

Three improvements to the Reduceron

Matthew Naylor and Colin Runciman
University of York

(Talk given at HFL, March 2009)

The Reduceron

A custom computer designed to run functional programs,



not restricted by conventional architectural constraints,



implemented on an FPGA using a functional language.

This talk

1

Graph reduction and widening the von Neumann bottleneck.

2

Three improvements to the Reduceron since September 2008.

3

How the Reduceron is described.

Graph Reduction

Suppose that **f** is defined by

$$\mathbf{f\ x\ y\ z} = \mathbf{g\ y\ (h\ z\ x)}$$

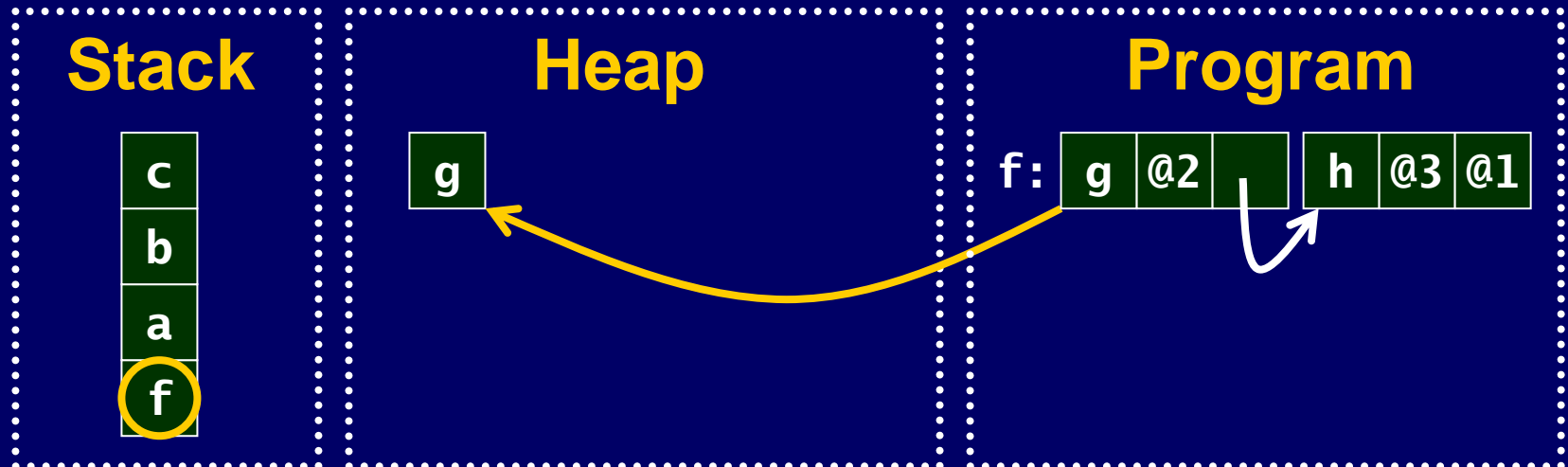
where **g** and **h** are functions and the following machine-state arises during reduction.



Graph Reduction

Operation: $f \leftarrow \text{Stack}[0]$
 $g \leftarrow \text{Code}[f]$
 $g \rightarrow \text{Heap}$

Count: 3



Graph Reduction

Operation: `arg` \leftarrow `Code[f+1]`
 `b` \leftarrow `Stack[arg]`
 `b` \rightarrow `Heap`

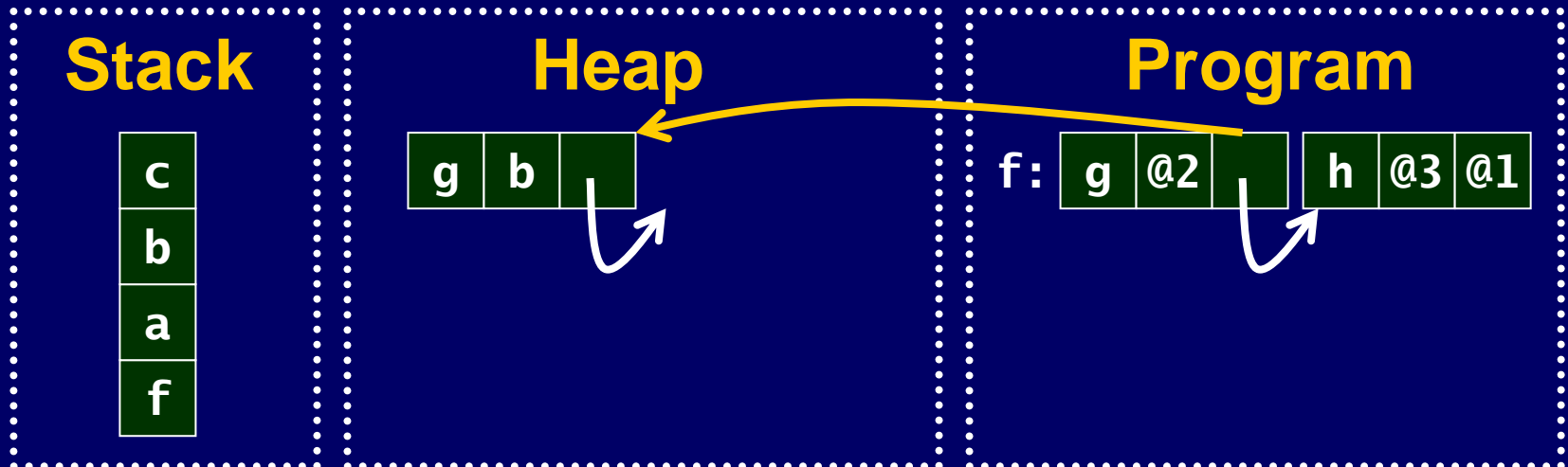
Count: 6



Graph Reduction

Operation: `ptr` \leftarrow `Code[f+2]`
 `ptr'` \rightarrow `Heap`

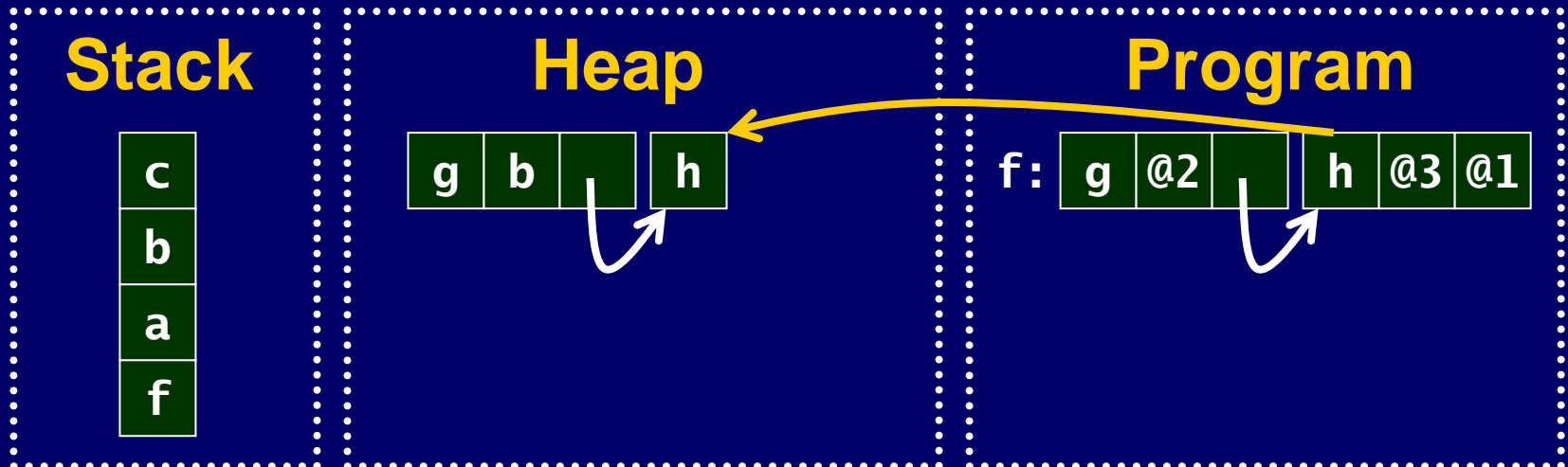
Count: 8



Graph Reduction

Operation: $h \leftarrow \text{Code}[f+3]$
 $h \rightarrow \text{Heap}$

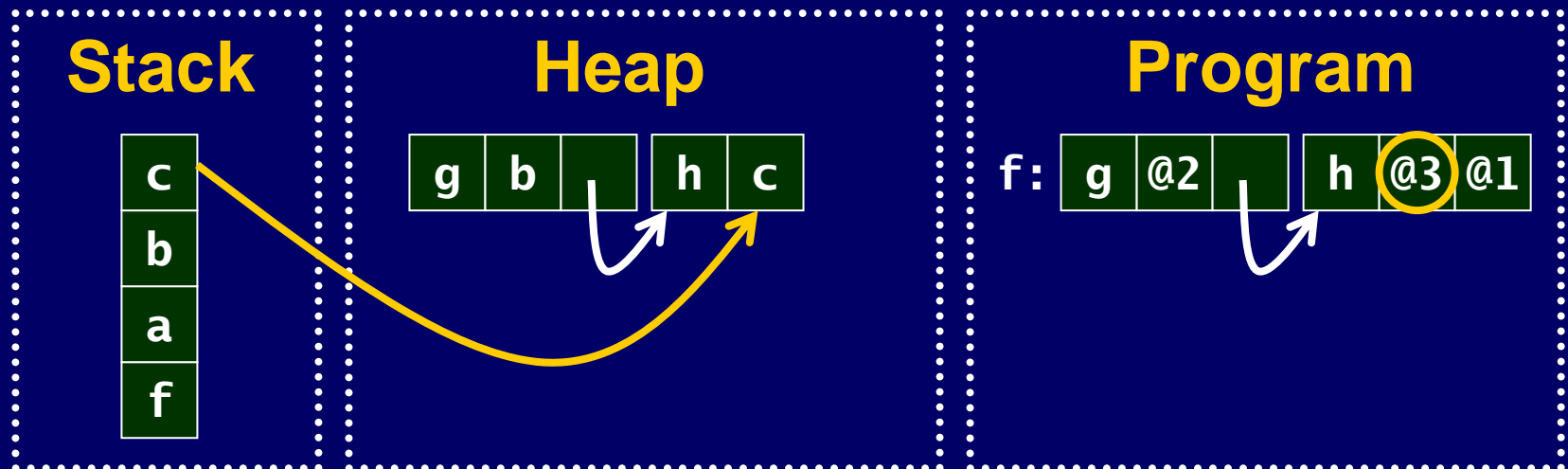
Count: 10



Graph Reduction

Operation: `arg` \leftarrow `Code[f+4]`
 `c` \leftarrow `Stack[arg]`
 `c` \rightarrow `Heap`

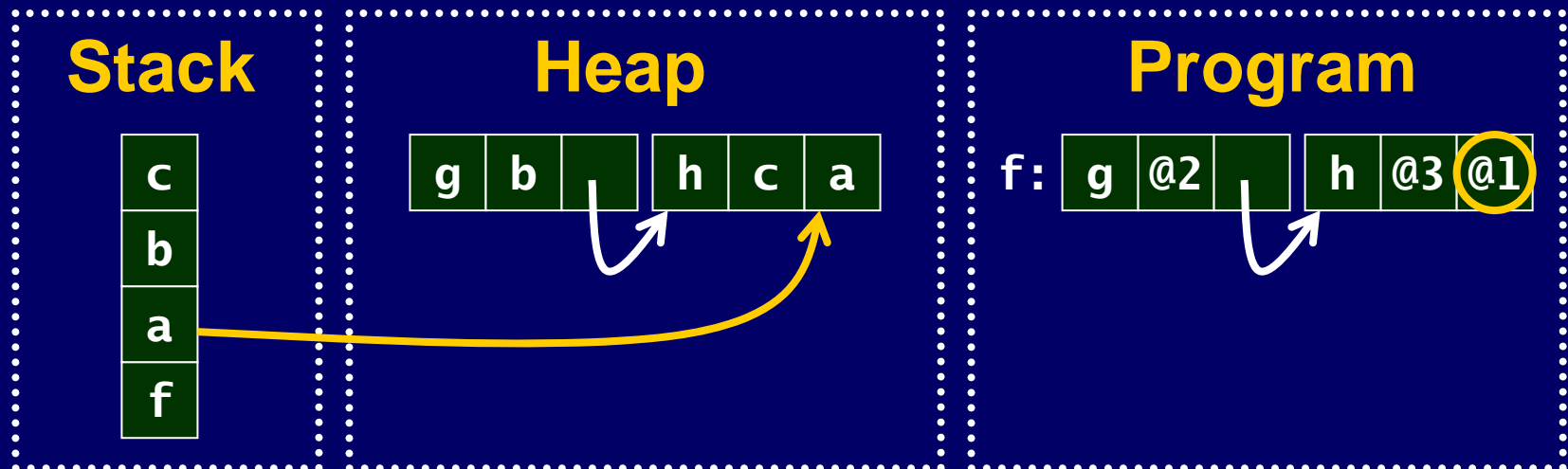
Count: 13



Graph Reduction

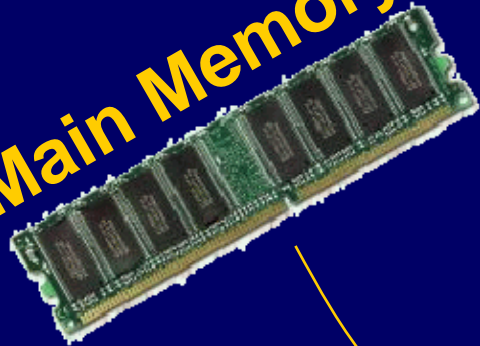
Operation: `arg` \leftarrow `Code[f+5]`
 `a` \leftarrow `Stack[arg]`
 `a` \rightarrow `Heap`

Count: 16



The von Neumann Bottleneck

Main Memory

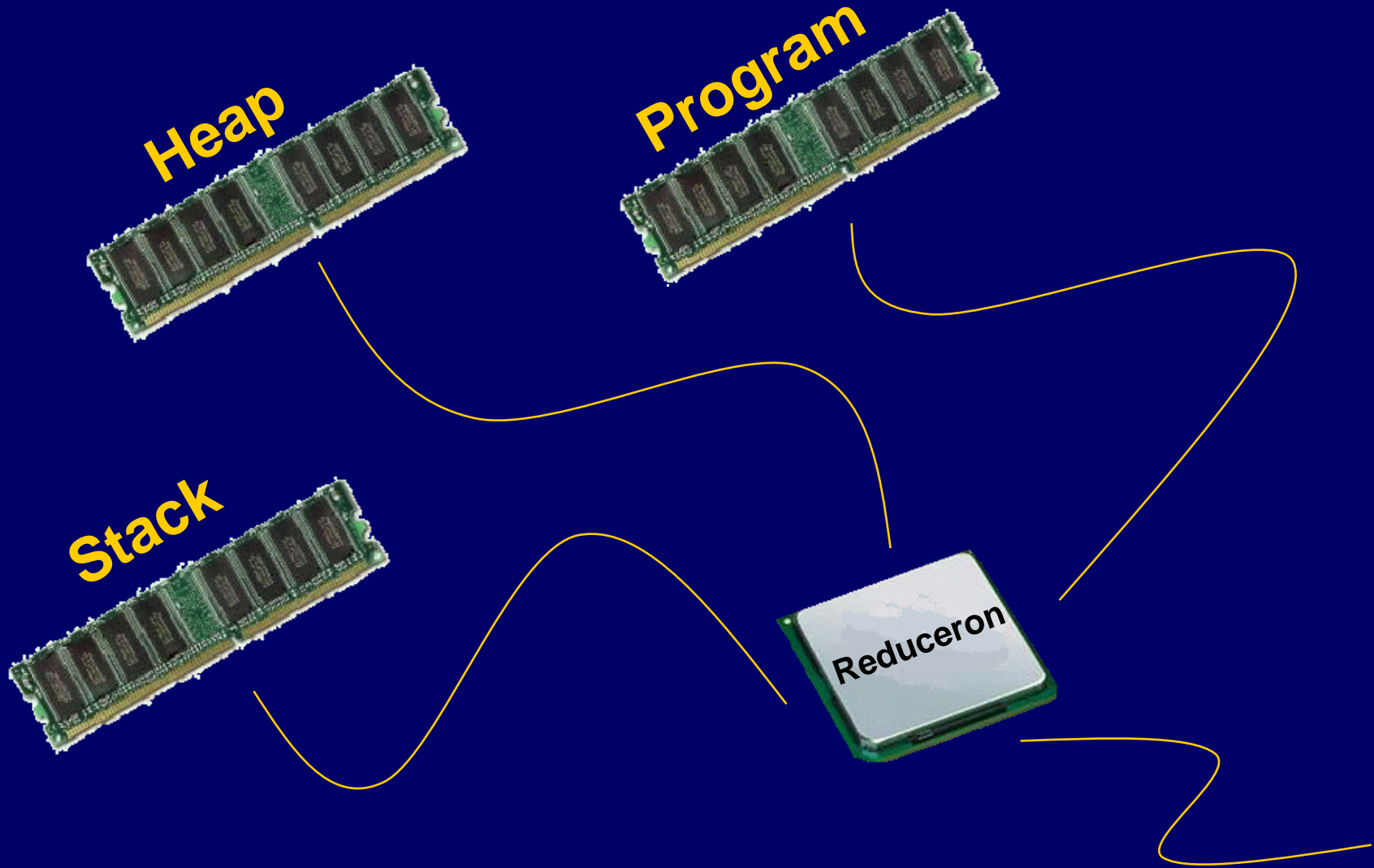


One word at a time.

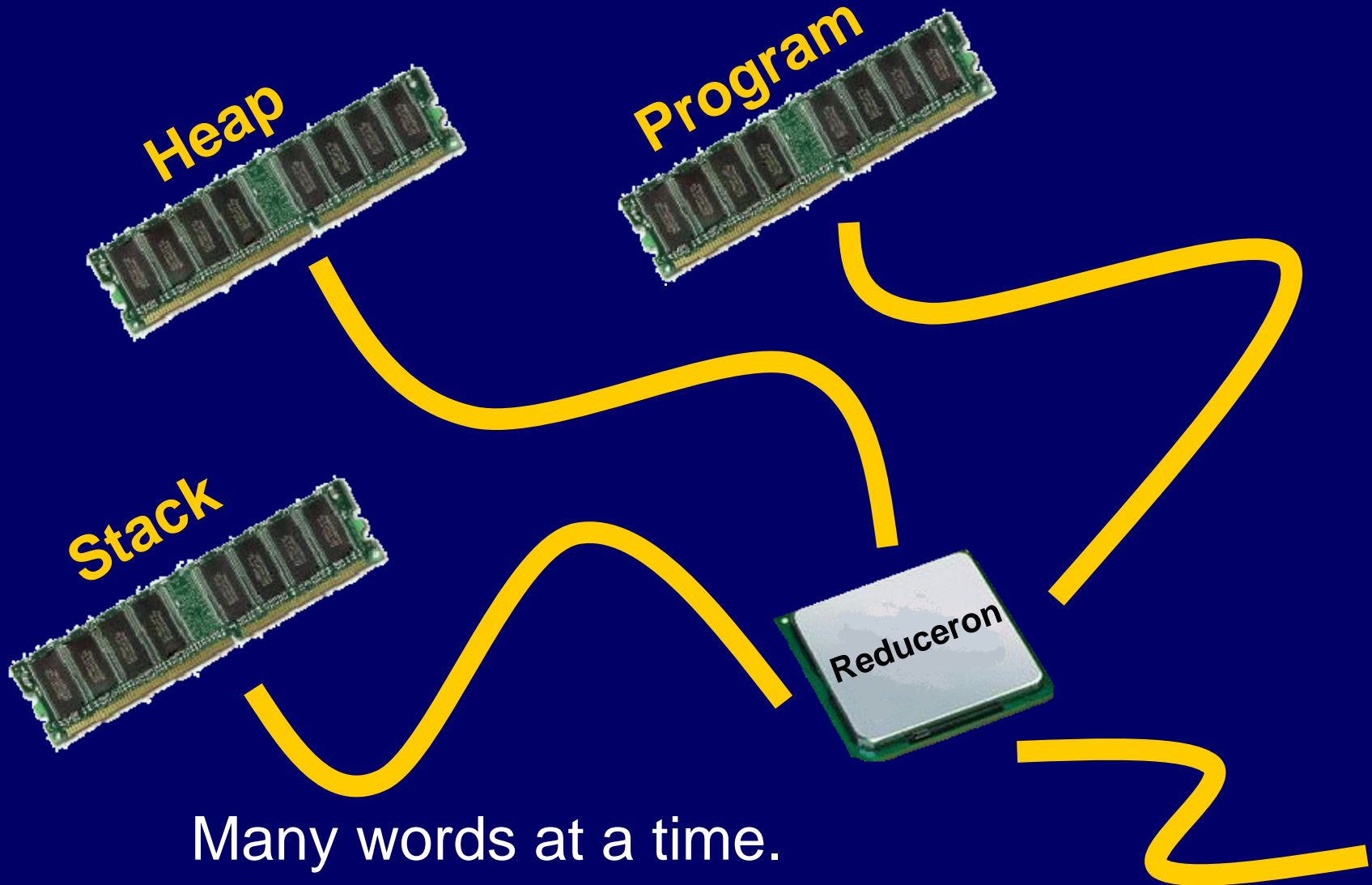
Each of the **16** memory transactions is done sequentially.



Widening the Bottleneck

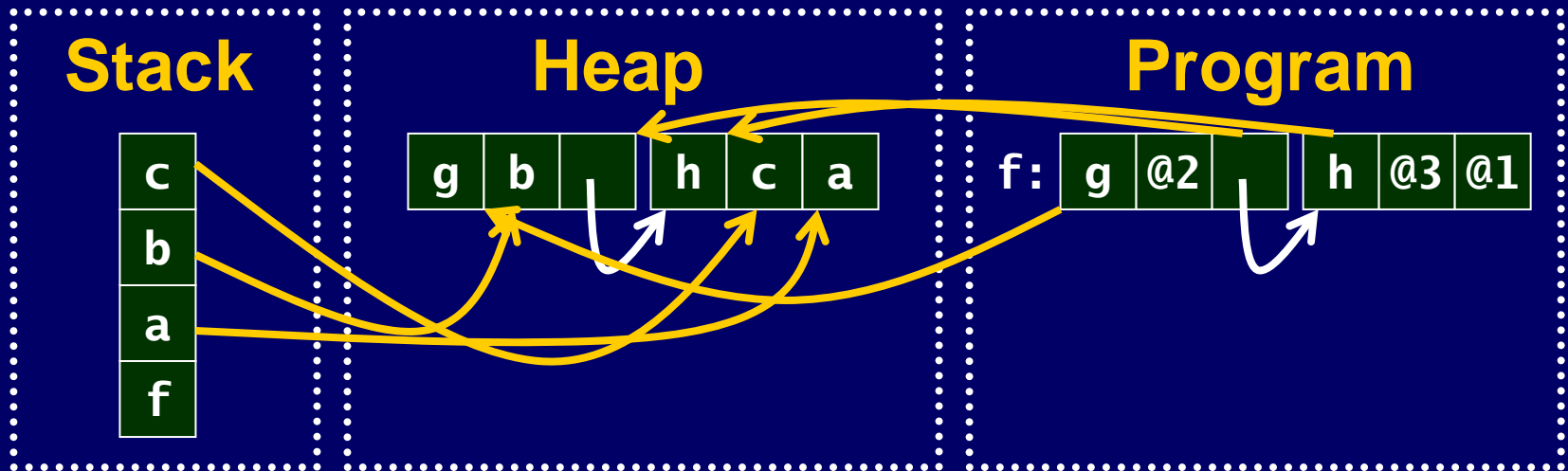


Widening the Bottleneck, again



Many words at a time.

Applying a function “in one go”



Reduceron, September 2008

Operation	Clock cycles
Apply	$3 + \lfloor n/8 \rfloor$
Unwind	2
Swap	2
Primitive Apply	3

Includes updating

Where n = number of *nodes* in function body.

Reduceron, September 2008

Wide Reduceron

(uses wide, parallel memories)

5x faster than

Narrow Reduceron

(single connection to memory)

Wide Reduceron

at 92MHz on Virtex-II FPGA

5x slower than

GHC -O2

(advanced optimising compiler)

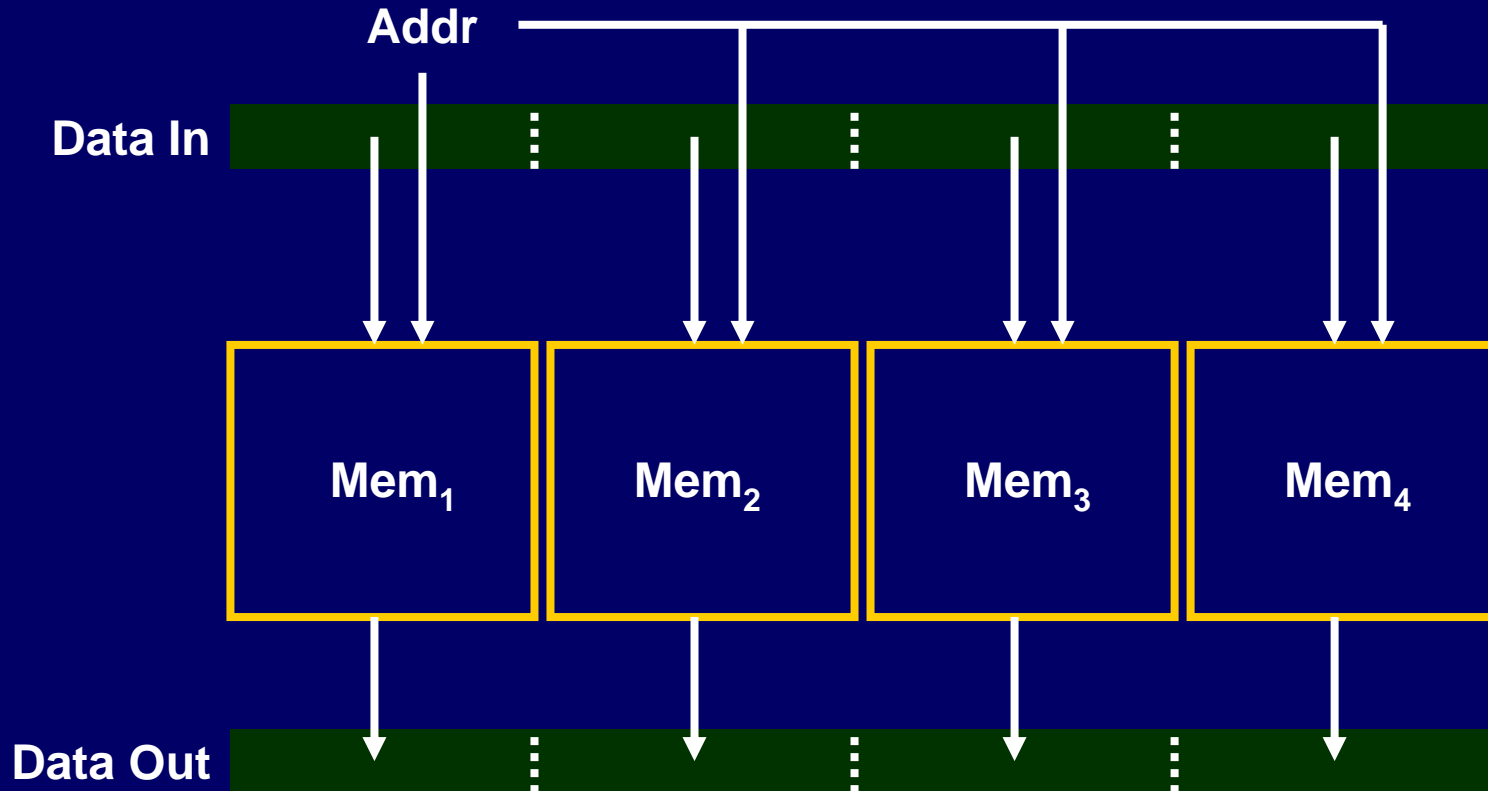
at 2800MHz on Pentium-4 PC

(On “symbolic programs”.)

Improvement 1

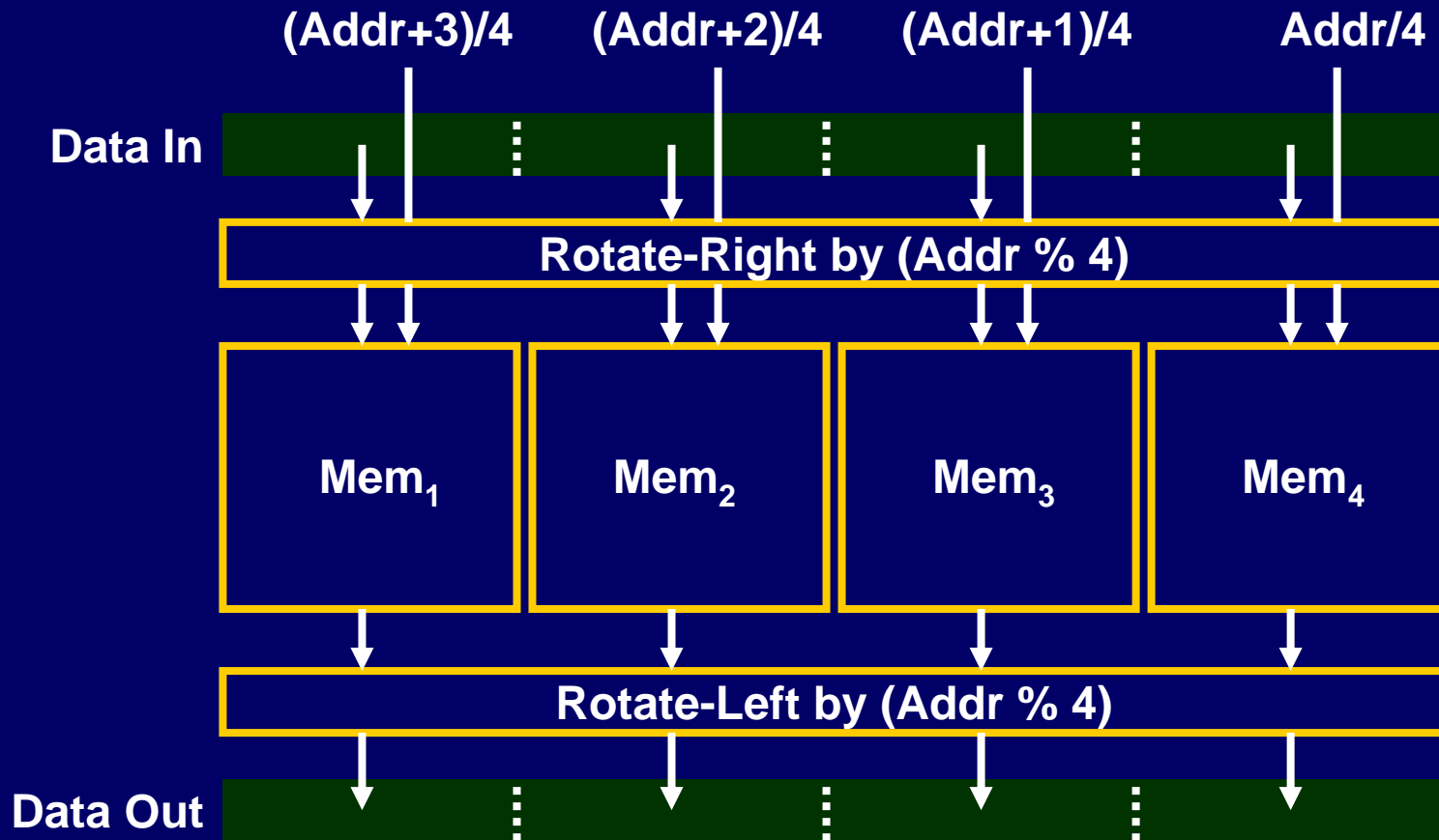
Heap and stack layout

Making a wide memory: Method 1



Cannot address individual words, *only blocks of 4.*

Making a wide memory: Method 2



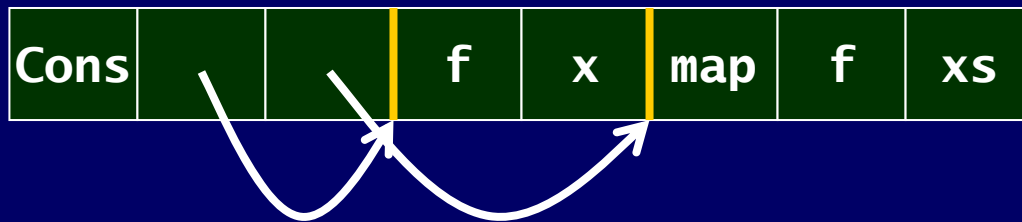
Can address *any* 4 consecutive words,
but extra logic is needed which may need buffered.

Old Heap Layout

Used Method 2, so the expression

Cons (f x) (map f xs)

was represented in memory as



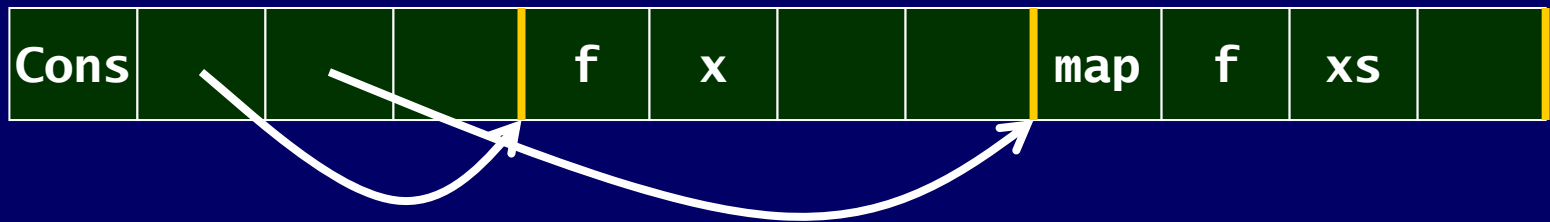
Great utilisation, but buffer on memory bus resulted in ***2-cycle reads***.

New Heap Layout

Uses Method 1, so the expression

Cons (f x) (map f xs)

is represented in memory as



Poor utilisation, but allows ***1-cycle reads***.

Also permits updating without indirections.

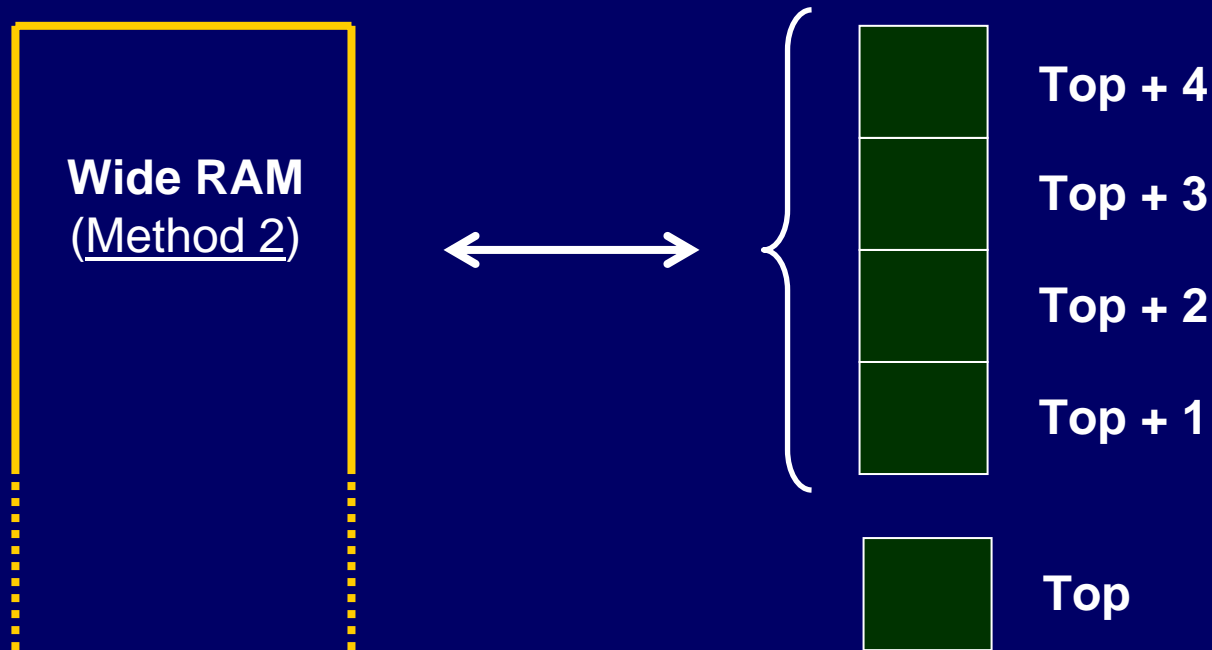
Old Stack Layout

Stack cannot have gaps, so *must* use Method 2!

Now 2 cycles are needed to read from stack...

New Stack Layout

But top elements can be stored in registers.



Top elements can be read in 0 cycles.

(This is the critical path in my current design – suggestions welcome!)

New clock counts

Using new heap/stack layout (& spineless reduction).

Operation	Clock cycles
Apply	$\lceil n/2 \rceil$
Unwind	1
Update	1
Swap	1
Primitive Apply	1

Where **n** = number of *applications* in function body.

Clock frequency not affected.

Improvement 2

Dealing with case expressions

Case expressions

In general

case e **of**

$C_1 \ v_1 \dots v_{\#C_1} \rightarrow e_1$

\vdots

$C_n \ v_1 \dots v_{\#C_n} \rightarrow e_n$

Example

app $xs \ ys =$

case xs **of**

Cons $v_1 \ v_2 \rightarrow$

Cons $v_1 \ (\text{app } v_2 \ ys)$

Nil $\rightarrow ys$

Case expressions

In general

case e **of**

$C_1 \ v_1 \dots v_{\#C_1} \rightarrow e_1$

\vdots

$C_n \ v_1 \dots v_{\#C_n} \rightarrow e_n$

Example

app xs ys =

case xs **of**

Cons $v_1 \ v_2 \rightarrow$

Cons v_1 (**app** v_2 **ys**)

Nil \rightarrow **ys**

Free variable with respect to **case** expression.

Case elimination, part 1

(The Scott/Parigot/Jansen/... encoding.)

case e of

$C_1 \ v_1 \dots v_{\#C_1} \rightarrow e_1$

\vdots

$C_n \ v_1 \dots v_{\#C_n} \rightarrow e_n$



$e \ (a\lambda t_1 \ \nu(e_1))$

\vdots

$(a\lambda t_n \ \nu(e_n))$

where

$a\lambda t_1 \ \nu(e_1) \ v_1 \dots v_{\#C_1} = e_1$

\vdots

$a\lambda t_n \ \nu(e_n) \ v_1 \dots v_{\#C_n} = e_n$

$\nu(e)$ denotes the *free variables* in e .

Case elimination, part 1, example

```
app xs ys =  
  case xs of  
    Cons v0 v1 -> Cons v0 (app v1 ys)  
    Nil -> ys
```



```
app xs ys = xs (alt1 ys) ys  
alt1 ys v0 v1 = Cons v0 (app v1 ys)
```

Case elimination, part 2

(The Scott/Parigot/Jansen/... encoding.)

For each constructor C_i , introduce function

$$C_i \ v_1 \dots v_{\#C_i} \ k_1 \dots k_n = k_i \ v_1 \dots v_{\#C_i}$$

For example, the list constructors:

$$\mathbf{Nil} \ n \ c = n$$

$$\mathbf{Cons} \ x \ xs \ n \ c = c \ x \ xs$$

Case elimination, bigger example

```
data Exp = X
         | Y
         | Neg Exp
         | Add Exp Exp
         | Sub Exp Exp
```

```
eval x y e =
```

```
  case e of
```

```
    X -> x
```

```
    Y -> y
```

```
    Neg n -> 0 - eval x y n
```

```
    Add n m -> eval x y n + eval x y m
```

```
    Sub n m -> eval x y n - eval x y m
```

Case elimination, bigger example

X x y neg add sub = x

Y x y neg add sub = y

Neg n m x neg add sub = neg n

Add n m x y neg add sub = add n m

Sub n m x y neg add sub = sub n m

eval x y e = e x y (negAlt x y)
(addAlt x y)
(subAlt x y)

negAlt x y n = 0 - eval x y n

addAlt x y n m = eval x y n + eval x y m

subAlt x y n m = eval x y n - eval x y m

Large arities.

Large bodies, with repetition.

Abstraction

$e \ x \ y \ (a \ t_3 \ x \ y) \ (a \ t_4 \ x \ y) \ (a \ t_5 \ x \ y)$

$= \{ a \ t_1 \ x \ y = x, \ a \ t_2 \ x \ y = y \}$

$e \ (a \ t_1 \ x \ y) \ (a \ t_2 \ x \ y)$
 $(a \ t_3 \ x \ y) \ (a \ t_4 \ x \ y) \ (a \ t_5 \ x \ y)$

$= \{ \text{abstraction} \}$

No repetition

$e \ a \ t_1 \ a \ t_2 \ a \ t_3 \ a \ t_4 \ a \ t_5 \ x \ y$

Row of constants

Case elimination, revisited

For each case alternative, introduce function

$$\text{alt } t_i \ v_1 \dots v_{\#C_i} \ \mathcal{V}(e_1 \dots e_n) = e_i$$

Transform each **case** expression to

$$e \ \text{alt } t_1 \ \mathcal{V}(e_1 \dots e_n) \quad (\text{Case alts are } \textit{aligned} \text{ – next slide})$$

Evaluate constructor C_i to function at address

$$\text{alt } t_1 \ + \ (i-1)$$

↑

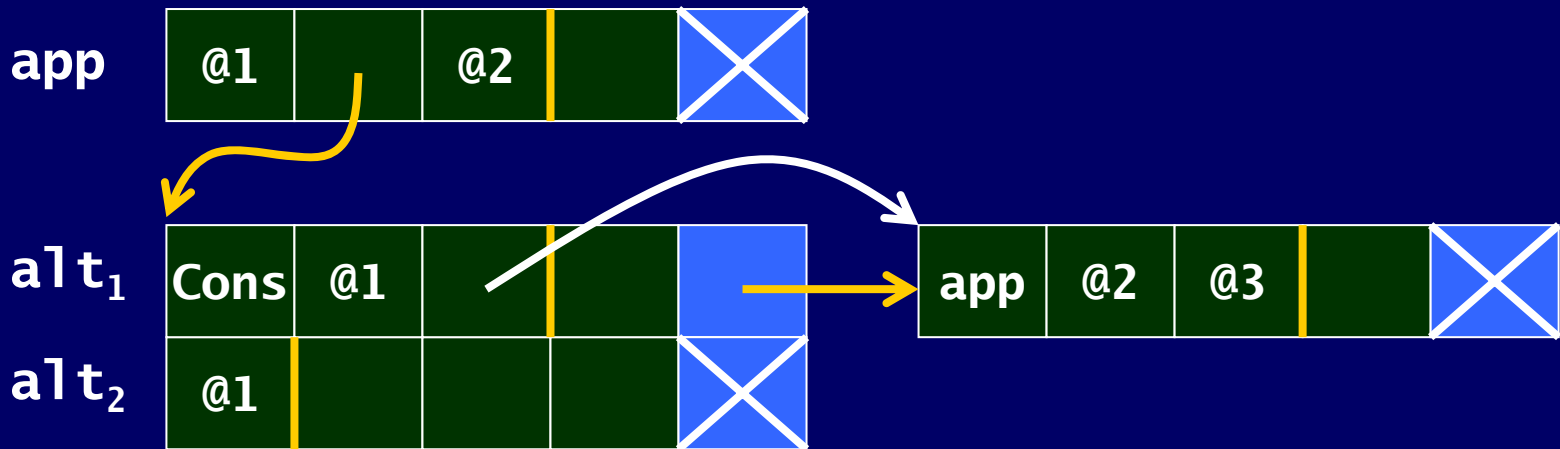
Simple addition - can be computed in 0 cycles?

Jumping in 0 cycles, part 1

The code for **app** (list append)

```
app xs ys      = xs alt1 ys
alt1 x xs ys = Cons x (app xs ys)
alt2 ys      = ys
```

is represented in memory as follows.



Jumping in 0 cycles, part 2

Evaluate constructor C_i to function at address

$$a\uparrow t_1 + (i-1)$$


Not such a simple addition!

Must fetch $a\uparrow t_1$ from stack, $\#C_i$ places from the top

Solution: store $a\uparrow t_1$ on a separate (parallel) stack.

Clock frequency not affected.

Improvement 3

Dynamic update avoidance

Shared applications

Distinguish between

- *unshared* applications, and
- *possibly-shared* applications.

Idea: When an unshared application is reduced to normal form, no update is needed.

Dynamic v. Static Analysis

*“Create all closures as [unshared], and dynamically change their tag to [possibly-shared] if they become shared. We call this operation **dashing**.”*

*“In general we strongly suspect that the cost of **dashing** greatly outweighs the advantages of precision when compared to the [static analysis] method.”*

--- Simon Peyton Jones

Dashing when applying

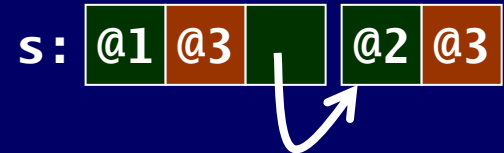
Before

Stack



Heap

Program



After

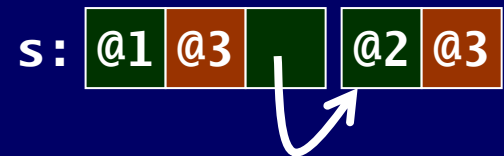
Stack



Heap

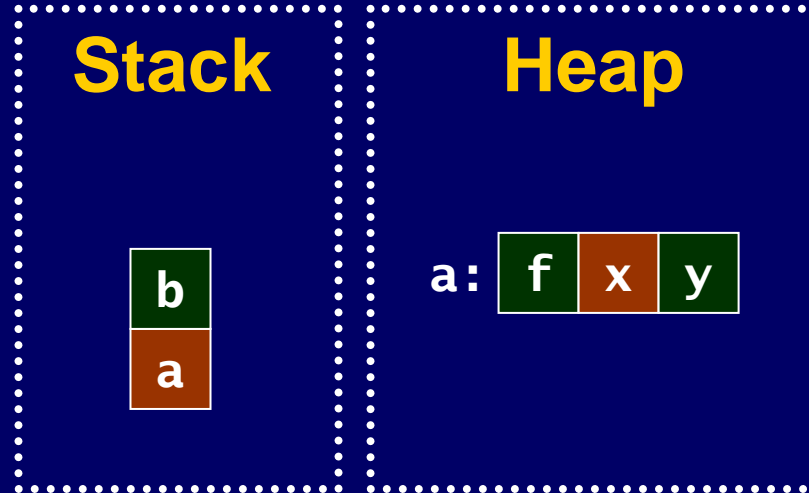


Program

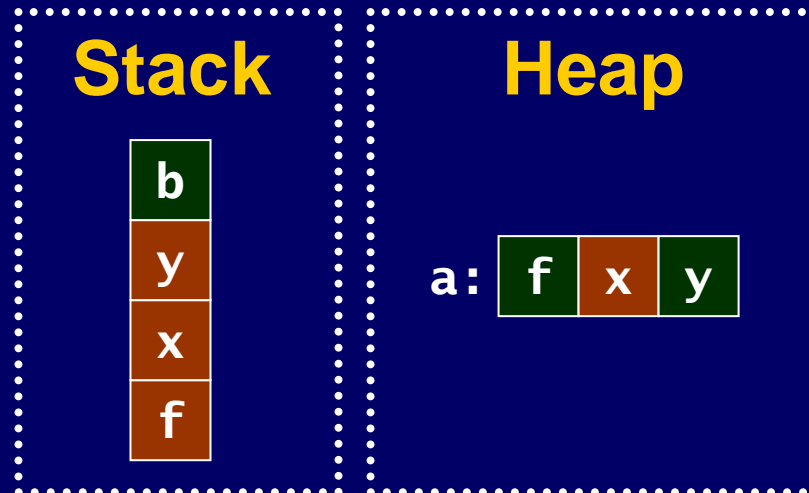


Dashing when unwinding

Before



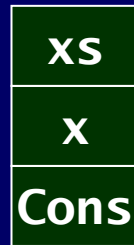
After



Dashing when updating

Before

Stack



Heap



After

Stack



Heap



Dynamic v. Static Analysis

In the Reduceron, dynamic update avoidance is rather cheap – it's just bit-flipping under some simple-to-compute conditions.

Clock frequency not affected.

How the Reduceron is described

Description language

- York Lava
 - Multi-output primitives
 - RAMs
 - Modular - easy to add new primitives and back-ends
 - Behavioural description
 - Statically-sized bit-vectors

My dream...

Compile small-step machine semantics directly to efficient hardware.

For example, here's the unwind rule from a structural operational semantics.

$$\begin{array}{l} \langle \text{APP } \text{addr} : s, u, h, c \rangle \\ \text{-->} \quad \langle (h! \text{addr}) ++ s, (\text{length } s, \text{addr}) : u, h, c \rangle \end{array}$$

But for now...

```
unwind top s u h c doUpdate =
  do top <== n
    vpush len ns s
    upush (mkUpdate (stackSize s)
              (apAddr $ val top)) u
  doUpdate <== (arity n |>| len)
  cread (funAddr n) c
  hreadB (apAddr n) h
  tick
where
  app = heapOutB h
  len = appArity app
  (n:ns) = appNodes app
```

Argh –pre-fetching and pre-computing!

Preliminary results and our to-do list

Performance improvement

Program	Speed-up
Queens	2.1
Queens ₂	2.9
PermSort	2.9
MSS	2.7
PropInsert	3.0
Sudoku	4.0
Adjoxo	3.1
While	2.8
Clausify	3.6
Average	3.0

To-do list

- Critical path reduction
- Parallel garbage collection (low or high-level?)
- Compile-time optimisation
 - Supercompilation (Neil Mitchell, Jason Reich)
- Speculative evaluation of primitive redexes
- Multi-core Reduceron
- Relax memory restrictions
 - large, off-chip heap
- Efficient hardware from small-step semantics?