendsOf y
```

The combinators `<$>` and `<*>` are defined in Figure 1, but for now we simply note that the two calls to `friendsOf` are now manifestly independent of one another. And indeed the implementation of the `Haxl` monad¹ can take advantage of that independence to perform the two `friendsOf` queries in parallel; in fact it collects them together and batches them into a single query.

But there is still a problem; programmers should not have to spot where they can use `<*>` to gain its advantages, because they are likely to miss some opportunities, especially when code is refactored. Moreover there are maintainability and comprehensibility benefits in using a single universal notation, namely `do` notation. In this paper we show how to have our cake and eat it too: the programmer writes `do` notation, and the compiler desugars it automatically into the efficient parallel code that uses `Applicative` combinators. We make these contributions:

- Rather than desugaring `do` notation uniformly into `Monad` combinators, we show how to take advantage of the program's dependency structure to selectively use `Applicative` combinators.

Why is this useful?

- We don't have to represent parallelism explicitly with `<*>`
- Just write a sequence of statements
- Compiler analyses the dependencies and extracts the maximum parallelism by transforming the sequence using `<*>` where possible
- We don't have to think about dependencies
- We cannot miss any opportunities accidentally

Start with a simple example

```
do x1 <- A
   x2 <- B
   return (x1,x2)
```

Standard
Haskell
desugaring

```
A >>= \x1 ->
B >>= \x2 ->
return (x1,x2)
```

ApplicativeDo

```
(,) <$> A <*> B
```

equivalent

```
f <$> ma = ma >>= \a ->
           return (f a)
```

```
mf <*> ma = mf >>= \f ->
            ma >>= \a ->
            return (f a)
```


Dependencies prevent `<*>`

```
do x1 <- A
   x2 <- B[x1]
   return (x1,x2)
```

- Now we cannot use `<*>`, because B depends on x1
- This is the essence of the difference between Applicative and Monad:

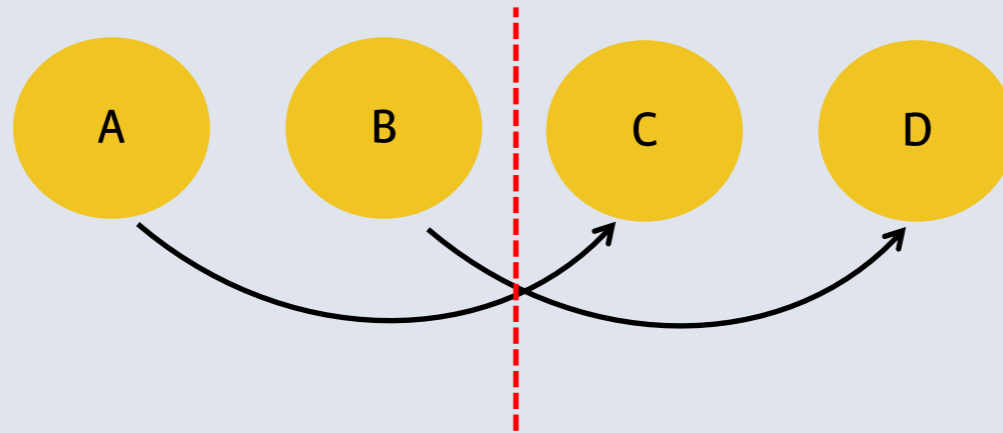
```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
(>>=) :: Monad f      => f a -> (a -> f b) -> f b
```

- So we want to use Applicative when *possible* but Monad when *necessary*.

Mixing it up

- What about

```
do x1 <- A  
  x2 <- B  
  x3 <- C[x1]  
  x4 <- D[x2]  
  return (x3, x4)
```



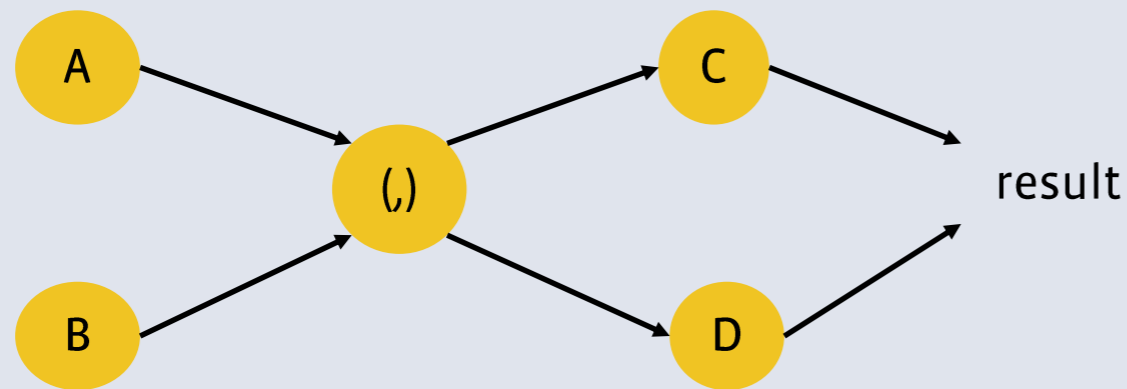
```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```

ApplicativeDo



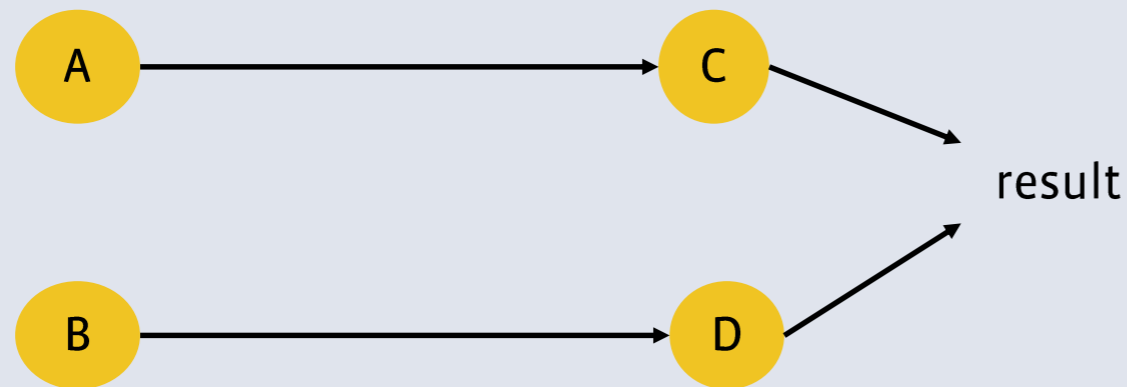
```
((,) <$> A <*> B) >>= \ (x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

But is that the best translation?



$(A \mid B) ; (C \mid D)$

- But we only have dependencies $A \rightarrow C$ and $B \rightarrow D$, so why not



$(A ; C) \mid (B ; D)$

Evaluating cost

- Take a simple parallel cost model
 - “|” = “max”
 - “;” = “+”
- e.g. take $A = 2, B = 1, C = 1, D = 2$

$(A | B); (C | D)$ cost: 4

$(A ; C) | (B ; D)$ cost: 3

Alternative translations

$(A \mid B); (C \mid D)$

```
((, ) <$> A <*> B) >>= \ (x1, x2) ->  
(, ) <$> C[x1] <*> D[x2]
```

$(A ; C) \mid (B ; D)$

```
(, ) <$> (A >>= \x1 -> C[x1])  
      <*> (B >>= \x2 -> D[x2])
```

But...

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```



NO!

```
(, ) <$> (A >>= \x1 -> C[x1])
      <*> (B >>= \x2 -> D[x2])
```

- This is not semantically equivalent to the original
- Effects would take place in the order A,C,B,D

But do we really care about ordering?

- Haxl doesn't – or at least, there are no effects to observe
- But we do want exceptions to be deterministic:

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3, x4)
```

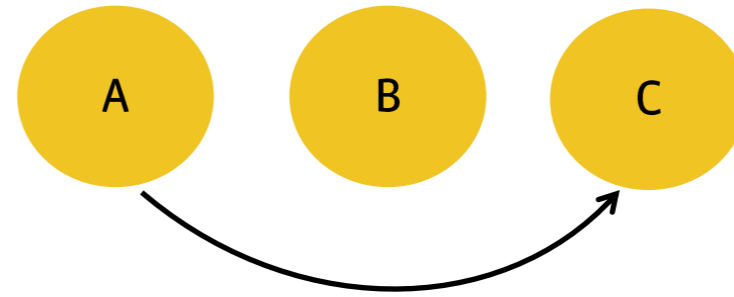
- If B and C throw exceptions, I always want B's exception.
- Reordering to A,C,B,D would break this.

Preserving equivalence is good

- It means `ApplicativeDo` works with any `Monad/Applicative` that satisfies the laws.
- If we reordered things, it would only work on commutative `Monads`.

What does optimal mean?

```
do x <- A  
  y <- B  
  z <- C[x]  
  return (y + z)
```



- Choices:

	A=0, B=2, C=1	A=1, B=2, C=0
(A B); C A; (B C)		

- We don't know the costs at compile time.
- Therefore, be conservative.
- Our goal:

Choose a translation that is optimal when all statements have equal cost.

- (there may be multiple valid solutions)

Refinement: use “join”

```
do x1 <- A
   x2 <- B
   x3 <- C[x1]
   x4 <- D[x2]
   return (x3,x4)
```

ApplicativeDo



```
((,) <$> A <*> B) >>= \ (x1,x2) ->
(,) <$> C[x1] <*> D[x2]
```

- Better:

```
join :: Monad m => m (m a) -> m a
join m = m >>= id
```

```
join ((\x1 x2 -> (,) C[x1] <*> D[x2]) <$> A <*> B)
```

Algorithm sketch

- Two stages:

```
do x1 <- A  
   x2 <- B[x1]  
   x3 <- C  
   return (x2, x3)
```

rearrangement

```
{ x1 <- A; x2 <- B[x1] } | { x3 <- C }
```

desugaring

```
(\x2 x3 -> (x2, x3))  
  <$> (A >>= \x1 -> B[x1])  
  <*> C
```

Rearrangement

- Start with a list of statements $L = \{ s_1 ; \dots ; s_n \}$
- Introduce “parallel blocks” $s = (L_1 \mid \dots \mid L_n)$
- Meaning: just flatten the list
- A parallel block will turn into an applicative expression

```
do x1 <- A  
   x2 <- B[x1]  
   x3 <- C  
   return (x2, x3)
```



rearrangement

```
{ x1 <- A; x2 <- B[x1] } | { x3 <- C }
```

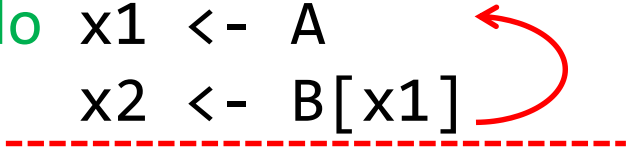
Where do we introduce parallel blocks?

- Take the sequence without the final return
- (desugaring will put it back later)

```
do x1 <- A
   x2 <- B[x1]
   x3 <- C
   return (x2,x3)
```

- Split the sequence into *segments*
- Place a segment boundary between two statements when there are no dependencies that cross the boundary
- Make a parallel block from the segments; apply recursively

```
do x1 <- A
   x2 <- B[x1]
   x3 <- C
   return (x2,x3)
```



```
rearrange { x1 <- A; x2 <- B[x1] }
| rearrange { x3 <- C }
```

What if there are no segments?

- If it's a single statement: we're done
- Otherwise we need a “;” somewhere
- In this case we have no choice:
- (we'll do a more complex example shortly)
- Result of rearrangement:

```
rearrange { x3 <- C }  
= { x3 <- C }
```

```
rearrange { x1 <- A; x2 <- B[x1] }  
= { x1 <- A; x2 <- B[x1] }
```

```
{ x1 <- A; x2 <- B[x1] } | { x3 <- C }
```


Next, desugar to get an expression

```
desugar ({ x1 <- A; x2 <- B[x1] } | { x3 <- C }) (x2,x3)
```

The expression
from “return”

- desugaring a parallel block yields an Applicative expression:

```
(\x2 x3 -> (x2,x3))  
  <$> desugar { x1 <- A; x2 <- B[x1] } x2  
  <*> desugar { x3 <- C } x3
```

```
(\x2 x3 -> (x2,x3))
  <$> desugar { x1 <- A; x2 <- B[x1] } x2
  <*> desugar { x3 <- C } x3
```

- First, deal with this:

```
desugar { x3 <- C } x3
  =
  C
```

- Next:

```
desugar { x1 <- A; x2 <- B[x1] } x2
  =
  A >>= \x1 -> desugar { x2 <- B[x1] } x2
  =
  A >>= \x1 -> B[x1]
```

Result

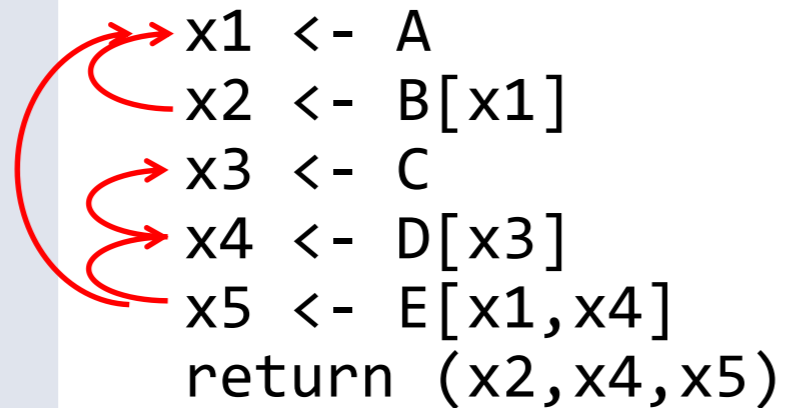
```
do x1 <- A  
   x2 <- B[x1]  
   x3 <- C  
   return (x2, x3)
```



```
(\x2 x3 -> (x2, x3))  
  <$> (A >>= \x1 -> B[x1])  
  <*> C
```

A more complex example

```
x1 <- A  
x2 <- B[x1]  
x3 <- C  
x4 <- D[x3]  
x5 <- E[x1, x4]  
return (x2, x4, x5)
```



- Rearrange:
 - There are no segments
 - We have to insert “;” somewhere
 - And end up with the optimal result

Finding the optimal result

- Just evaluate all possibilities:

- Starting with $\{ s_1 ; \dots ; s_n \}$

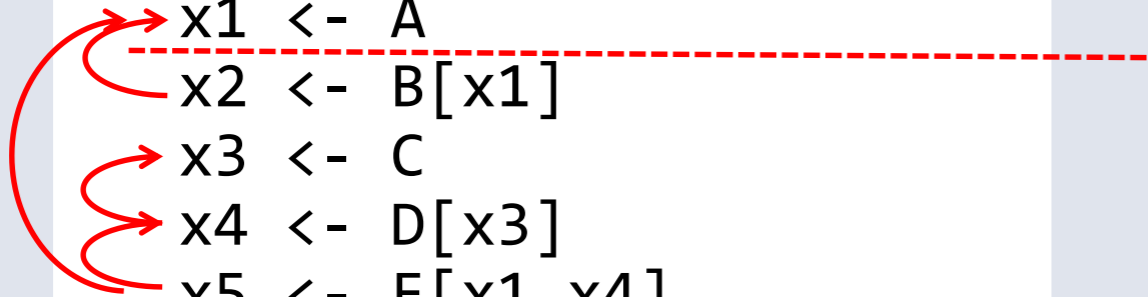
- For each i in $2..n$, compute

rearrange $\{ s_1 ; \dots ; s_{i-1} \}$; rearrange $\{ s_i ; \dots ; s_n \}$

- Evaluate with parallel cost model, with “|” = “max” and “;” = “+”
- Every statement costs 1
- Pick the cheapest!

```
x1 <- A  
x2 <- B[x1]  
x3 <- C  
x4 <- D[x3]  
x5 <- E[x1, x4]  
return (x2, x4, x5)
```

```
A ; (B|{C;D;E}) (cost 4)
```



```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```

A;B ; C;D;E (cost 5)

```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1, x4]
return (x2, x4, x5)
```

({A;B} | C) ; D;E (cost 4)


```
x1 <- A
x2 <- B[x1]
x3 <- C
x4 <- D[x3]
x5 <- E[x1,x4]
return (x2,x4,x5)
```

({A;B} | {C;D}) ; E (cost 3)

We have a winner!

- After desugaring:

```
join (\(x1,x2) x4 ->
      E[x1,x4] >>= \x5 -> pure (x2,x4,x5))
  <$> (A >>= \x1 -> B[x1] >>= \x2 -> return (x1,x2))
  <*> (C >>= \x3 -> D[x3])
```

- Exhaustive search is $O(n^3)$
- The “segments” part of the algorithm cuts down the search space
 - (in the paper we have a proof that it doesn’t affect optimality)
- There are other tricks to cut down the search space
- ... but in practice we use a heuristic instead of the full $O(n^3)$ search
 - heuristic is worse in 1.4% of cases
- Full details in the paper, “Translating Haskell’s do-notation into Applicative operations” (under submission)

Results

- This transform is being used across our codebase at Facebook
- Users typically don't worry about concurrency
- There are a few pitfalls:
 - explicit use of `>>=`
 - “shortcut” functions that take Haxl computations as arguments can break things
 - we want all the Haxl computations visible in the same do-block

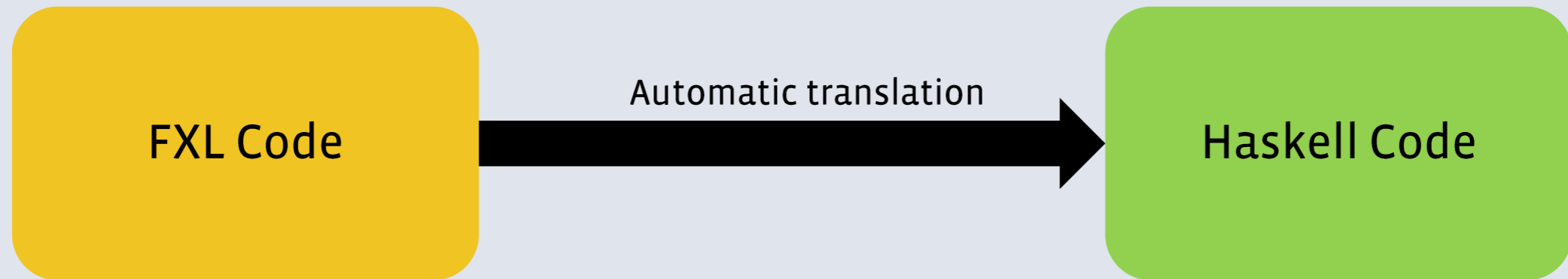
Haxl project: experience

The Haxl project

- We had an existing system and home-grown DSL, called FXL, and lots of code written in it
- Started April 2013
- By July 2015 we had deleted all the FXL code and replaced it with Haskell, and trained our engineers to use Haskell.

1. Existing source code

- hundreds of thousands of lines of existing FXL code
- Impractical and error-prone to translate code by hand
- Wrote a tool to do the migration



- Source code still FXL (during the migration), run the tool each time the code changes.

2. Migrating running code to Haskell



2. Migrating running code to Haskell

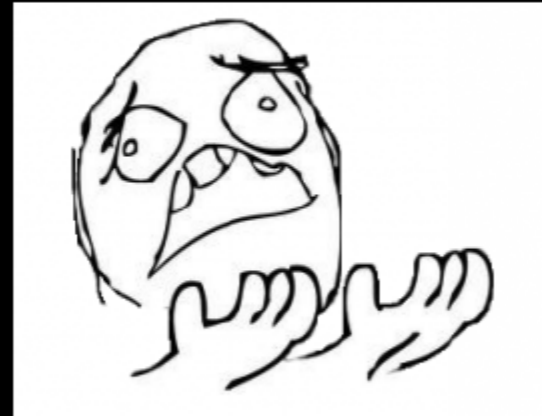
- hundreds of different requests (one for each write action)
- had to ensure that each one:
 - performed well enough
 - gave exactly the same answers
 - (otherwise we introduce false positives/negatives)
- As each request type is ready, we want to switch it over to running on Haskell in production

```
fxlsh> Round(0.5)
```

```
1
```

```
haxlsh> round 0.5
```

```
0
```



At scale, the edge cases happen all the time

- Invalid Unicode
- Invalid arguments to primitives & library functions
- Exception behaviour, values of exceptions
- Is “NaN” a valid number in JSON? What about “infinity”?
- Floating-point:
 - round 0.5
 - printing floating-point values
- divide-by-zero throws in FXL
- semantics of `\s` in regex with Unicode
- etc. etc.

3. Migrating the users

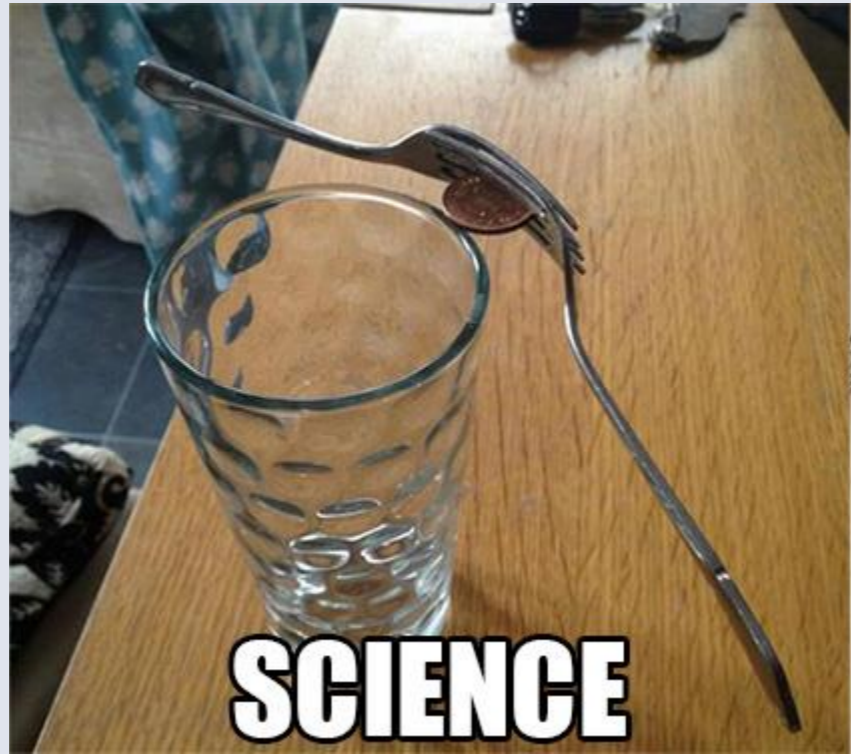
- Dozens of FXL users in multiple geographical locations
- Wrote a lot of teaching material
- Ran multi-day hands-on workshops
- Created internal Facebook group for questions (“Haxl Therapy”)
- Haxl team helped with code reviews



How did users cope with the switch?

- Still committing happily
- Some struggles with do-notation vs. `fmap`, `<$>`, `=<<`
 - “How do I convert a `Haxl t` to `t`?”
- Users started embracing the new features
 - Started building abstractions, adding types, creating tests
- Unblocks some large-scale rewrites and redesigns of subsystems

Does it *work*?

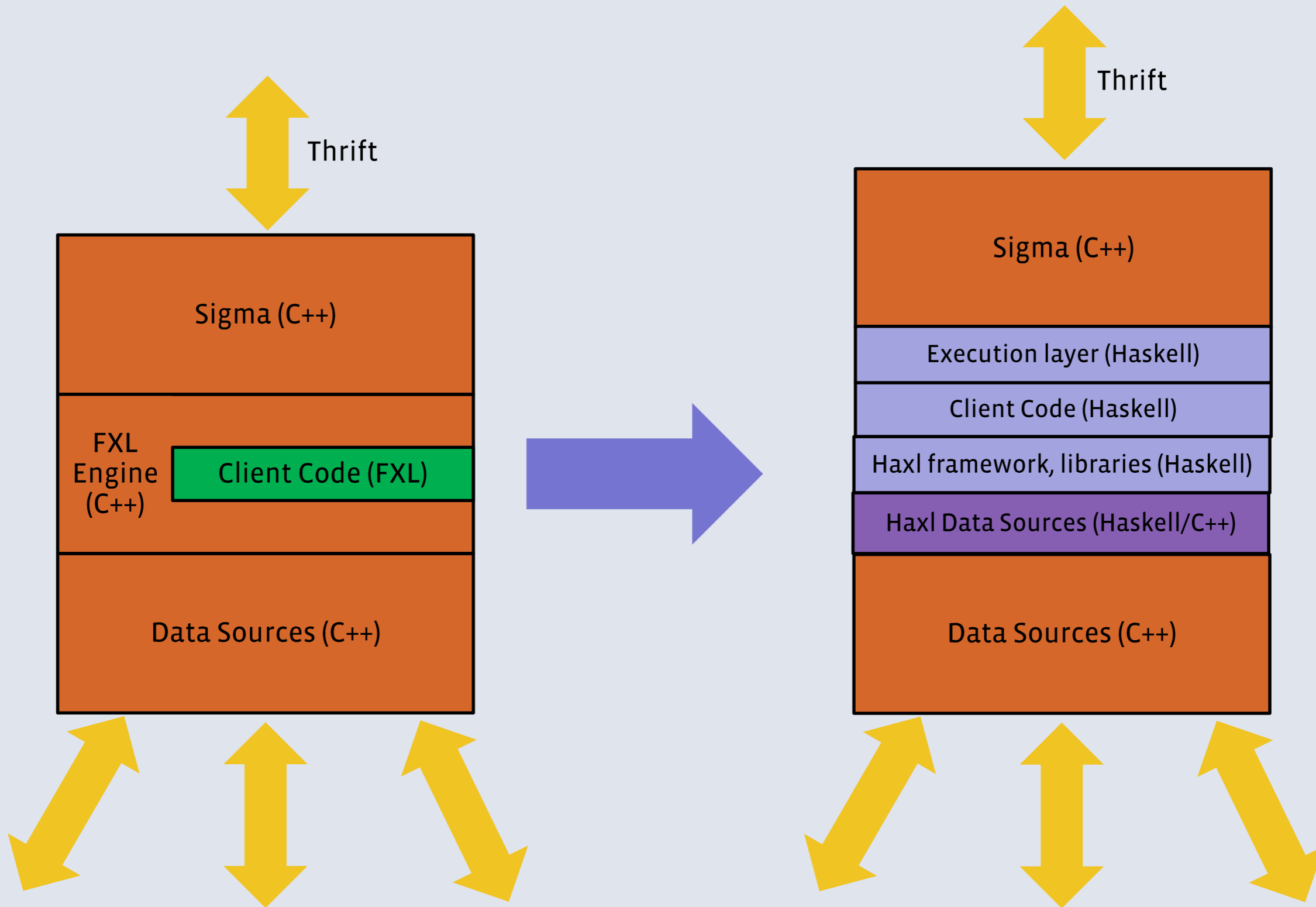


SCIENCE



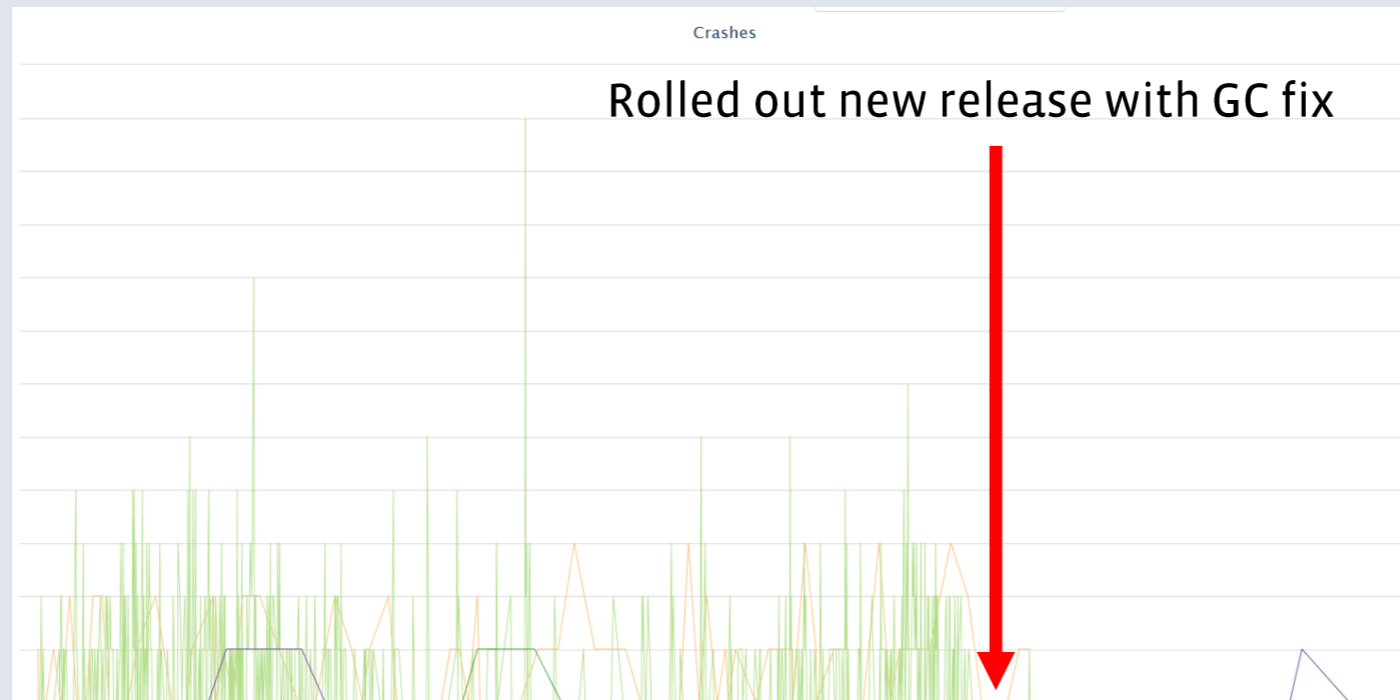
ENGINEERING

- Is it stable enough?
- Is performance good enough?
- Did we have to hack the compiler?
- What about build systems, packages, cabal hell, etc. etc.
- How do we do live updates?



Found one bug in the GHC garbage collector

- There was a multi-year-old bug in the GC that caused our machines to crash every few hours under constant load.



- ~~This is the only runtime bug we've found~~
 - (we found one more that only affected shutdown)

Stability

- The Haskell code just doesn't crash. (*)
 - which is good, because diagnosing a crash in Haskell is Very Hard

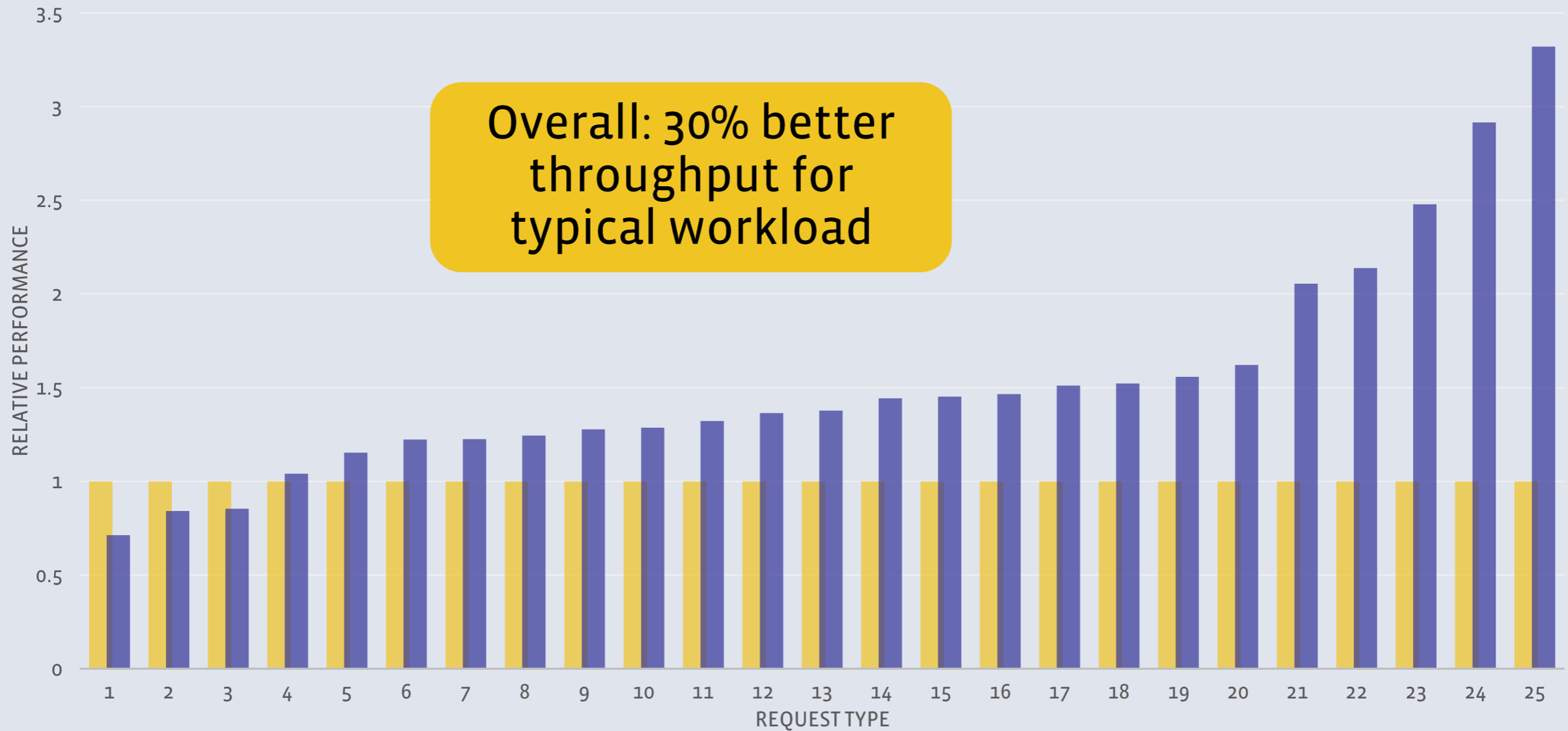
- (*) except for FFI code
 - One FFI-related memory bug was particularly hard to find

Performance

Performance

Haxl performance vs. FXL

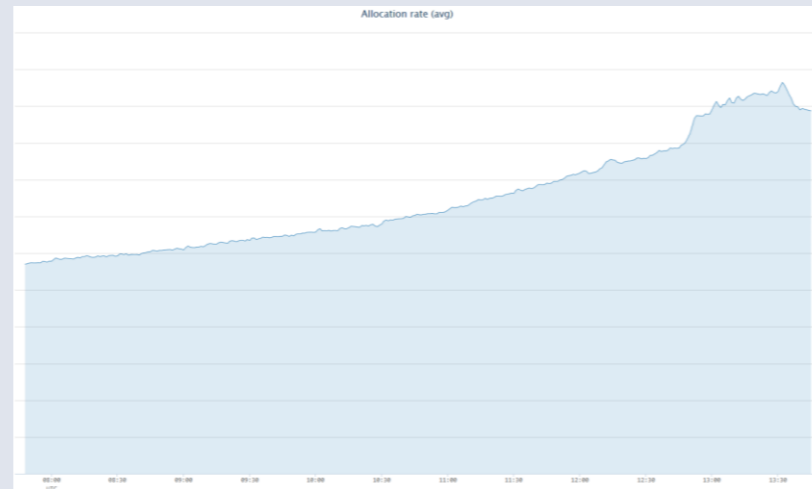
■ FXL ■ Haxl



Overall: 30% better throughput for typical workload

Good monitoring is essential

- Noisy environment:
 - multiple sources of changes
 - dependencies on other services
 - traffic isn't constant
 - spam/malware attacks are bursty
- Hooked up GHC's garbage collector stats API to Facebook's monitoring infrastructure



Resource limits

- Server resources are finite, and we have latency bounds
- Our job is to keep the system responsive, so we cannot allow individual requests to hog resources
- Fact of life: individual requests *will* sometimes hog resources if allowed to
 - Usually: doing some innocuous operation on untypically large data
 - e.g. regex engines sometimes go out to lunch

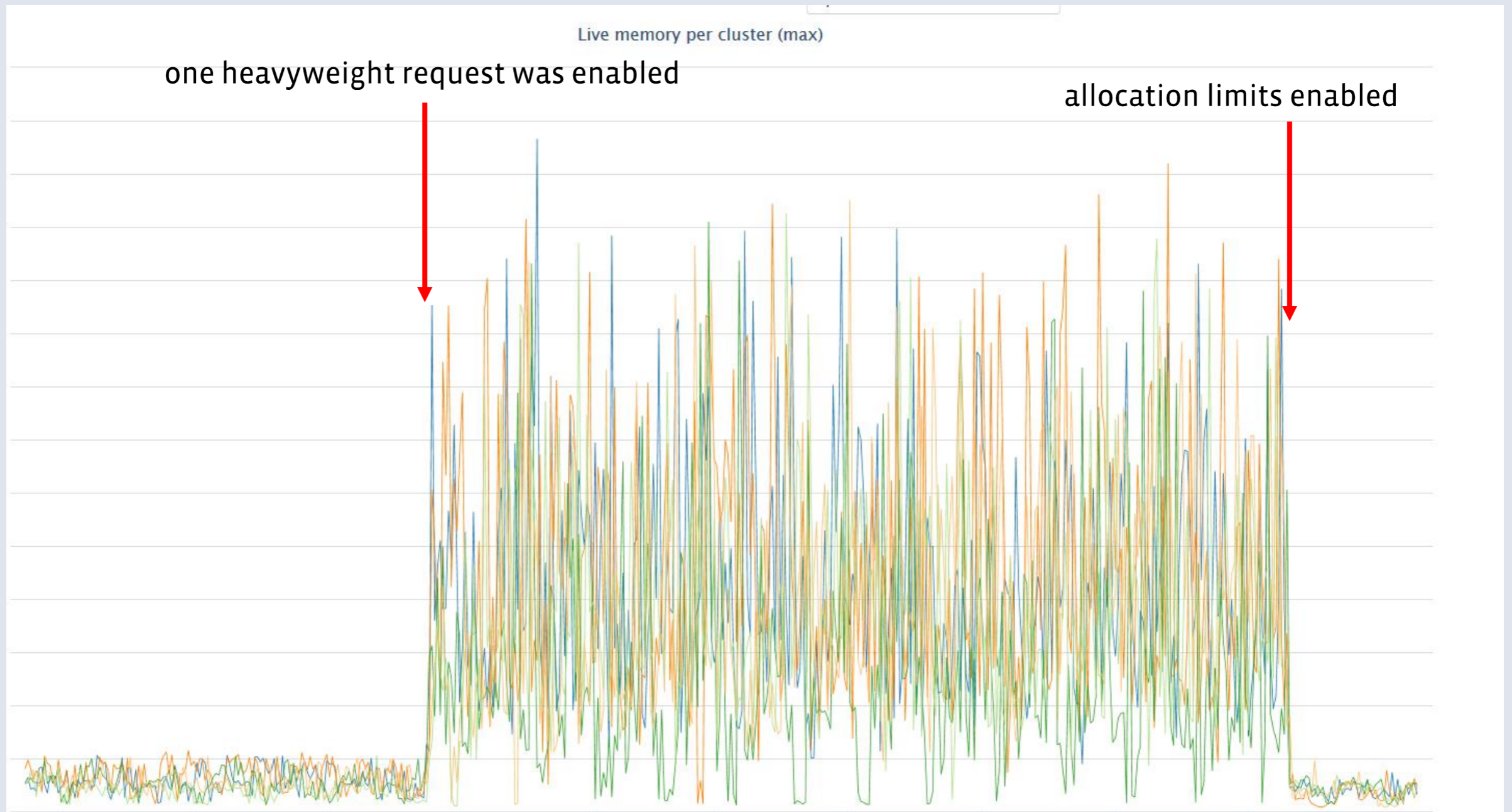
Allocation limits

- So we implemented allocation limits in GHC

```
setAllocationCounter    :: Int64 -> IO ()  
getAllocationCounter   :: IO Int64  
  
enableAllocationLimit  :: IO ()  
disableAllocationLimit :: IO ()
```

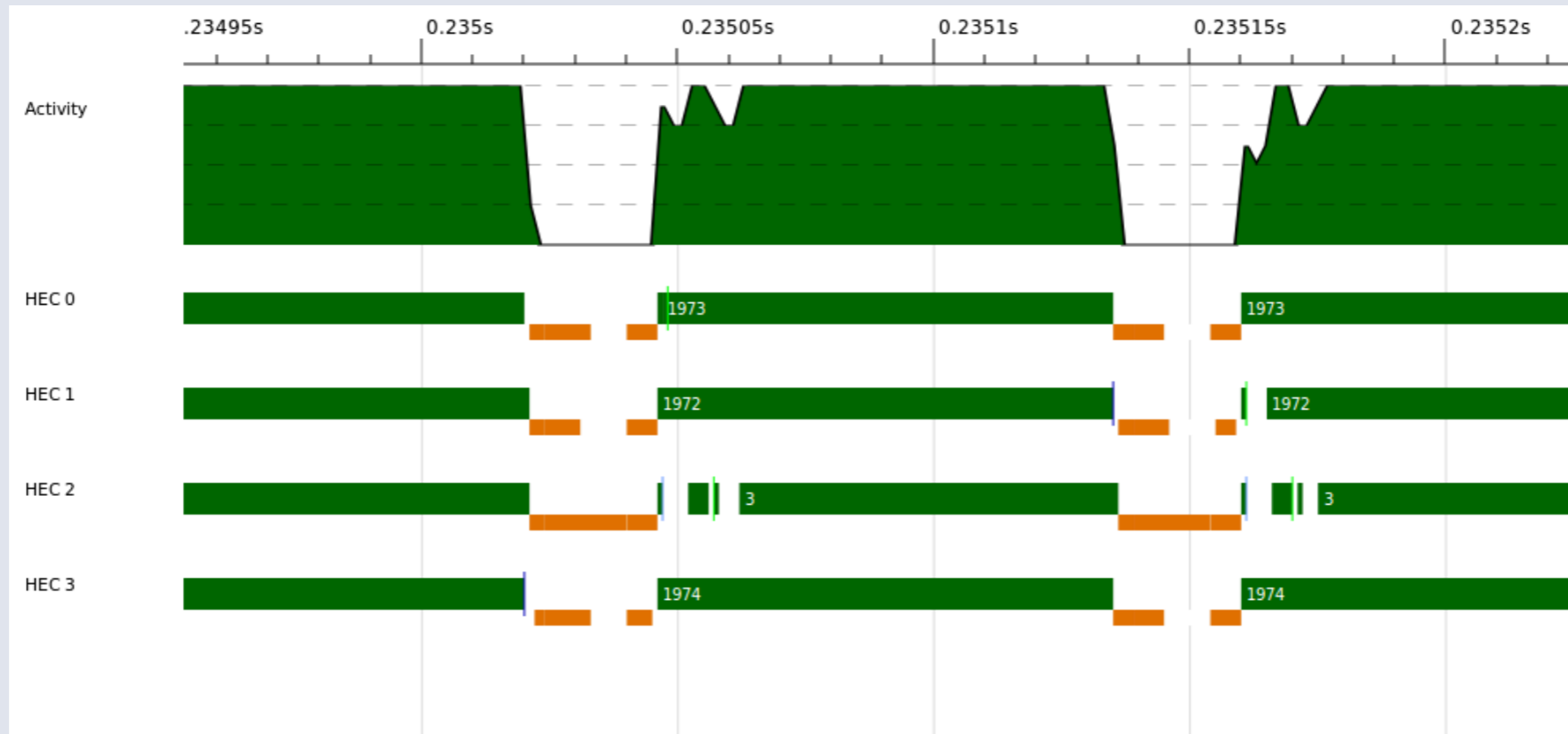
- Counter is per-thread, ticks down with memory allocation
- Triggers an exception when the counter reaches zero
- Easy in Haskell, very difficult in C++

How important are allocation limits?



GC vs. latency

- GHC has a stop-the-world parallel collector



- obviously to meet our latency goals we cannot GC for too long

GC vs. latency

- So how do we manage this?
- Fixed number of worker threads + allocation limits
 - effectively puts a bound on the amount of work we are doing at any given time
- Very little persistent state (a few MB).
- A handful of GC improvements, all upstreamed

Hot-swapping code

Fast deployment

- The faster we can get new rules into production, the more spam we catch before people see it, the faster we stop viral malware, etc.
- (Not all changes need immediate deployment: code review is the norm)
- “Code in the repo is what is running in production”
- Deployment typically on the order of a few minutes

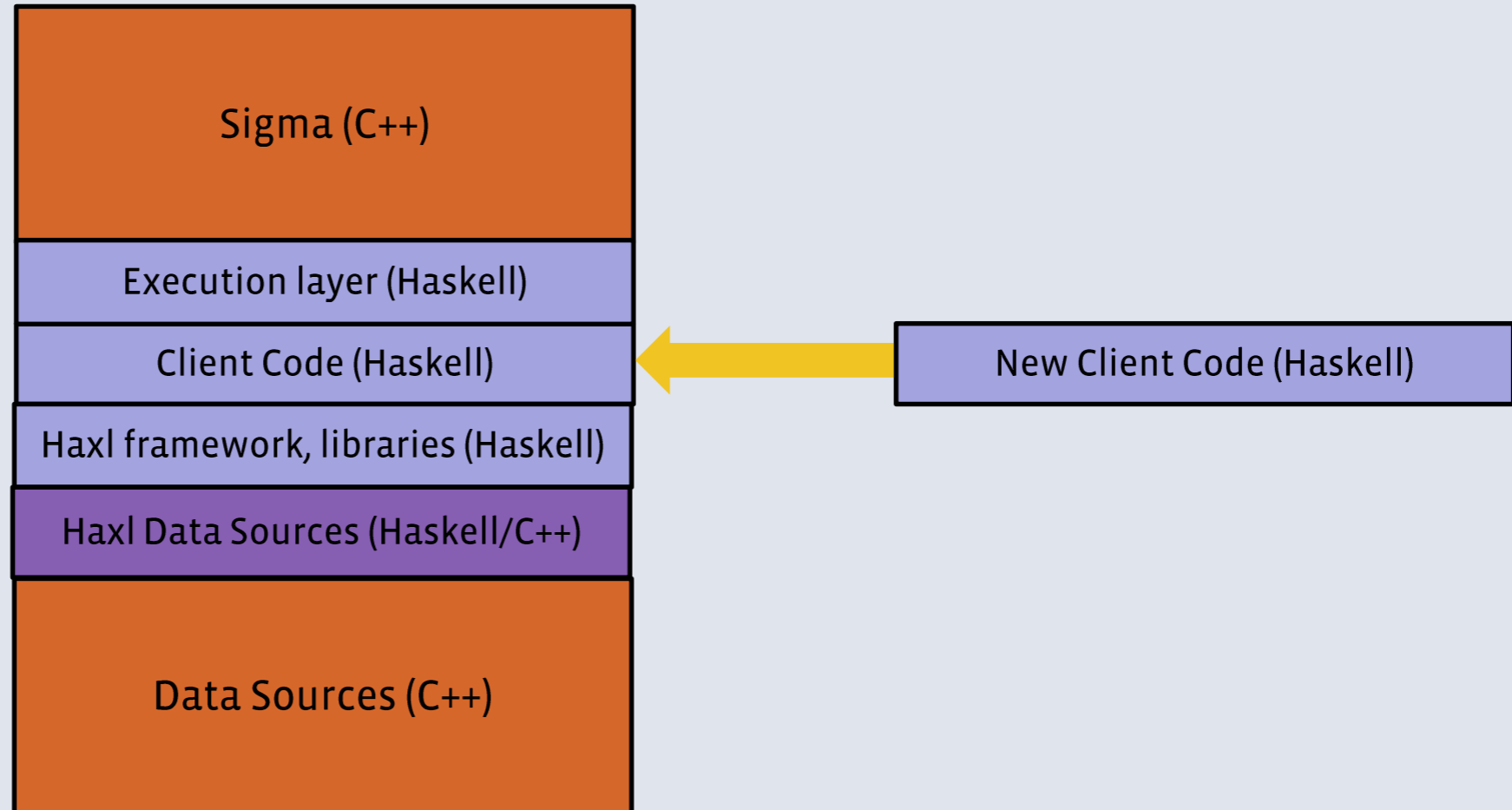
How can we deploy new code?

- Haskell has an optimising compiler, runs native code
- Haskell code needs to be compiled and distributed to servers
- Servers need to start running the new code somehow

Restarting the process doesn't work

- Takes a while to start up
- Caches would be cold
- A rolling restart would take too long

Live updates



Live Updates

- Main idea
 - load the new code directly into the running process
 - needs a dynamic linker
 - Start taking requests using the new code
 - When all requests running on the old code have finished, remove it from the process
- GHC's runtime has a built-in linker
- We added support for unloading objects with GC integration

facebook

Questions?

The Haxl Team, past and present

Jonathan Coens

Bartosz Nitka

Aaron Roth

Kubo Kováč

Katie Ots

Jon Purdy

Zejun Wu

Jake Lengyel

Andrew Farmer

Louis Brandy

Noam Zilberstein

Simon Marlow