Dynamic analysis in the Reduceron

Matthew Naylor and Colin Runciman
University of York

A question

“I wonder how popular Haskell needs to become for Intel to optimize their processors for my runtime, rather than the other way around.”

Simon Marlow, 2009

Underlying problem?

Constraints imposed by conventional processors make efficient implementations of functional languages ***sophisticated*** and ***limited***.

Possible solution

Build a machine especially to run functional programs.

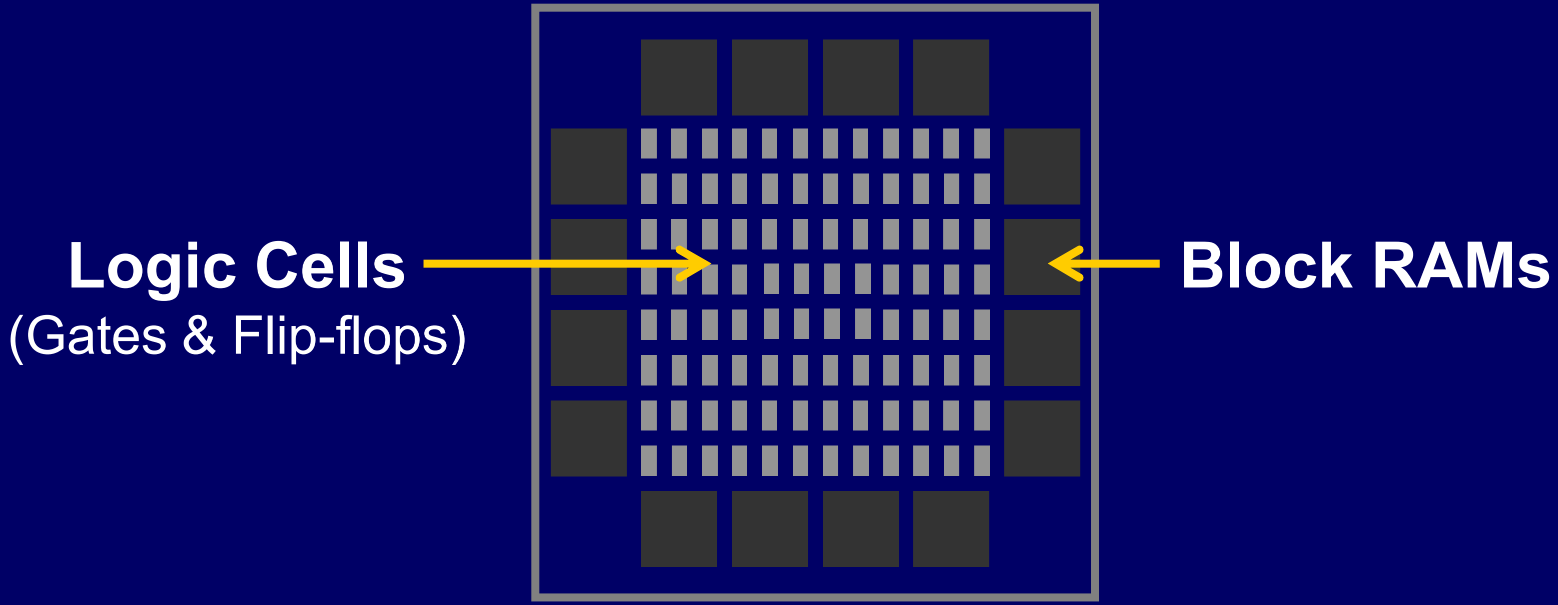
> *"Current RISC technology will probably have increased in speed enough by the time a graph-reduction chip could be designed and fabricated to make the exercise pointless."*

Koopman, 1990

But now the situation is different…

FPGAs (Reconfigurable H/W)

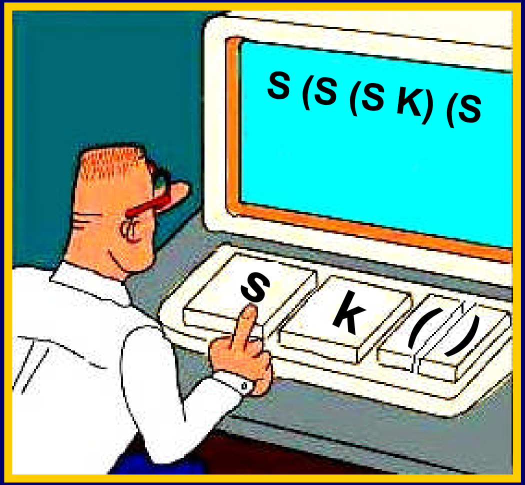
Contain a large set of components that can be connected together in any desired way.



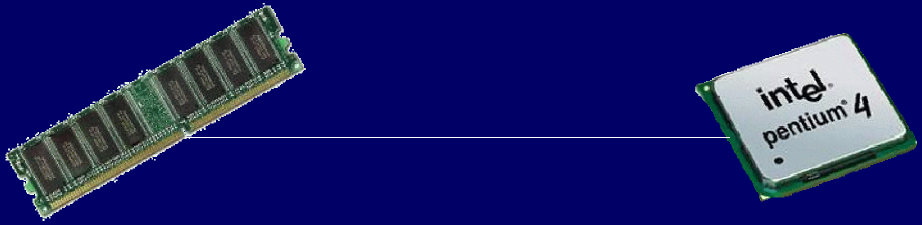
Widely available, quick to program, autonomous.

The Reduceron

A computer designed to run lazy functional programs,



not restricted by conventional architectural constraints,



implemented on an FPGA, using a functional language.

Graph reduction

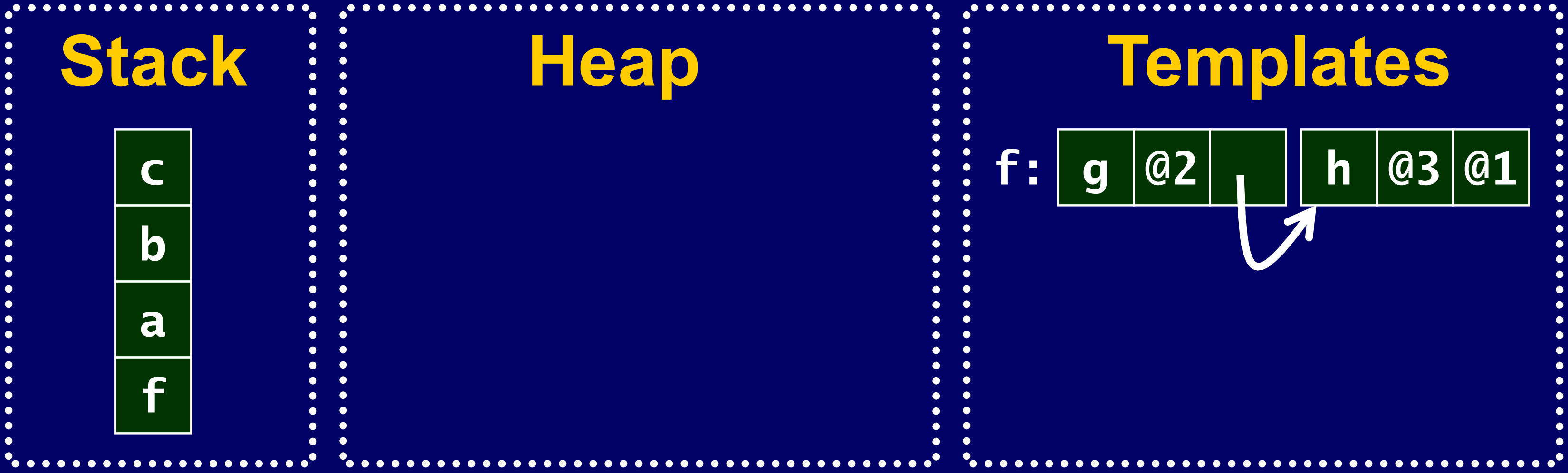
Step by step

Graph reduction, step by step

Suppose that function `f` is defined by

```
f ys x xs   =   g x (h xs ys)
```

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



Stack

c

b

a

f

Heap

Templates

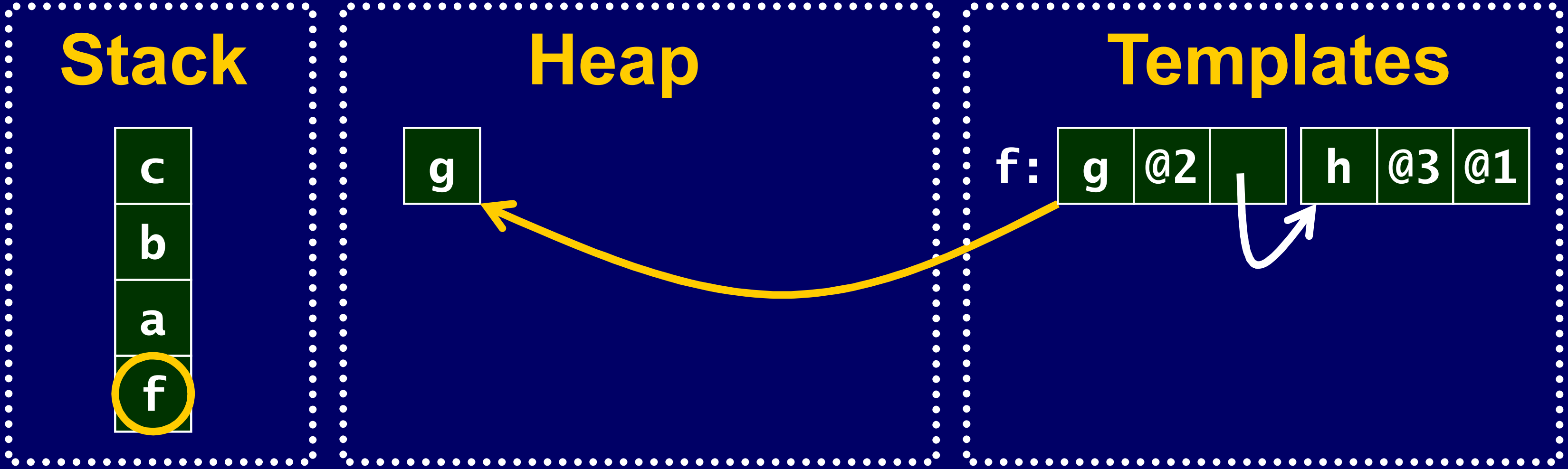
f: g @2 | h @3 @1

Graph reduction, step by step

Operation:

```
f <- Stack[0]
g <- Code[f]
g -> Heap
```

Steps: 3

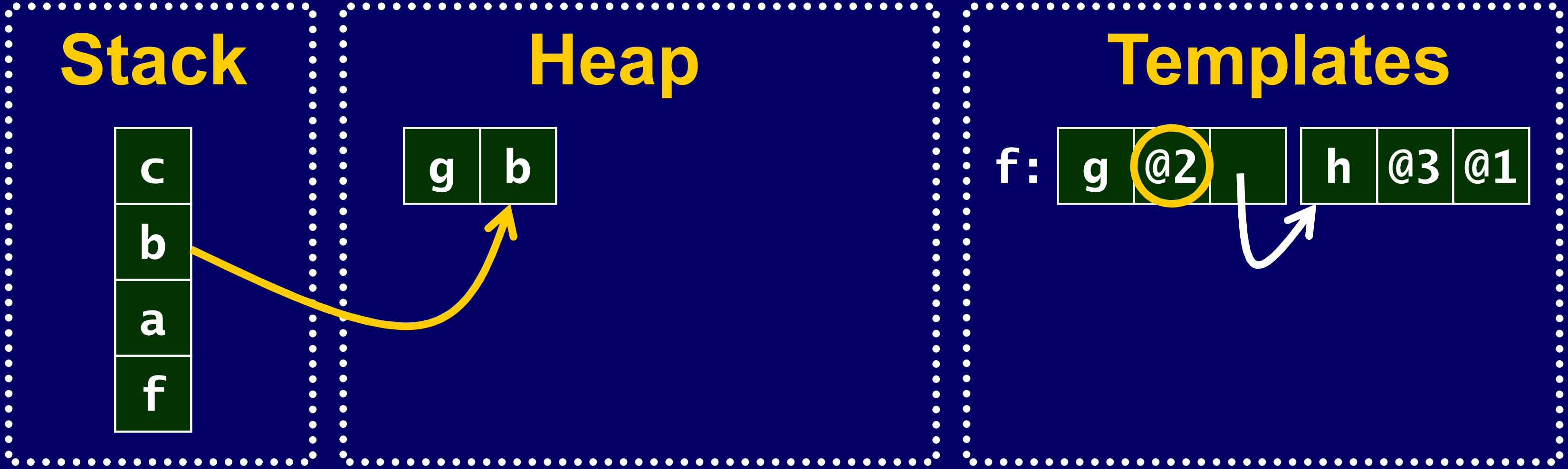


Graph reduction, step by step

Operation:

```
arg <- Code[f+1]
b   <- Stack[arg]
b   -> Heap
```

Steps: 6

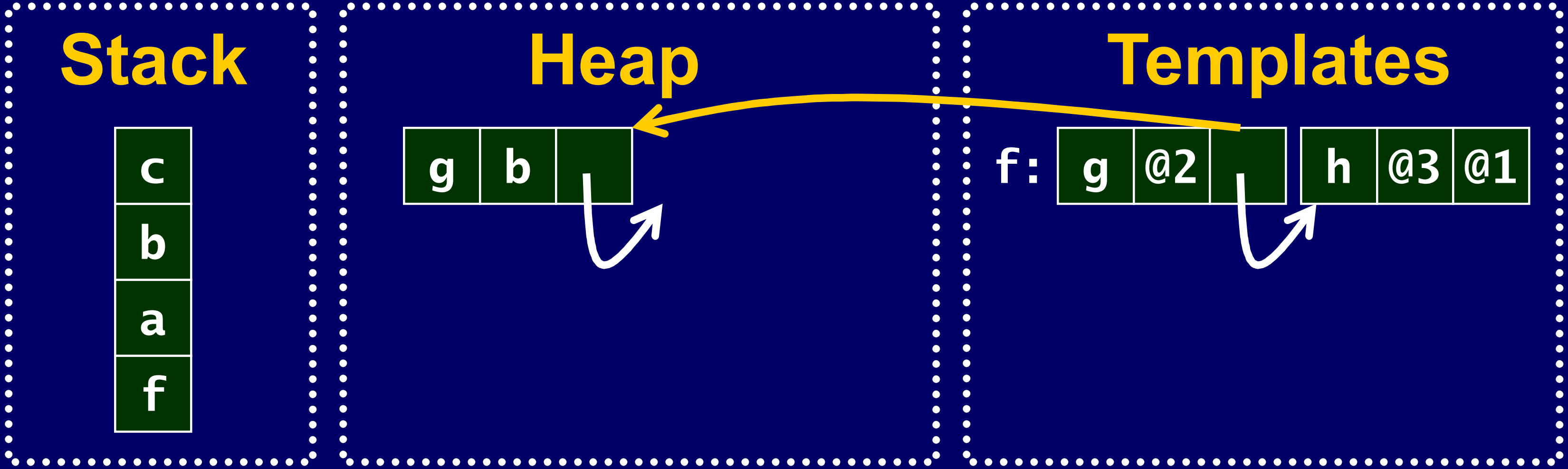


Graph reduction, step by step

Operation:

```
ptr  <- Code[f+2]
ptr’ -> Heap
```

Steps: 8

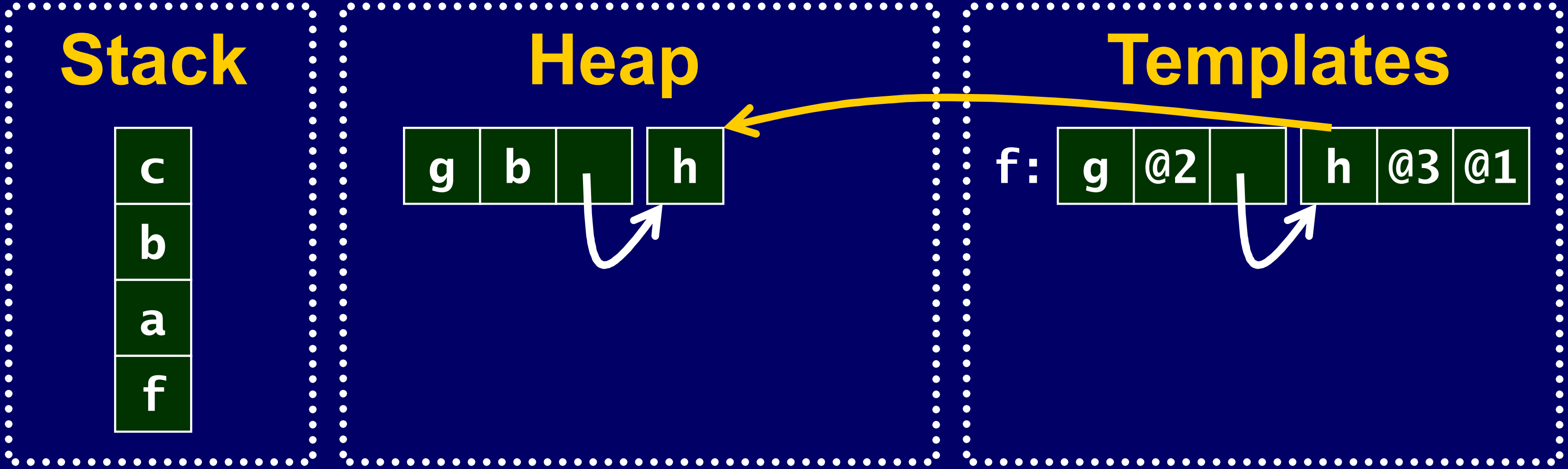


Graph reduction, step by step

Operation:

```
h <- Code[f+3]
h -> Heap
```

Steps: 10

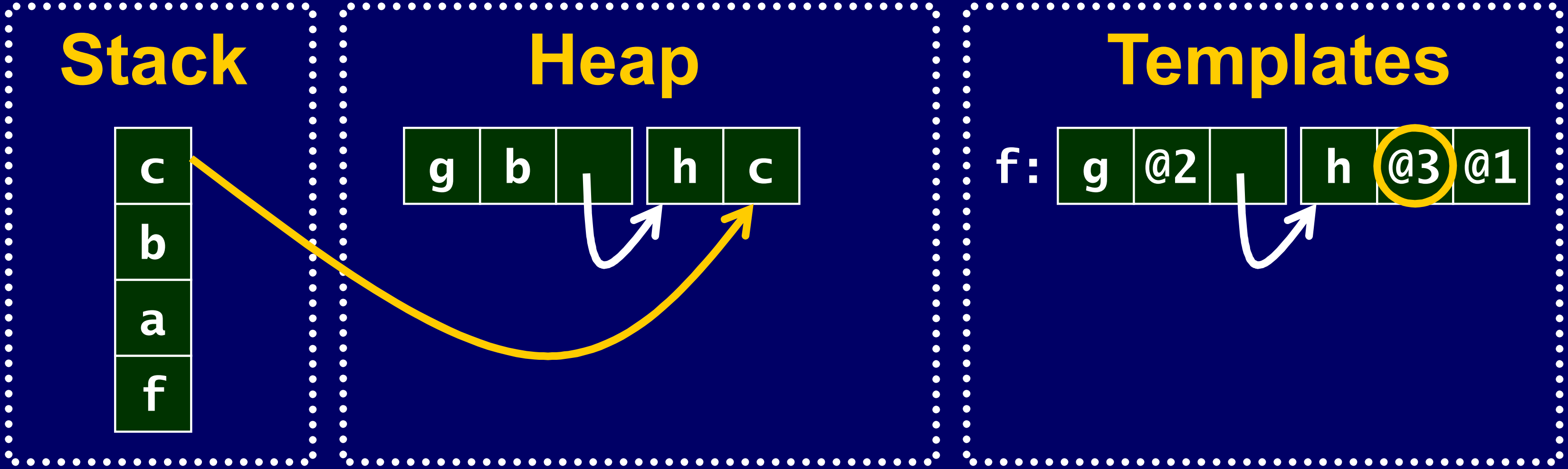


Graph reduction, step by step

Operation:

```
arg <- Code[f+4]
c   <- Stack[arg]
c   -> Heap
```

Steps: 13

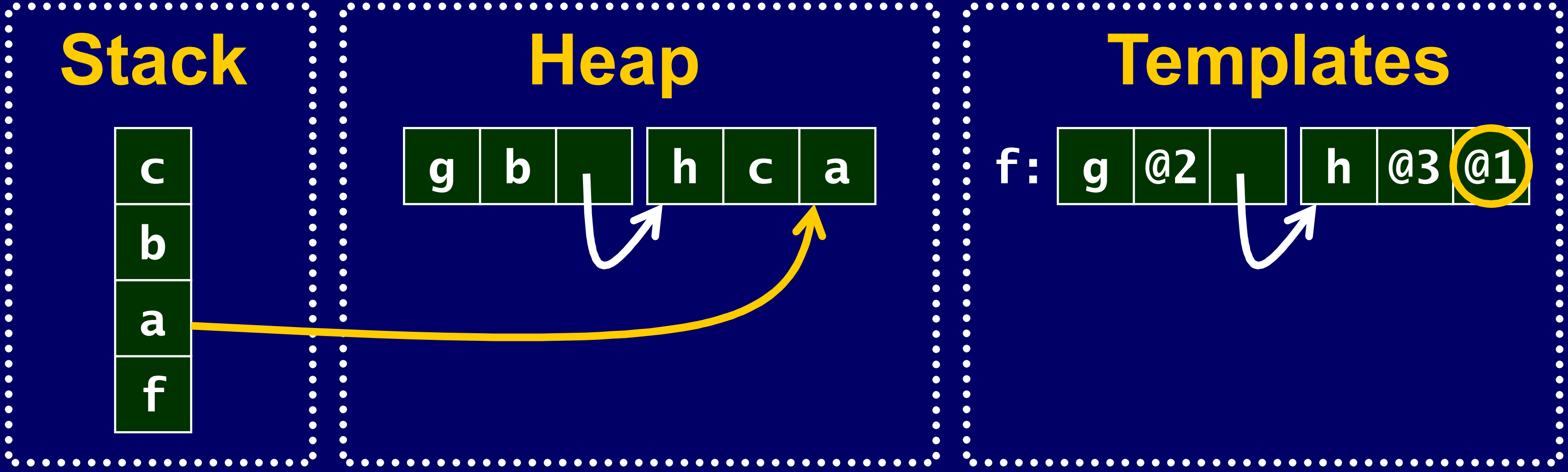


Graph reduction, step by step

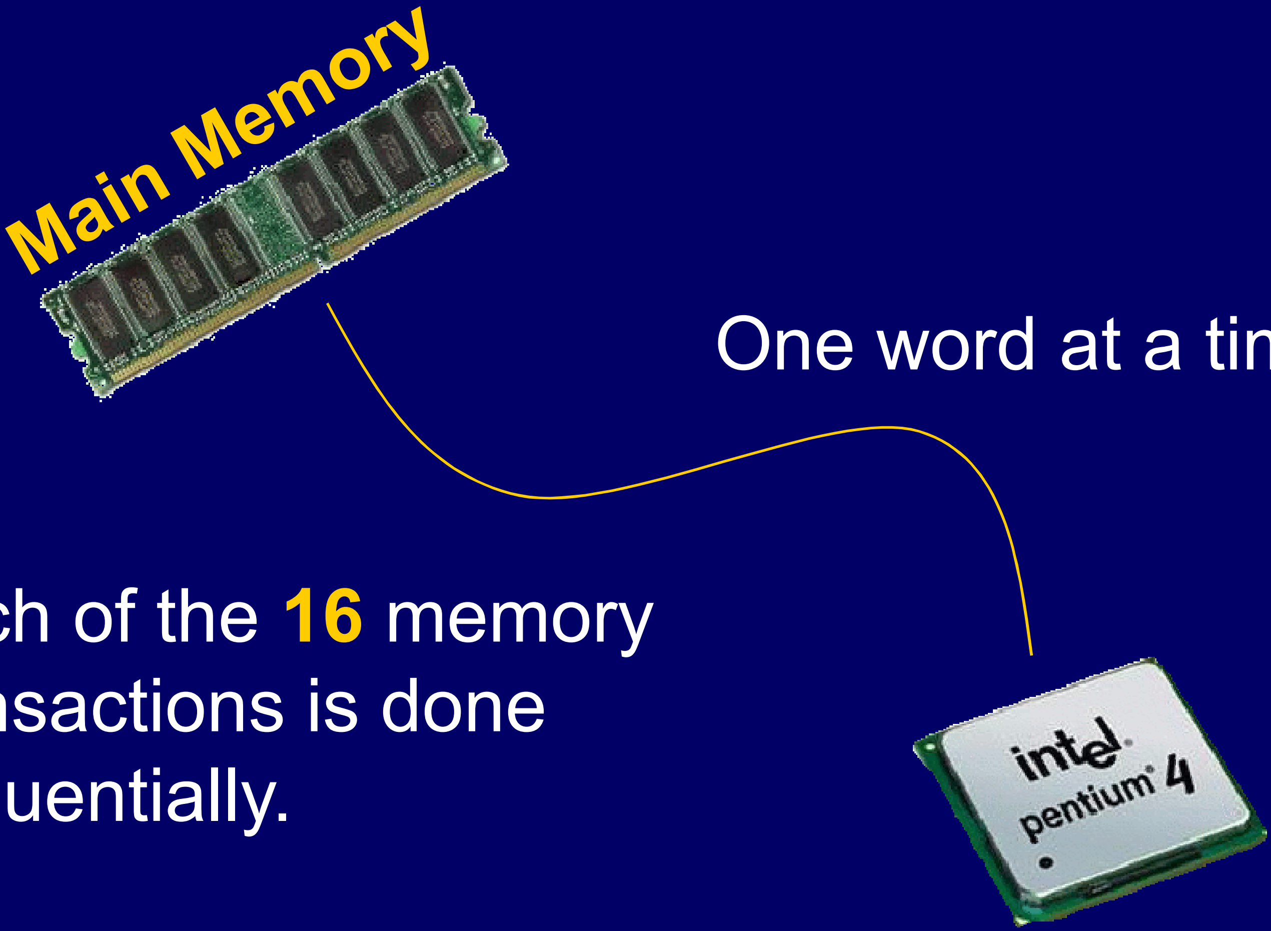
Operation:

```
arg <- Code[f+5]
a   <- Stack[arg]
a   -> Heap
```

Steps: 16



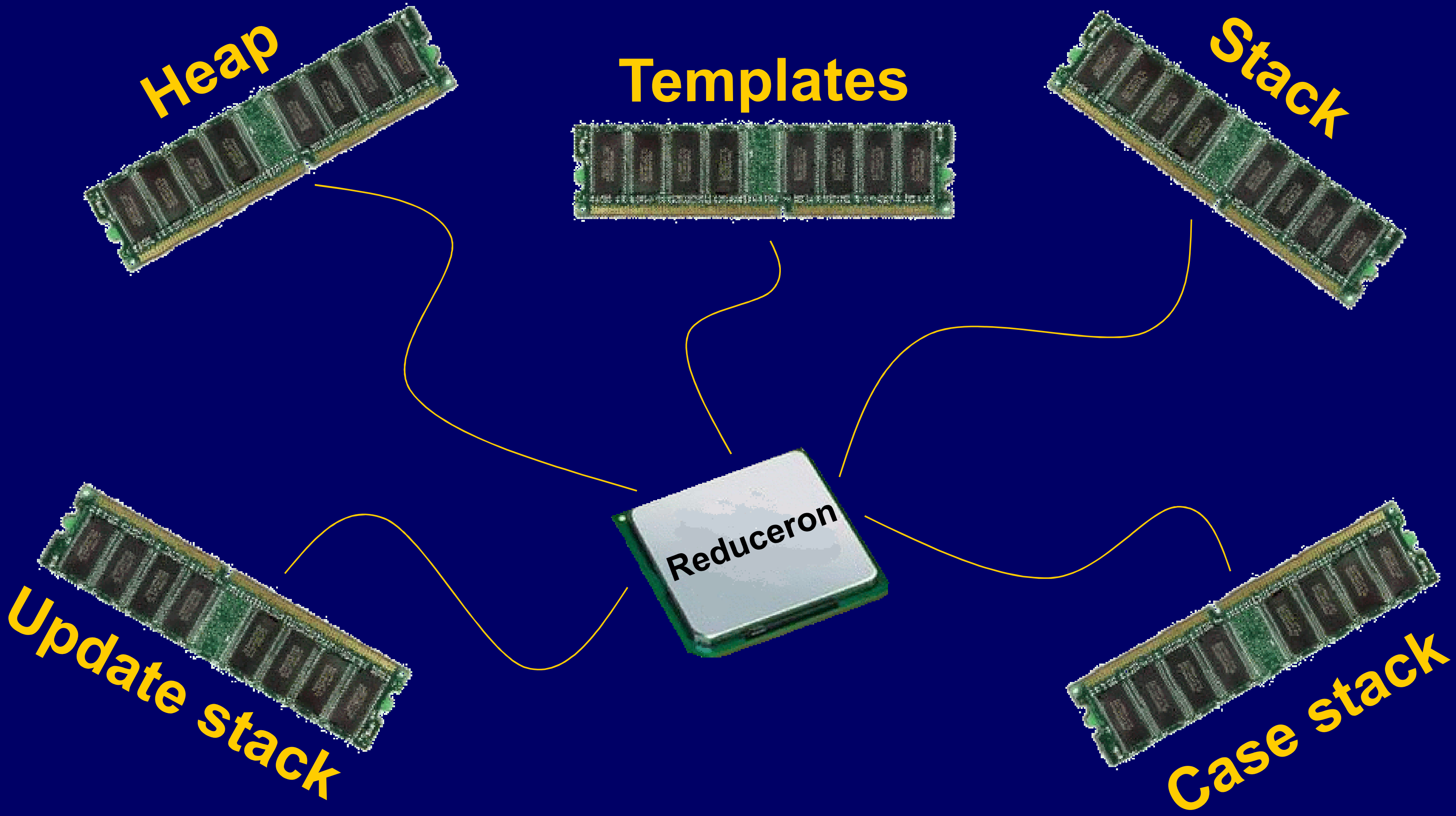
The von Neumann Bottleneck



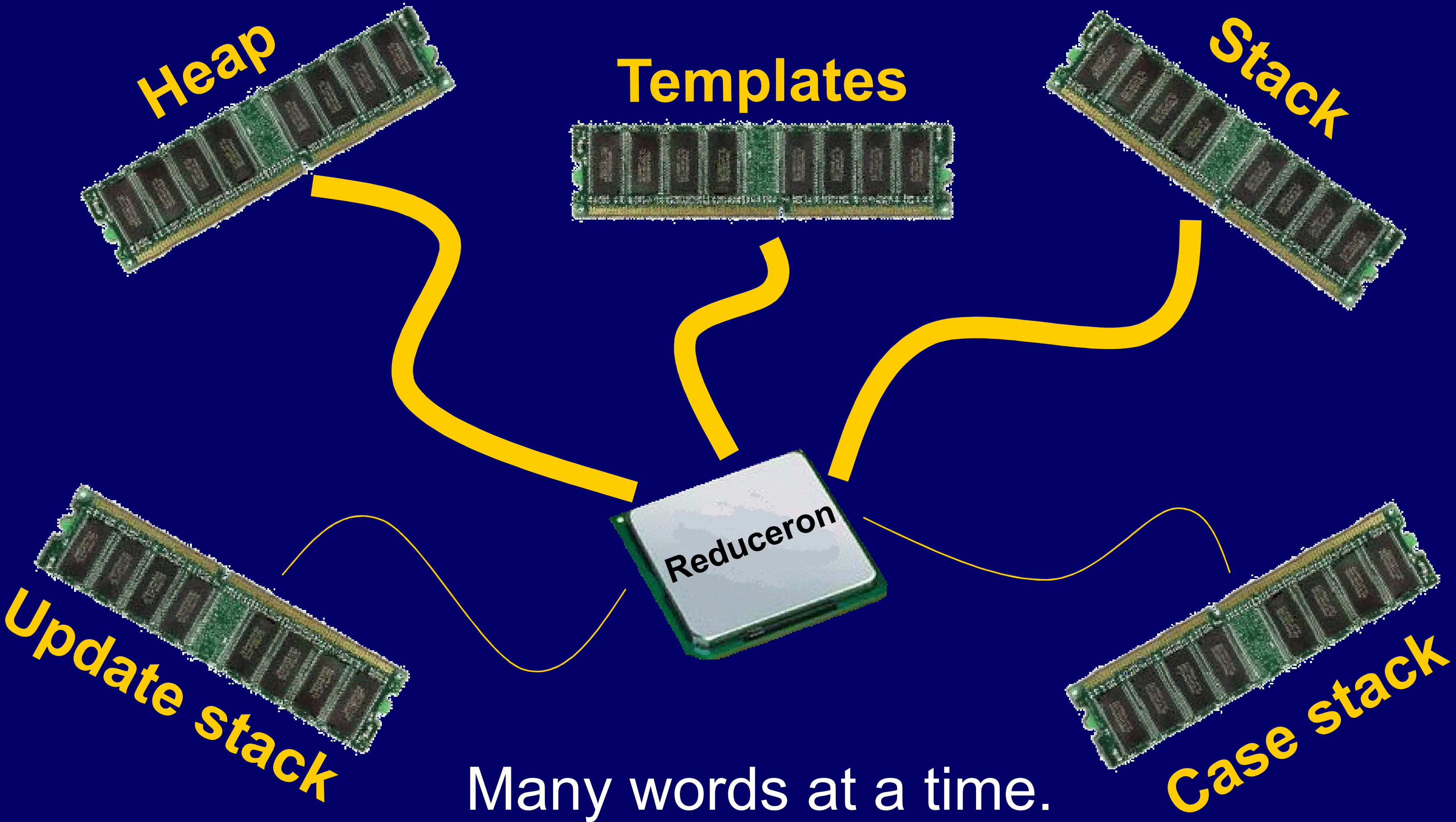
One word at a time.

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

Widening the bottleneck

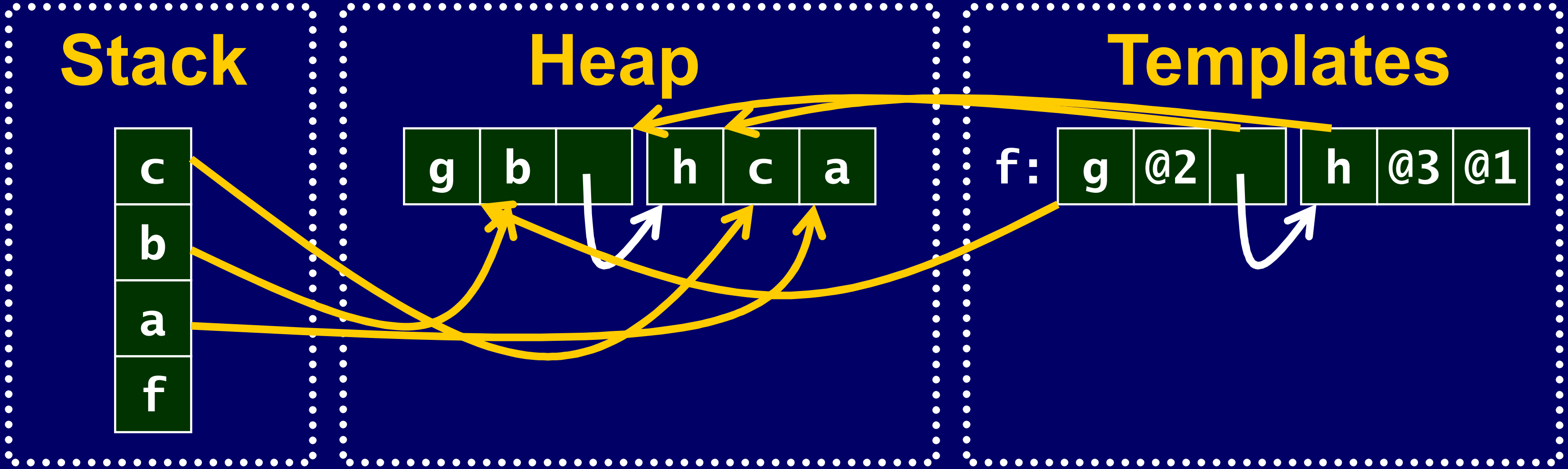


Widening the bottleneck, again



Many words at a time.

Applying a function "in one go"



The function **f** is applied in a *single clock cycle*.

Reduceron “instruction set”

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

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

Template instantiation?

Yes, the Reduceron actually does template instantiation!

But what about the G-machine and the ABC machine?

Template instantiation

> *“This chapter introduces the simplest possible implementation of a functional language: a graph-reducer based on template instantiation”*

Simon Peyton Jones, 1992

Dynamic analysis (1)

Update avoidance

Keeping track of sharing

Distinguish between:

- 🟩 *pointer to an **unshared*** application, and
- 🟫 *pointer to a **possibly-shared*** application.

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

Dynamic vs. static analysis

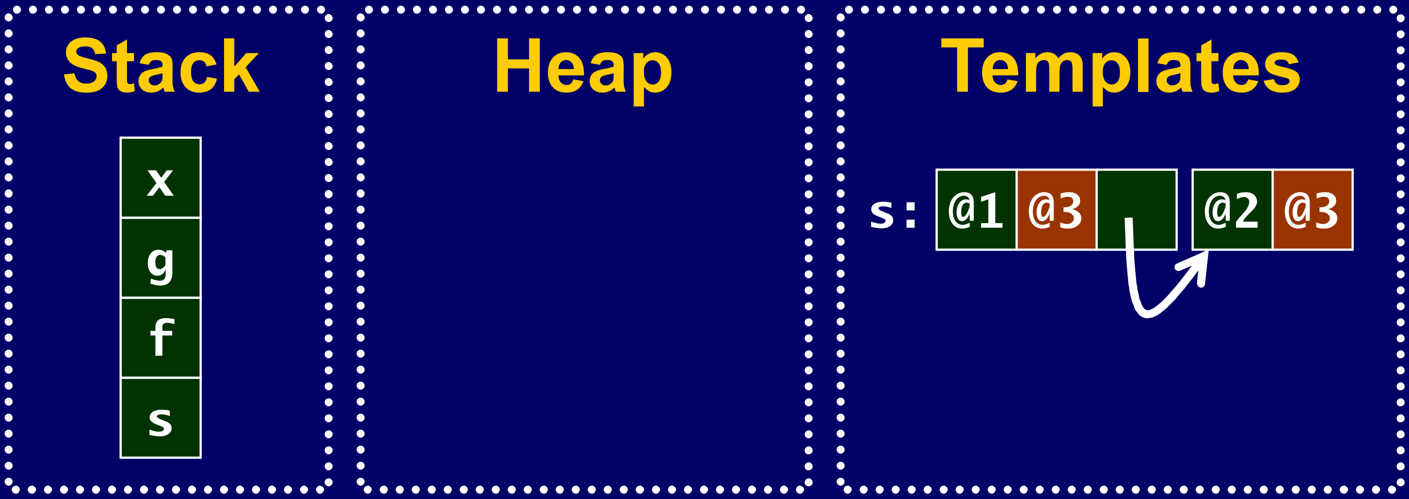
“Create all closures as [unshared], and dynamically change their tag to [possibly-shared] if they become shared. We call this operation ***dashing.****”*

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

Peyton Jones, 1988

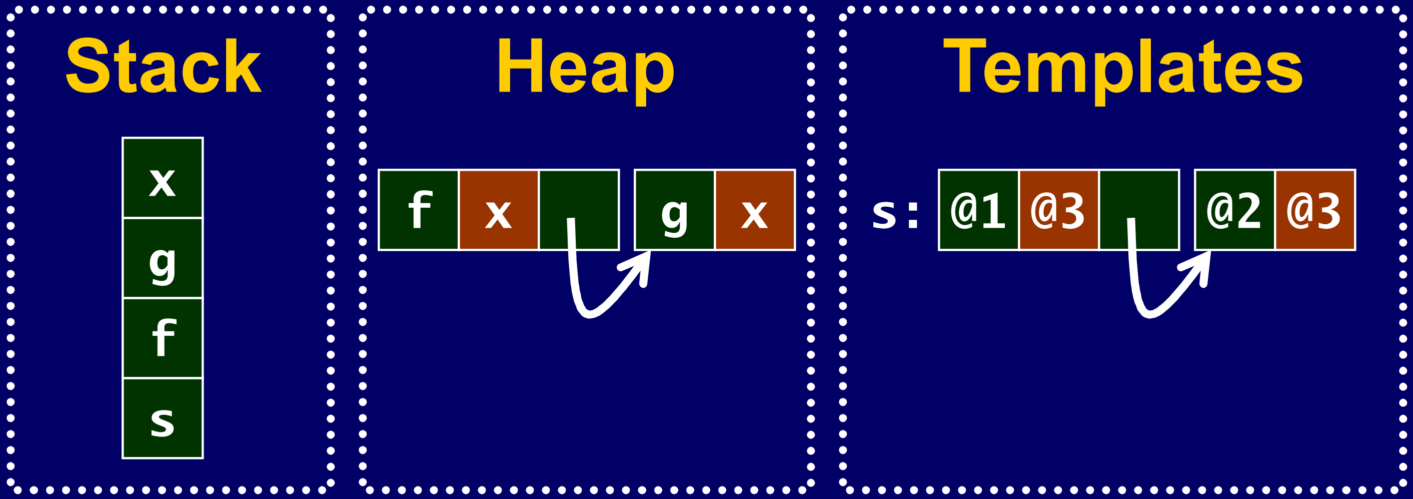
Dashing when applying

When the function **s**, containing two references to its 3rd argument, is applied,



Dashing when applying

When the function **s**, containing two references to its 3rd argument, is applied,



Stack

x

g

f

s

Heap

f x g x

Templates

s: @1 @3 @2 @3

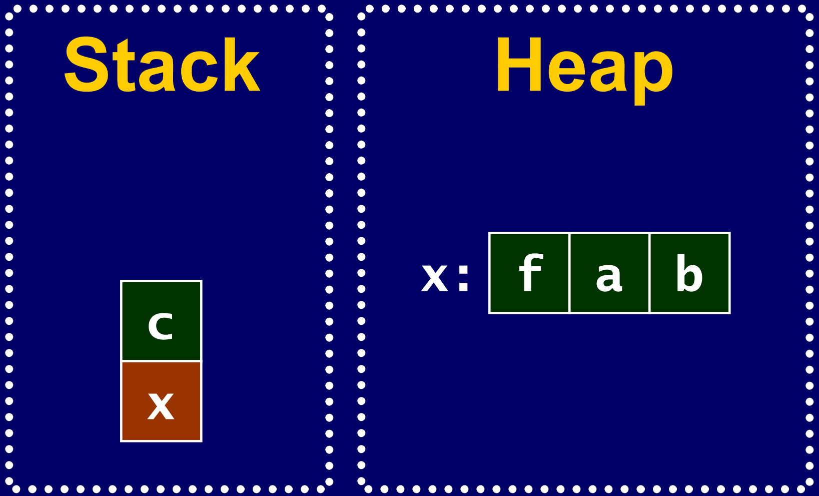
the 3rd argument is dashed.

Dashing when applying

```
inst :: Reduceron -> Atom -> Atom
inst r a =
  pick
   [ a!isARG    --> dash (a!isArgShared) x
   , a!isAP     -->
       makeAP (a!isShared)
              (r!heap!size + a!pointer)
   , otherwise --> a
   ]
  where
   otherwise = inv (a!isARG <|> a!isAP)
   x = pick (zip (a!argIndex!velems)
                 (r!vstack!tops!velems))
```

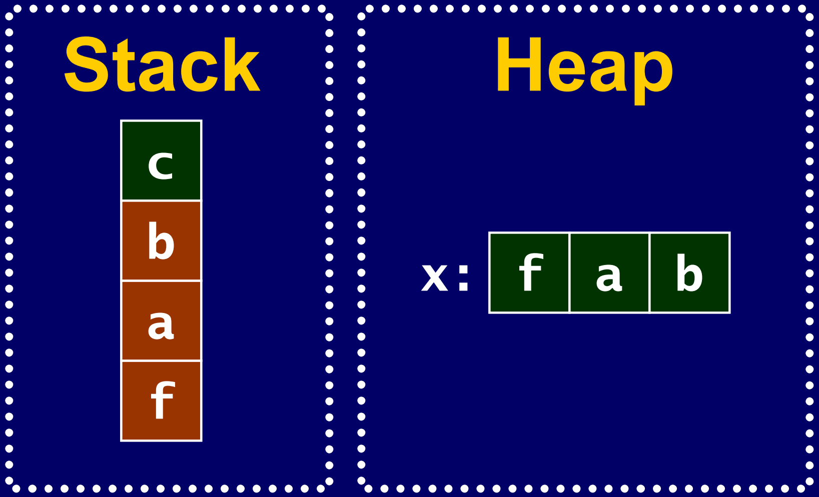
Dashing when unwinding

When a pointer **x** to a shared application appears on top of the stack,



Dashing when unwinding

When a pointer **x** to a shared application appears on top of the stack,



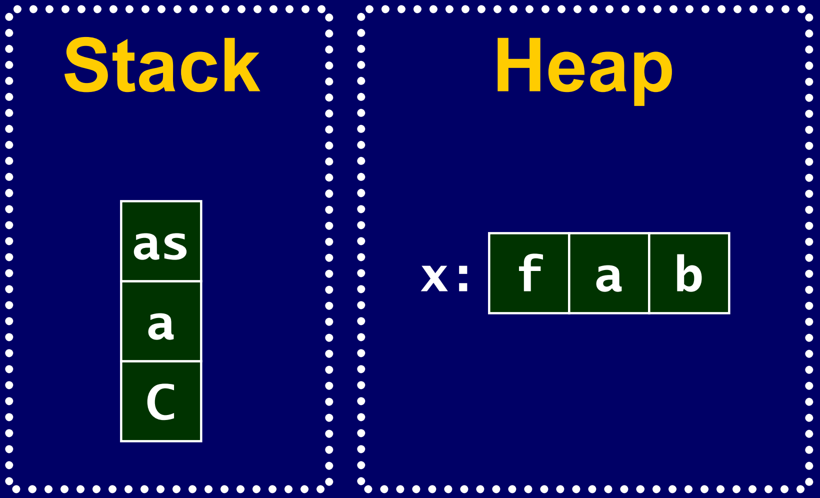
the unwound application is dashed.

Dashing when unwinding

```
unwind :: Reduceron -> Recipe
unwind r =
 Seq [
   r!newTop <== vhead as
 , r!vstack!update n (unwindMask n)
                     (as!vtail!velems)
 , app!hasAlts |> r!astack!push (app!alts)
 , upd |> r!ustack!push
           (makeUpdate (r!vstack!size)
                       (r!top!pointer))
 ]
 where
  app = r!heap!outputB
  n   = app!appArity
  as  = vmap (dash sh) (app!atoms)
  sh  = r!top!isShared
  upd = sh <&> inv (app!isNF)
```

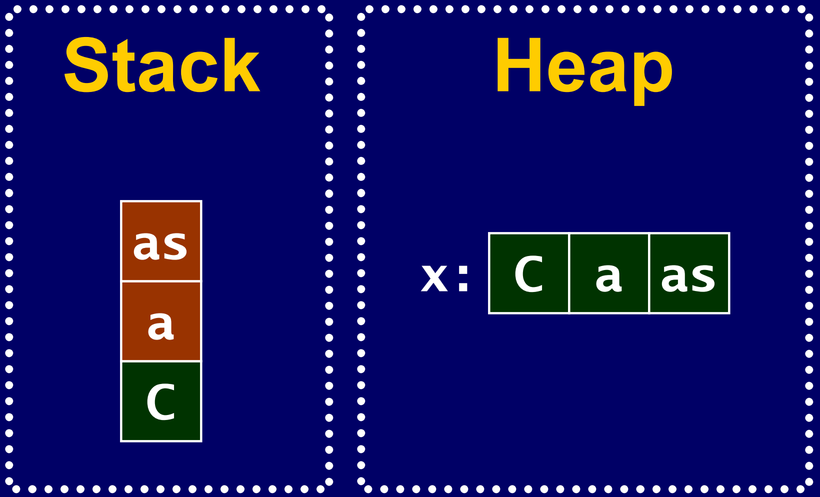
Dashing when updating

When a normal-form is reached,



Dashing when updating

When a normal-form is reached,

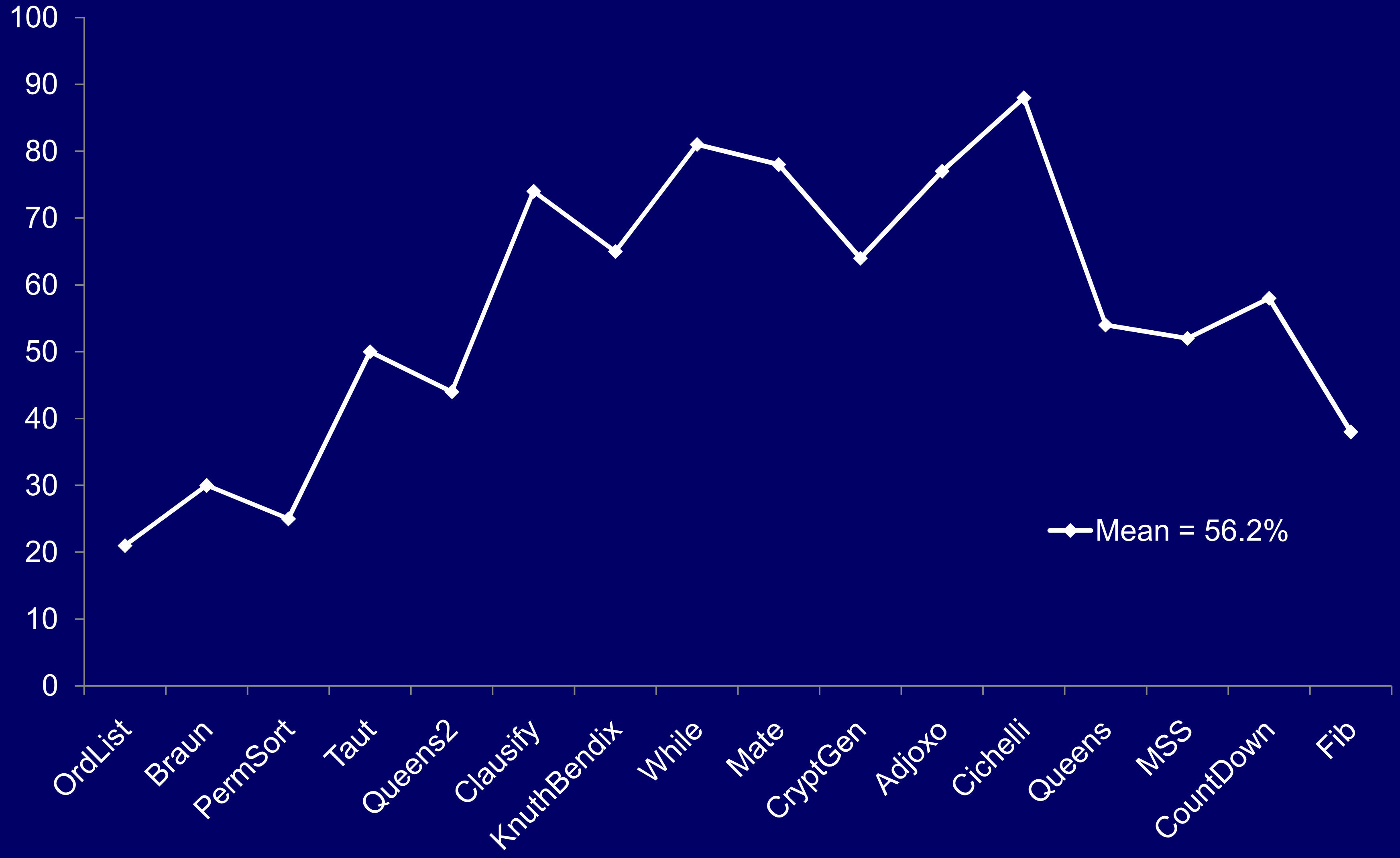


it is copied onto the heap, overwriting the original application, and its arguments are dashed.

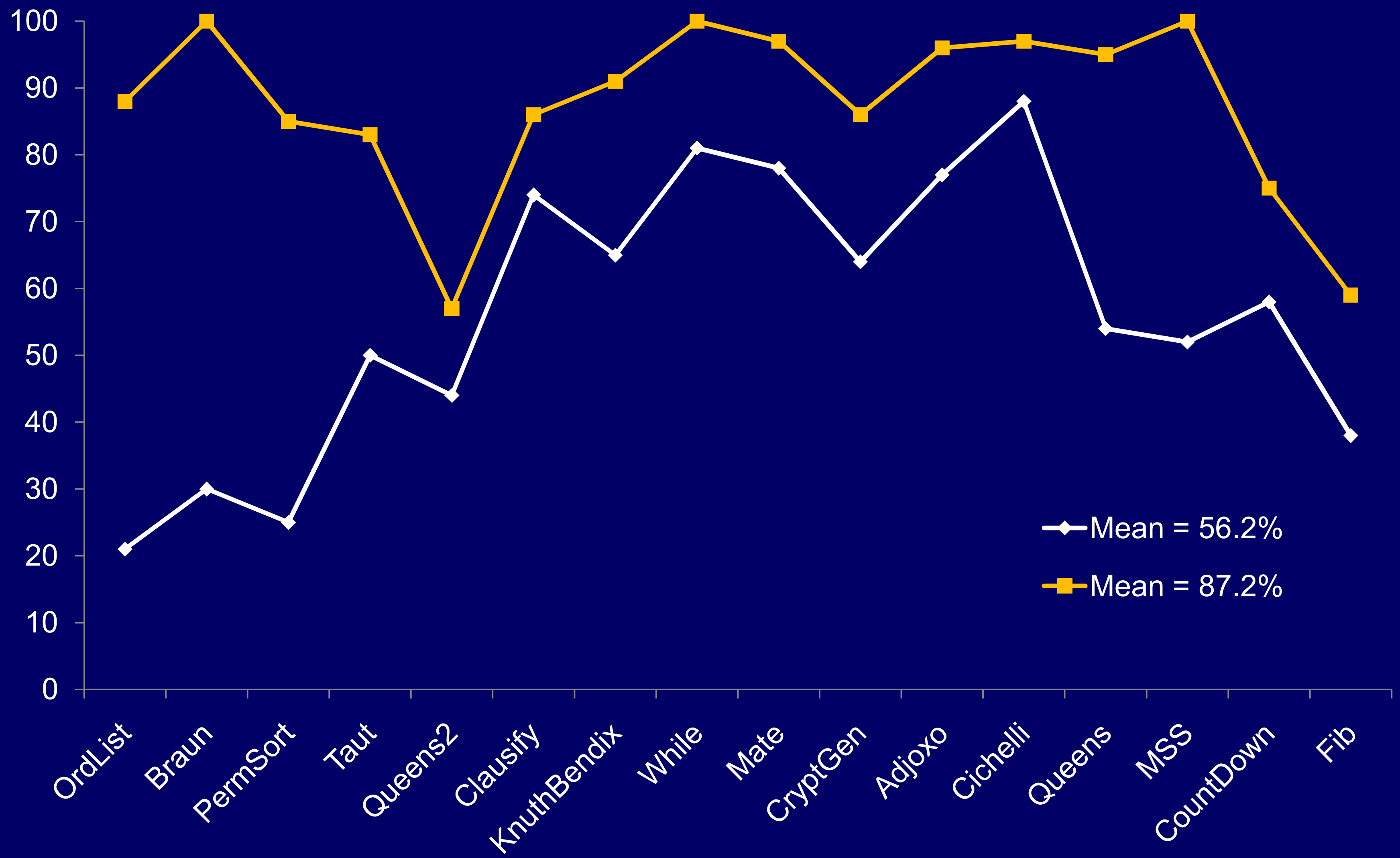
Dashing in the Reduceron

In each of the three cases, dashing is just bit-flipping under some simple-to-compute conditions.

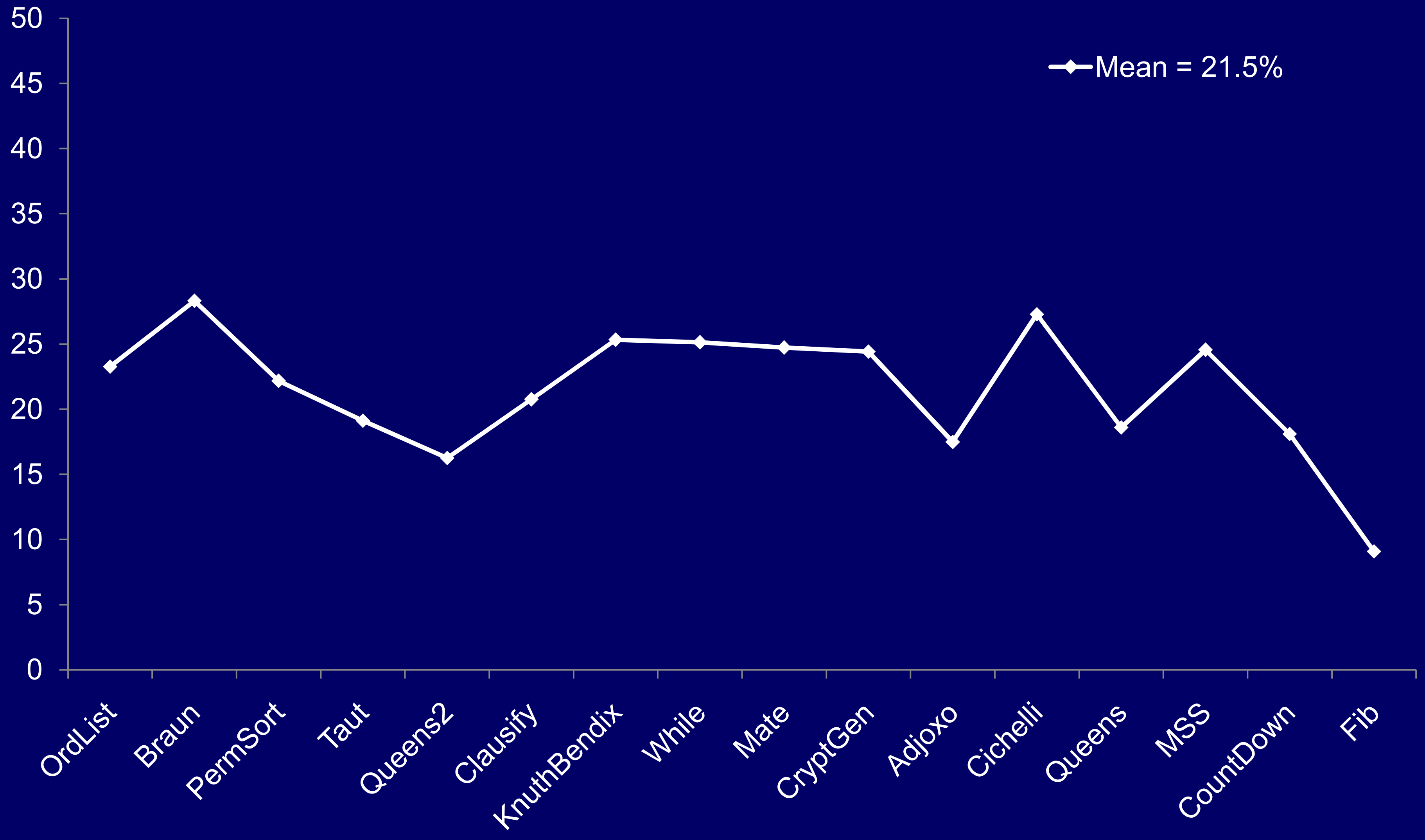
Normal-form updates avoided



Total updates avoided



% runtime saving



Dynamic analysis (2)

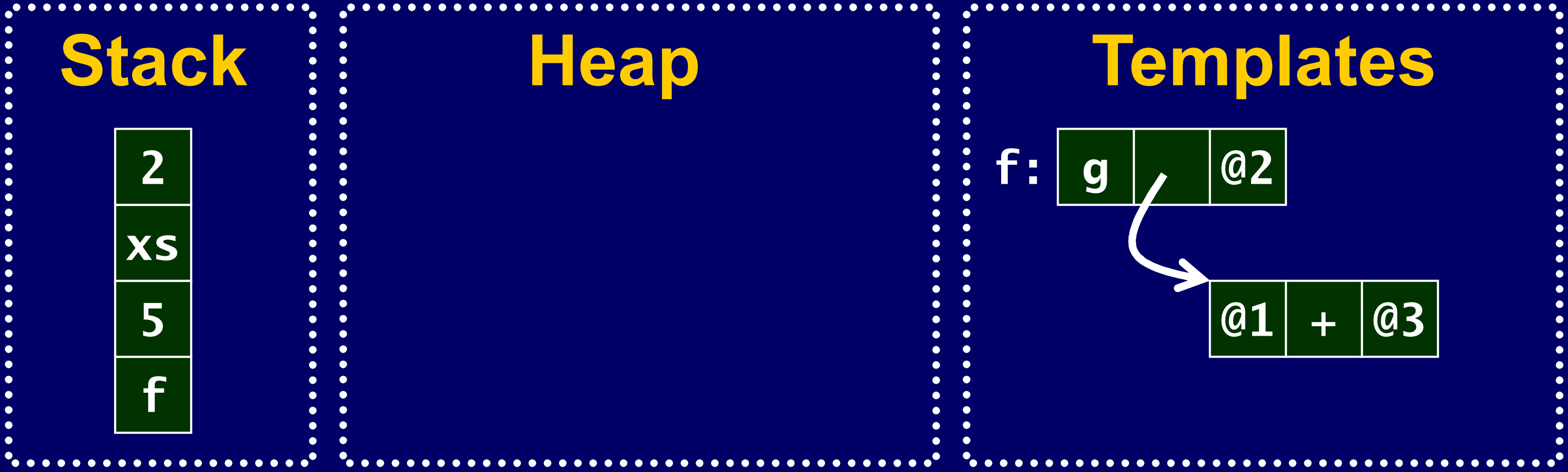
Speculative evaluation of primitive redexes (PRS)

Primitive redexes

Suppose that function **f** is defined by

```
f x xs a   =   g (x+a) xs
```

where **g** is a function, and + is primitive addition.



Stack

2

xs

5

f

Heap

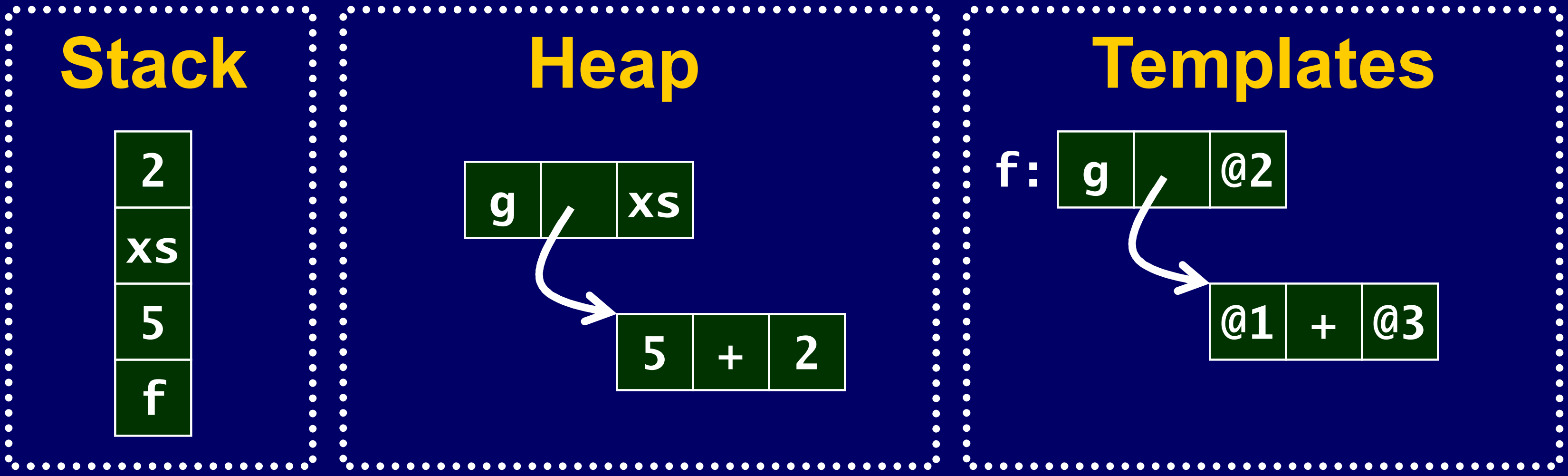
Templates

f: g @2

@1 + @3

Primitive redexes

Application of `f` results in the ***primitive redex*** `5+2` being instantiated on the heap.



Stack

2
xs
5
f

Heap

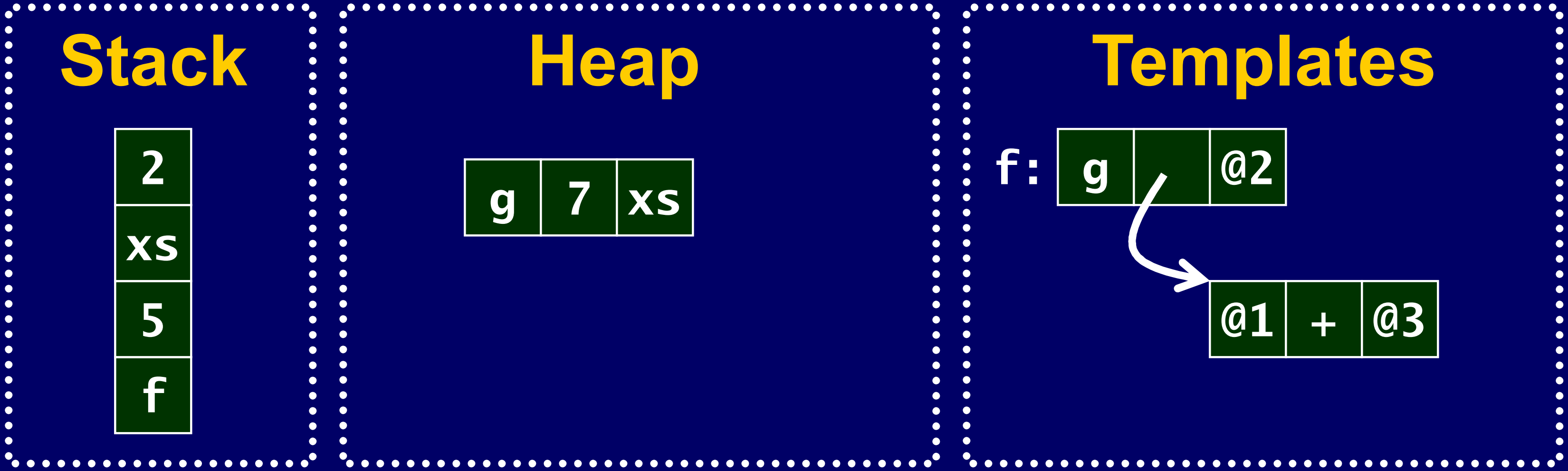
g | | xs
5 | + | 2

Templates

f: g | | @2
@1 | + | @3

Speculative evaluation

Idea: *At instantiation time, look at the arguments in a primitive application. If they are already evaluated, apply the primitive speculatively.*



Stack

2
xs
5
f

Heap

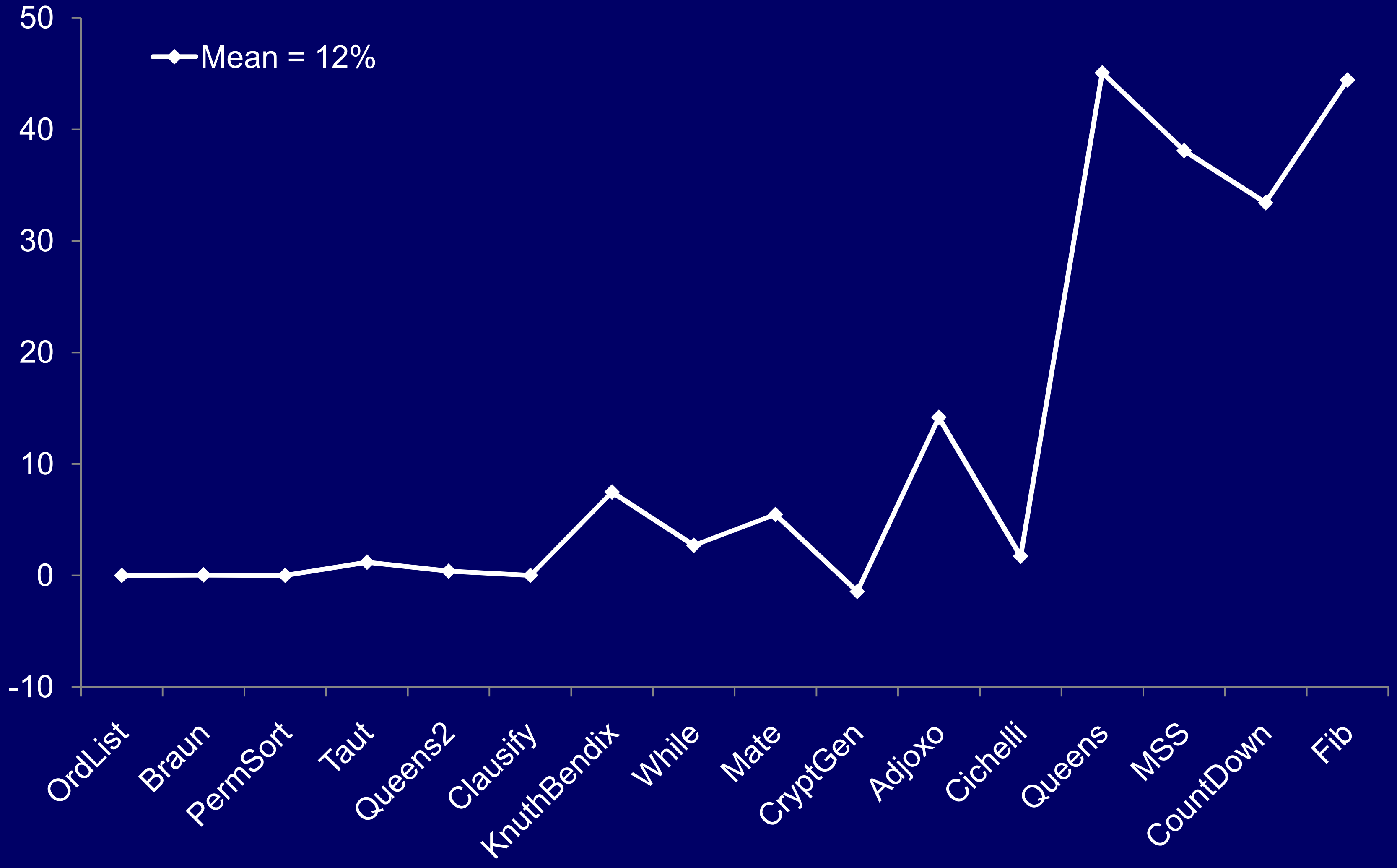
g 7 xs

Templates

f: g @2

@1 + @3

% runtime saving, due to PRS

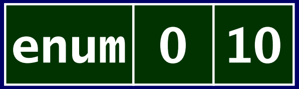


amount of primitive reduction →

Example of PRS success

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate: 

enum 0 10

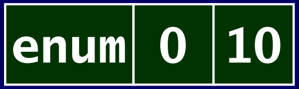
```
  enum 0 10
=
```

Example of PRS success

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate:



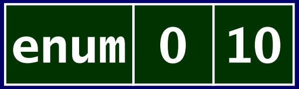
enum 0 10

```
  enum 0 10
= if   0 <= 1
  then Cons 0 (enum (0+1) 10)
  else Nil
```

Example of PRS success

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate: 

enum 0 10

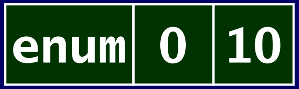
```
  enum 0 10
= if   True
  then Cons 0 (enum 1 10)
  else Nil
```

Example of PRS success

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate:



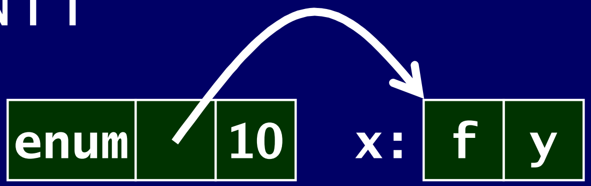
```
  enum 0 10
= if   True
  then Cons 0 (enum 1 10)
  else Nil
= Cons 0 (enum 1 10)
```

Example of PRS failure

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate:



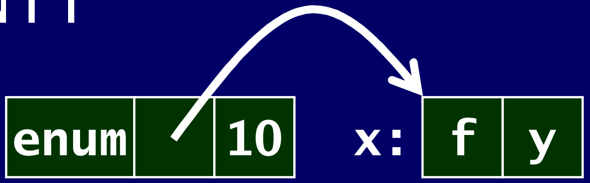
```
  enum x 10
=
```

Example of PRS failure

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate:

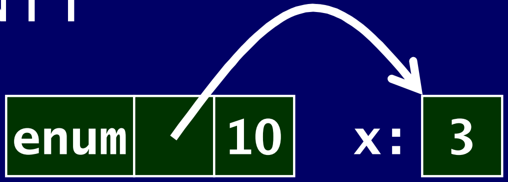


```
  enum x 10
= if   x <= 10
  then Cons x (enum (x+1) 10)
  else Nil
```

Example of PRS failure

Function to enumerate a range of integers:

```
enum n m = if   n <= m
           then Cons n (enum (n+1) m)
           else Nil
```

We wish to evaluate: 

```
  enum x 10
= if   x <= 10
  then Cons x (enum (x+1) 10)
  else Nil
= Cons x (enum (x+1) 10)
```

Possible solutions

Option 1: *Worker/wrapper transformation. Since enum is strict, introduce enumWrap which forces n and m before passing them to enum.*

Option 2: *Update the stack. Leave evaluated arguments of conditional primitives on the stack, to be observed by case alternatives.*

Conclusions

*The Reduceron enjoys **many freedoms**.*

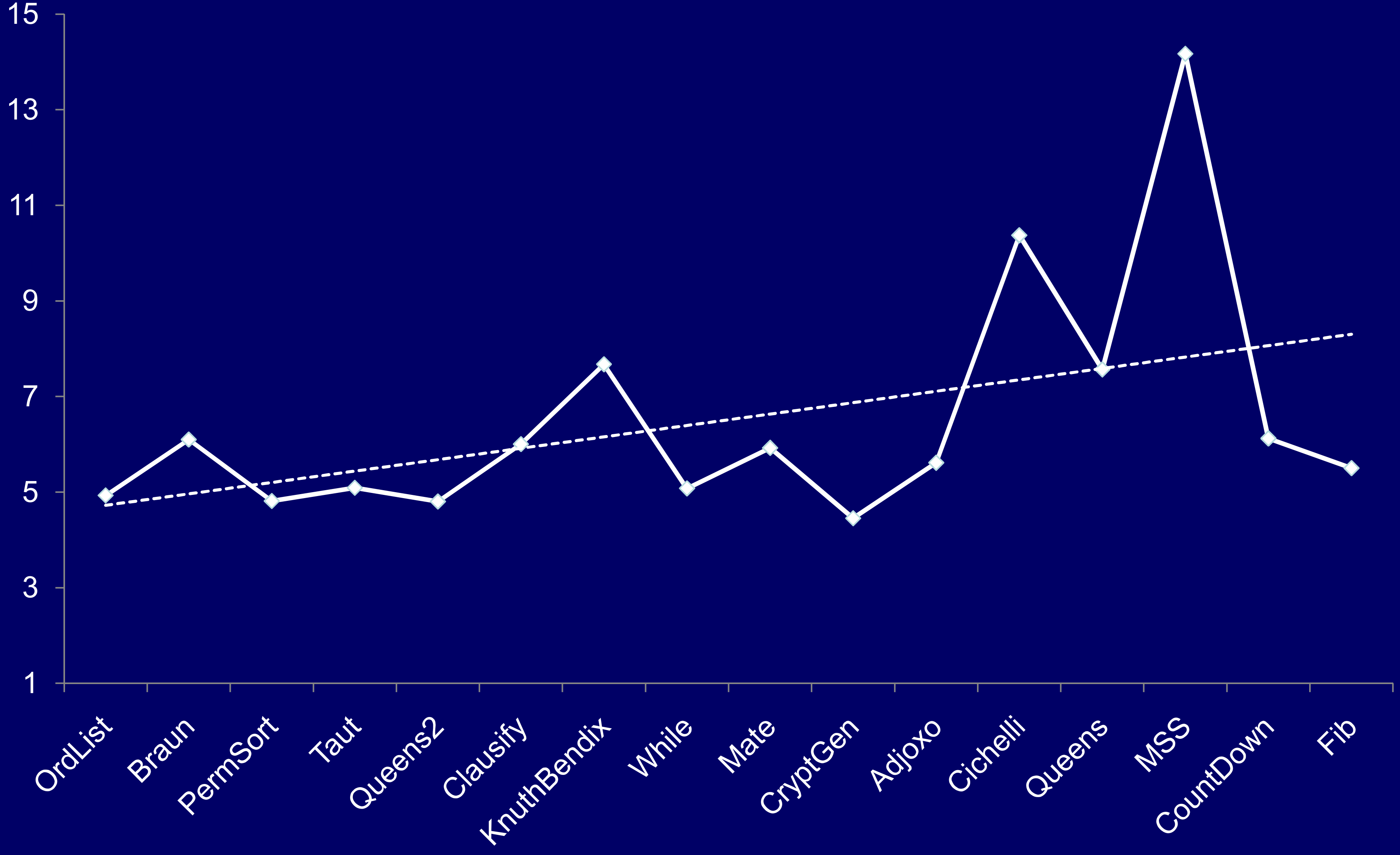
Dynamic analysis can be very attractive.

*There is **a lot** of **low-level parallelism** to exploit **even** in “sequential” graph reduction.*

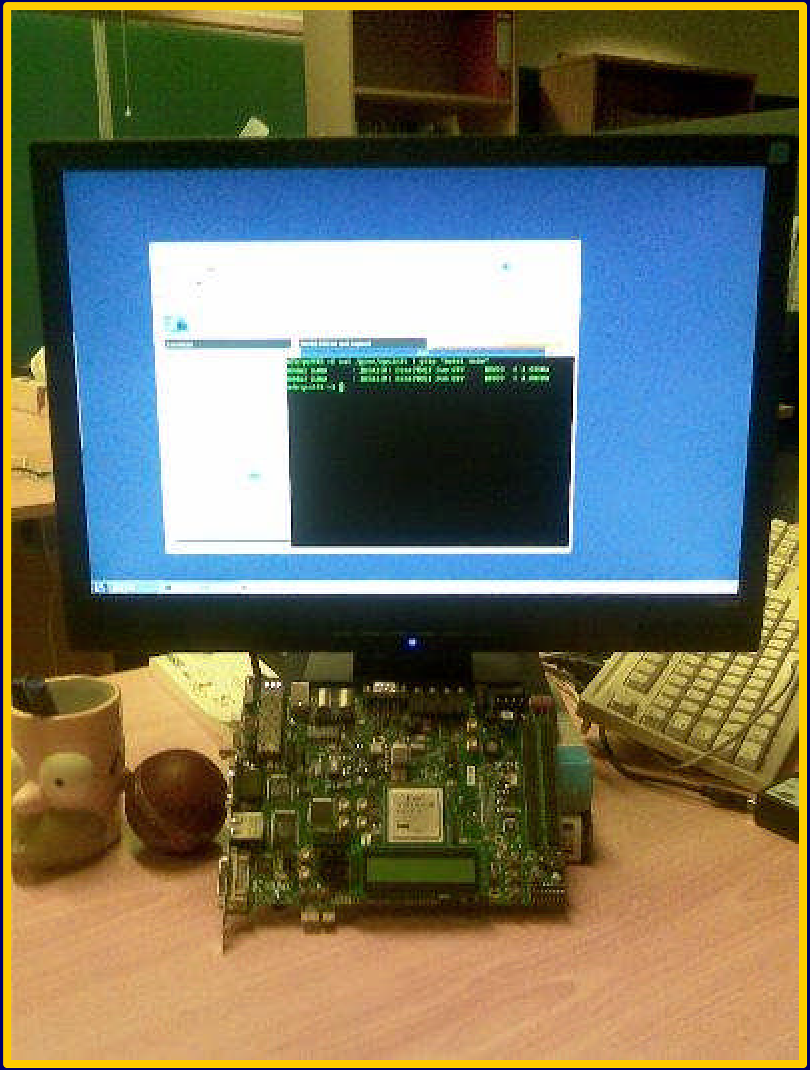
***Still to come:** Parallel reduction, PRS is just the beginning, and compiler optimisation.*

Quick progress report

Speed-up factor, since IFL'07



My desk



Intel Core 2 Duo E8400

** 6M Cache*
** 3.00 GHz*
** 1333 MHz FSB*
** Released in 2008*

Xilinx Virtex 5 FPGA

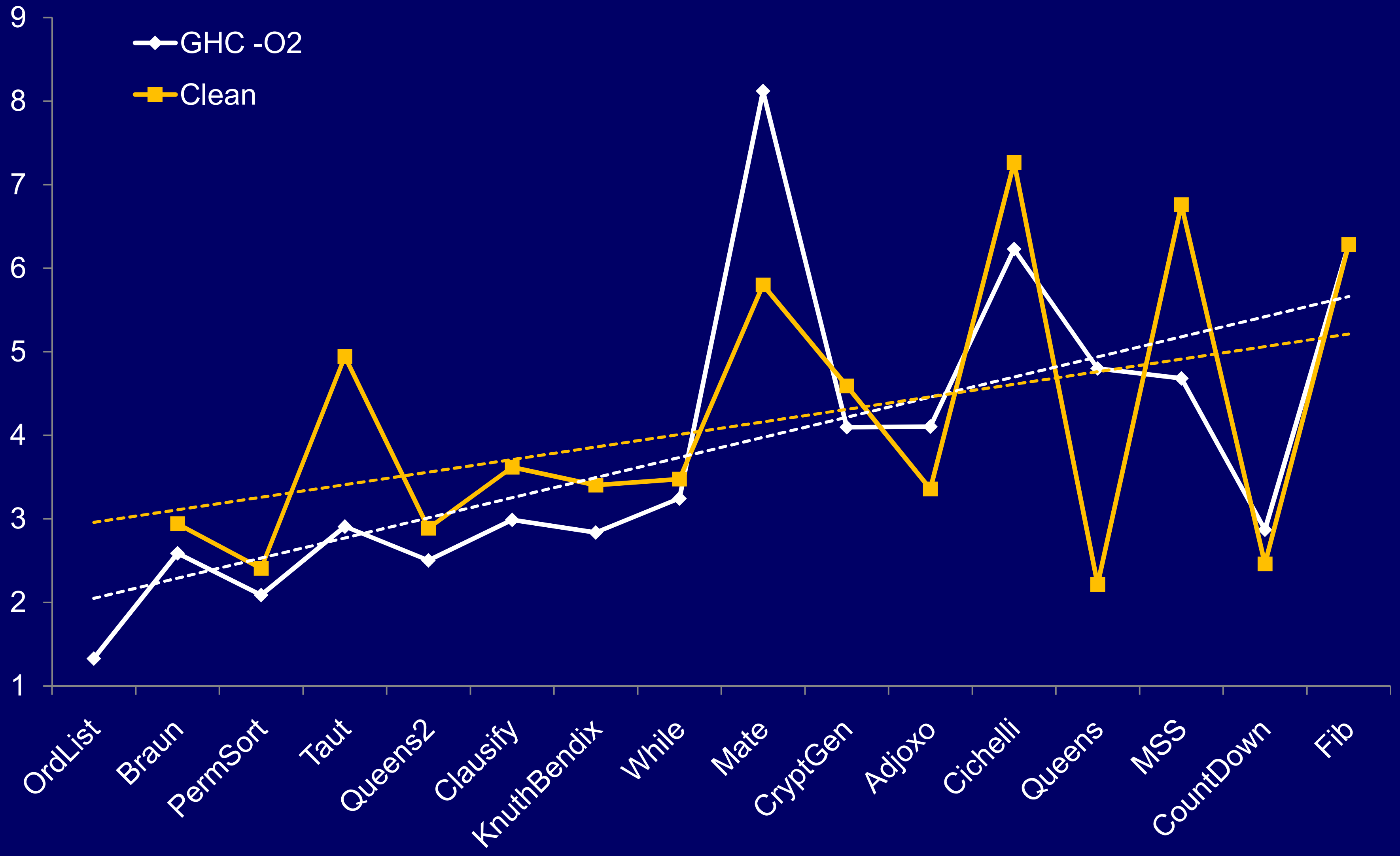
** LX110T (mid-range)*
** Speed-grade 1 (lowest)*
** Released in 2006*

Reduceron synthesis results

F_{max}	*110Mhz*	
Slice usage	*2284*	*(13%)*
LUTs	*5865*	*(8%)*
Flip-flops	*1018*	*(1%)*
BRAM usage	4.7 megabits	*(89%)*

NOTE: *memory capacity is very small – could be enhanced by moving the heap into off-chip memory.*

Slow-down factor, against PC



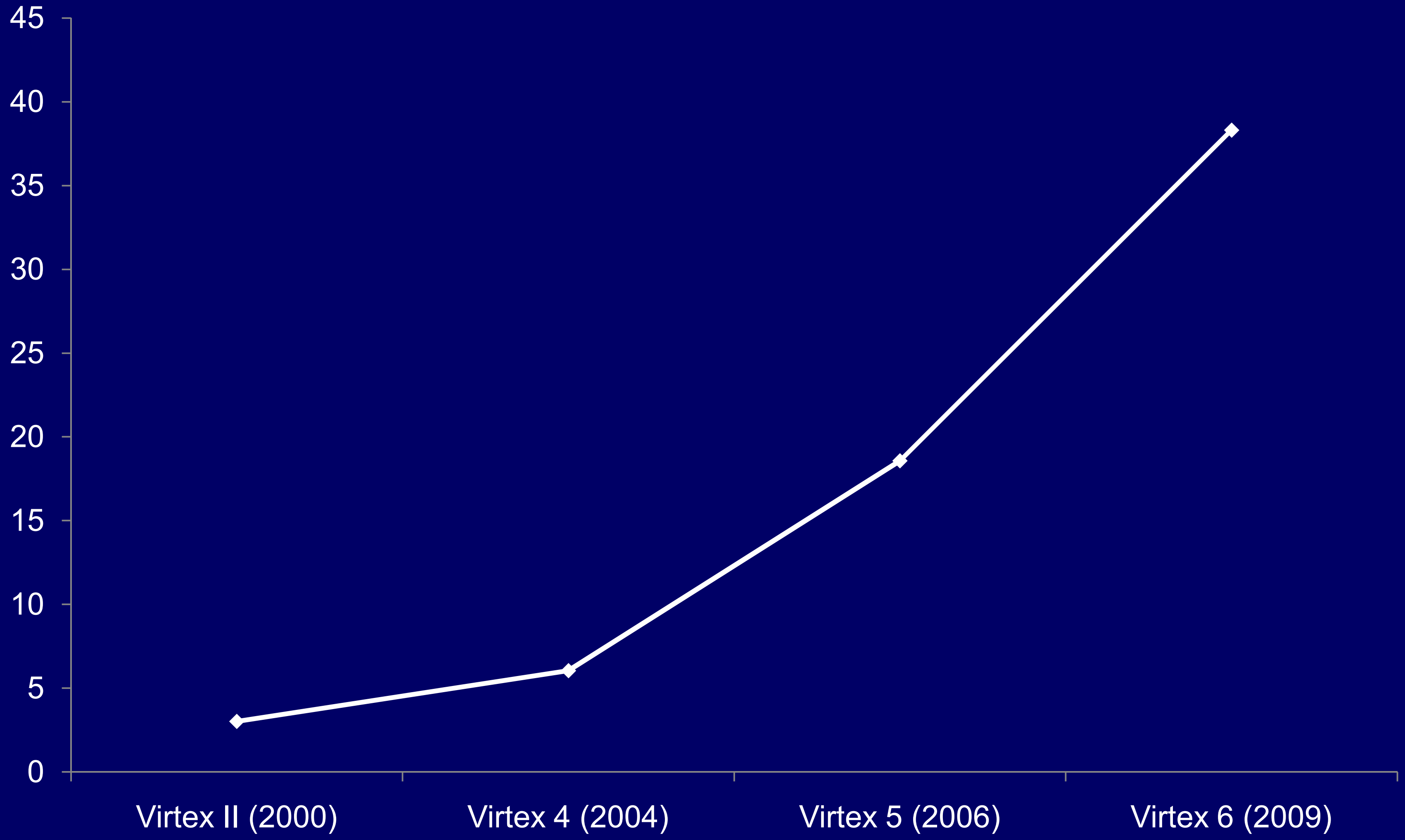
Acknowledgements

This research is supported by EPSRC grant EP/G011052/1.

Thanks to the Xilinx University Program for donating the FPGA and development board.

Thanks to Satnam Singh for his advice and contacts which enabled us to obtain this board.

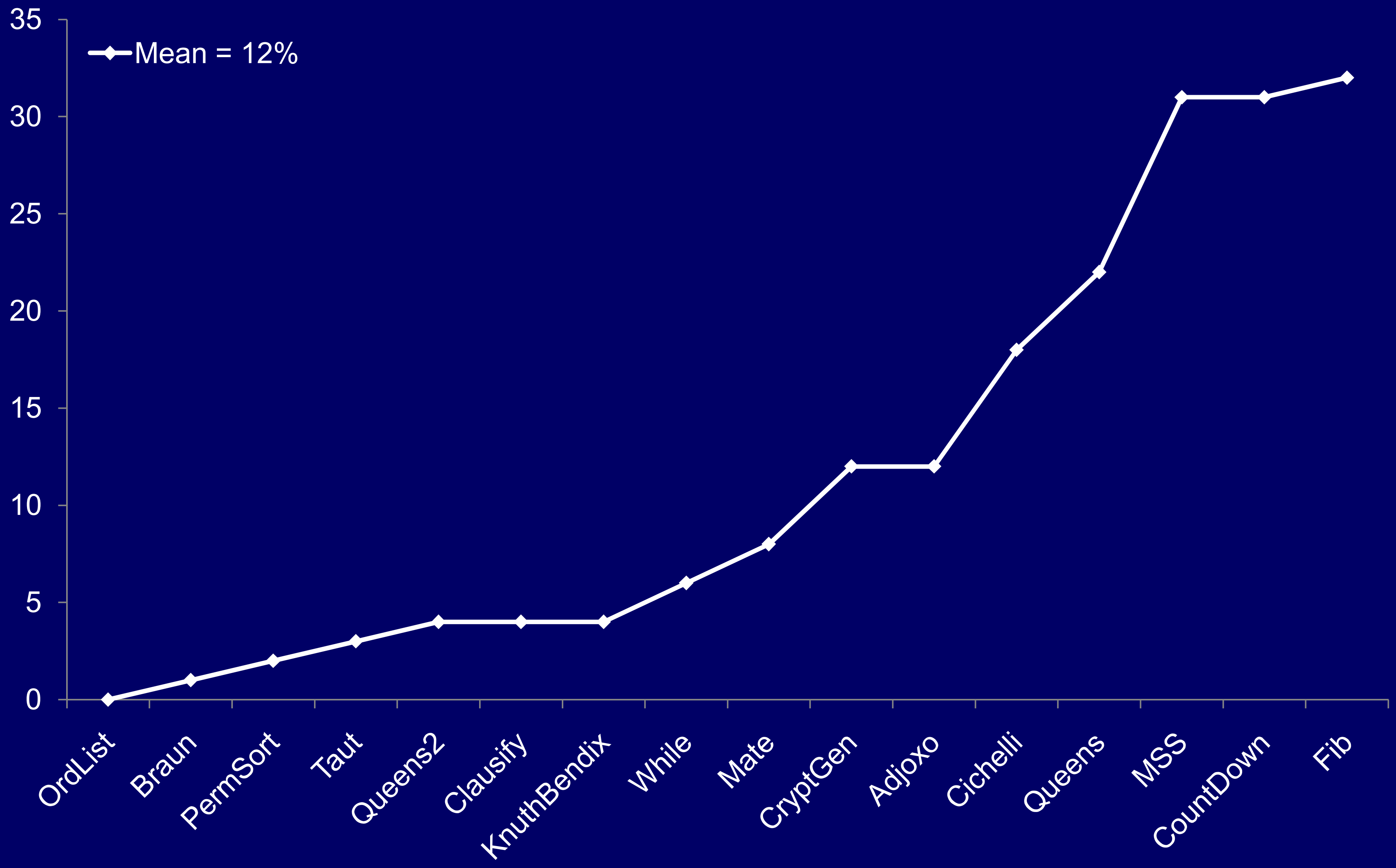
Megabits of block RAM, over time



An off-chip heap

- Heap could be moved into off-chip RAM.
  - To cater for memory-hungry programs.
- Should be possible without modifying existing design significantly.
  - Modern memory chips clock much higher than 110Mhz.
  - Example: reduced-latency RAM (4-cycle access) clocking at over 400Mhz.

% runtime spent on arithmetic


Mean = 12%
OrdList
Braun
PermSort
Taut
Queens2
Clausify
KnuthBendix
While
Mate
CryptGen
Adjoxo
Cichelli
Queens
MSS
CountDown
Fib