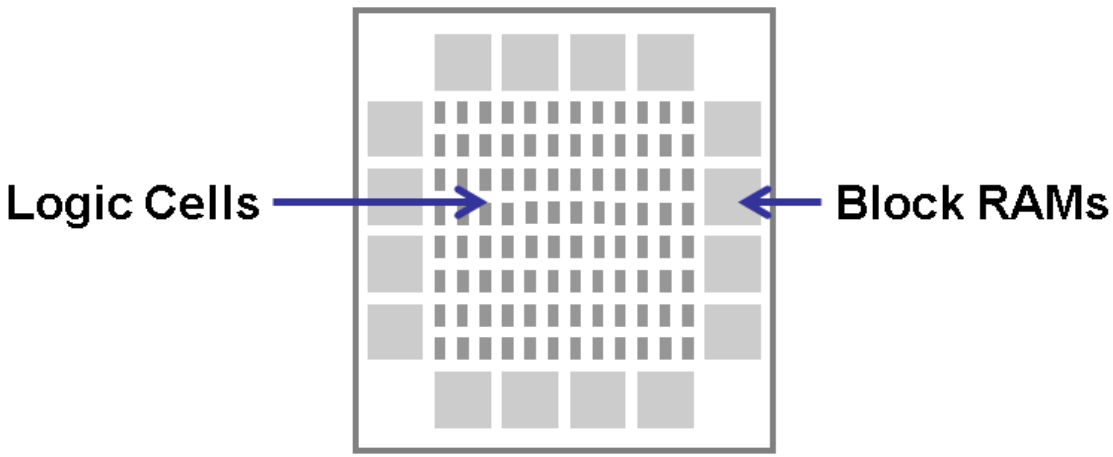
The Reduceron Reconfigured

Matthew Naylor & Colin Runciman
The University of York

The Reduceron?

- The Reduceron is a graph-reduction machine, described by a functional program, and implemented using reconfigurable hardware (FPGA).



- The Reduceron works by template instantiation, reducing function applications by substituting arguments in bodies.

Reconfigured?

- We reported an earlier Reduceron at IFL 2007.
- Reduceron 2010 has a lower clock frequency (96MHz v. 111MHz) yet on average runs programs over 5× faster.
- Each of its 6 reduction steps takes only a single clock cycle, using parallel memory transactions.
- Reductions steps are large: on average it takes less than two reduction steps to perform a complete application of a function or case alternative.
- It is a spineless graph reducer, reducing heap pressure and increasing capacity to exploit parallel memories.
- Two dynamic analyses avoid the normal costs of graph updates and primitive applications in many cases.

Compiling a core language (§2.4)

- We compile a non-strict, higher-order core language including case expressions over algebraic data types. For example:

```
append xs ys = case xs of
  {Nil -> ys ; Cons x xs -> Cons x (append xs ys)}
```

- A variant of Scott/Jansen encoding combines data construction and case selection. A constructor of datatype *d* translates to a function that in addition to components takes as arguments alternative continuations for each possible *d*-construction.
- To cut instantiation costs, alternatives are represented by a single case-table. For example:

```
append xs ys = xs <consCase, nilCase> ys
consCase x xs t ys =  Cons x (append xs ys)
nilCase       t ys =  ys
```

Reduceron template code (§2.7)

- Instead of instruction-level code, a program in the Reduceron is represented by a series of templates.

```
type Prog = [Template]
```

- A template has an arity, a spinal application and a list of non-spinal applications.

```
type Template = (Arity, App, [App])
type Arity = Int
```

- All applications are flat and represented as a series of atoms — tagged fixed-size units of information.

```
type App = [Atom]
data Atom = FUN Arity Int | ARG Int | PTR Int
          | CON Arity Int | INT Int | PRI String
          | TAB Int
```

An example of compiled template code

The top-level append case-selection function, and its alternative auxiliaries `consCase` and `nilCase`, each compile into a single template.

```
  -- append xs ys = xs <consCase, nilCase> ys
[ (2, [ARG 0, TAB 1, ARG 1], [])

  -- consCase x xs t ys = Cons x (append xs ys)
, (4, [CTR 2 0, ARG 0, PTR 0], [[FUN 2 0, ARG 1, ARG 3]])

  -- nilCase t ys = ys
, (2, [ARG 1], [])
]
```

Reduceron machine state (basic version, §3)

- A Reduceron state is a 4-tuple: program, heap, reduction stack, update stack. (Implementation: separate dual-port memories; accessible in parallel within a single cycle.)

```
type State = (Prog, Heap, Stack, UStack)
```

- The heap can be represented as a list of applications

```
type Heap = [App]
type HeapAddr = Int
```

- the reduction stack as a list of atoms

```
type Stack = [Atom]
type StackAddr = Int
```

- and the update stack as a partial association list pairing stack addresses with heap addresses.

```
type UStack = [(StackAddr, HeapAddr)]
```

Reduction rules: unwinding cycle (§3.3)

```
step (p, h, PTR x:s, u) = (p, h, h!!x ++ s, upd:u)
  where upd = (1+length s, x)
```

- Precondition: the top of the reduction stack points to an application on the heap.
- The application is copied from the heap to the reduction stack. (Wide memories allow an entire application to be copied at once.)
- For eventual updating of the heap with the result we push onto the update stack the heap address of the application and the current size of the reduction stack. (Data is written to the two stack memories in parallel.)

Reduction rules: updating cycle (§3.3)

```
step (p, h, top:s, (sa,ha):u)
  | arity top > n = (p, h', top:s, u)
  where n  = 1+length s - sa
        h' = update ha (top:take n s) h
```

- Precondition: the upper part of the stack represents a normal form (known by comparing its arity with stack-height above the stack address of the topmost update).
- This normal form is copied to the heap at the address popped from the update stack. (Wide memories allow an entire normal form to be copied at once.)

Reduction rules: primitive cycle (§3.3)

```
step (p, h, INT n:x:s, u) = (p, h, x:INT n:s, u)
step (p, h, PRI f:x:y:s, u) = (p, h, prim f x y:s, u)
```

- Assume saturated primitive binary applications which the compiler transforms by the rule $p\ e_0\ e_1 \longrightarrow e_1\ (e_0\ p)$.
- If the top of the reduction stack is an integer literal the top two elements are flipped.
- If the top of the reduction stack is a primitive application it is reduced.
- These rules present no opportunities for low-level parallelism, motivating later improvements.

Reduction rules: constructor cycle (§3.3)

```
step (p, h, CON n j:s, u) = (p, h, FUN 0 (i+j):s,u)
  where TAB i = s!!n
```

- Scott-encoded constructors are reduced by indexing a case table of addresses for case-alternative functions.
- The arity of the case-alternative function is unknown. But as every application of a case-alternative function is saturated, a dummy arity of 0 is safe.
- Again no parallelism: a simple stack-only transition takes an entire cycle, suggesting scope for improvement.

Reduction rules: application cycle (§3.3)

```
step (p, h, FUN n f:s, u) = (p, h', s', u)
  where (pop, spine, apps) = p !! f
        h' = h ++ map (instApp s h) apps
        s' = instApp s h spine ++ drop pop s

instApp s h = map (inst s (length h))
```

- Precondition: the topmost part of the reduction stack is an application of function f of arity n.
- $n + 1$ elements are popped off the reduction stack.
- the spine application of the body of f is instantiated and pushed onto the reduction stack (in parallel, separate dual-port wide memories).
- Other applications in the body are instantiated and appended to the heap (in parallel, all in the same cycle).

Bounding parameters for single-cycle reduction (§4.2)

Single-cycle application is made possible by fixing as design parameters:

- maximum widths, *sw* and *nsw*, of spinal and non-spinal applications, and
- a maximum number, *n*, of applications in a template body.

nsw	**reds.**	**heap**
2	1.00	1.00
3	0.84	1.00
4	0.83	1.30
5	0.82	1.57
6	0.82	1.89

sw	**reds.**	**heap**
2	1.00	1.00
3	0.82	0.76
4	0.76	0.67
5	0.71	0.60
6	0.70	0.57

n	**reds.**
1	1.00
2	0.89
3	0.85
4	0.85

If a function application or a body exceeds these parameters, it is bracketed into a nested application or split across more than one template.

Stack memory & single-cycle reduction (§4.3,§4.4)

At a lower implementation level, single-cycle reduction depends on

- simultaneous read-write access to the top n stack elements, for $n = 8$, by a crossbar switch of > 2000 logic gates;
- delay-free access to memory referenced from the stack top.

  Invariant 1: *if the top of the reduction stack is of the form* `PTR` *x then the application at heap address x is currently available on the heap memory's data bus.*

  Invariant 2: *if the top of the reduction stack is of the form* `FUN` *n f then the template at program address f is currently available on the program memory's data bus.*

Improvement 1: the case-table stack (§4.5)

Observation: constructor reduction modifies only the top of the reduction stack, adding the constructor index to a case-table address.

Idea: save a clock-cycle; make it combinatorial.

Snag: with case-table addresses at variable stack positions, we need a multiplexer, slowing the combinatorial logic.

Solution: a separate case-table stack — more low-level memory parallelism — so a needed case-table address is always at the top.

Improvement 2: update avoidance (§5.1)

Observation: an update is unnecessary if

- the application is already evaluated, or
- the application is not shared.

Idea: track un-shared applications by dynamic analysis.

Implementation: argument and pointer atoms gain an extra bit indicating possible shared reference.

```
data Atom = ... | ARG Bool Int | PTR Bool Int | ...
```

An argument tagged `True` is referenced more than once in a function body. A pointer tagged `False` is a unique pointer.

Invariant 3: *a unique pointer occurring on the reduction stack points to a non-shared application.*

Improvement 3: primitive-redex speculation (§5.3)

Consider this `safe` function from an N-queens program:

```
safe :: Int -> Int -> [Int] -> Bool
safe x d []       =  True
safe x d (q:qs)   =  x /= q && x /= q+d && x /= q-d &&
                     safe x (d+1) qs
```

When reducing the application `safe 1 1 [2]`, instead of mere substitution in the body

```
1 /= 2 && 1 /= 2+1 && 1 /= 2-1 && safe 1 (1+1) []
```

the Reduceron dynamically detects primitive redexes and speculatively evaluates them on-the-fly.

```
True && True && False && safe 1 2 []
```

Performance v. basic design (Table 3)

Implementation	**Clock-Cycles**	**Heap Usage**
Baseline	1.00	1.00
+ Flat In-lining	0.88	0.92
+ Case Stack	0.74	0.92
+ Update Avoidance	0.57	0.92
+ Infix Primitives	0.47	0.69
+ Primitive Speculation	0.40	0.50

Performance v. leading compilers for PC (Table 4)

Program	**Lines**	**GHC -O2 Run-time**	**Clean Run-time**	**Hand reds. per Cycle**
...				
Fib	10	0.14	0.14	0.90
KnuthBendix	551	0.37	0.21	0.47
...				
Mean (16 progs.)	135	0.29	0.23	0.55

- The GHC and Clean target is an Intel Core 2 Duo E8400 PC clocking at 3GHz.
- Times are normalised as fractions of run-time for Xilinx Virtex-5 FPGA Reduceron clocking at 96MHz (30× slower).
- Hand reductions are applications of defined functions, case alternatives or primitives — not updating, unwinding, etc.

Related Work (§6.5–§6.9)

SKIM (Stoye 1985): microcoded processor for SKI graph reduction; one-bit reference counts in pointers — to cut GC.

Spineless G-machine (Burn et. al. 1988): spineless! Static analysis for update avoidance — *"cost of [dynamic tags] greatly outweighs the advantages of precision"*.

Big Word Machine (Augustsson 1992): design for a graph-reduction machine; wide memory for rapid transfer to and from heap; Scott-style encoding; crossbar-switch stack; explicit instruction stream. Close in spirit to the Reduceron, but never actually built.

PilGRIM, (Boeijink et. al., IFL 2010): design of a pipelined processor with a high-level instruction set for graph reduction; motivated by Reduceron 2007; eventual target 1GHz; not yet built.

Conclusions and Future Work (§7)

- The results for Reduceron 2010 are promising, but not yet compelling.
- Leading compilers & current PCs are only a factor of 4 away — a narrow gap between a hard processor and a reconfigurable one.
- We have begun work on compiler optimization — eg. static prediction; supercompilation.
- Reduceron 2010 has only small heap and program memories. Can the design be adapted for larger off-chip memory?
- Despite low-level parallelism, and a small amount of speculation, Reduceron 2010 is essentially a sequential graph reducer. How about a multi-core Reduceron for parallel reduction?
- We have GPUs for graphics. We look forward to FPUs for functional applications.

Acknowledgements

Project funded by

FPGA kit donated by