The Reduceron Reconfigured

Matthew Naylor Colin Runciman
University of York, UK
{mfn,colin}@cs.york.ac.uk

Abstract

The leading implementations of graph reduction all target conventional processors designed for low-level imperative execution. In this paper, we present a processor *specially designed* to perform graph-reduction. Our processor – the Reduceron – is implemented using off-the-shelf *reconfigurable hardware*. We highlight the *low-level parallelism* present in sequential graph reduction, and show how *parallel memories* and *dynamic analyses* are used in the Reduceron to achieve an average reduction rate of 0.55 function applications per clock-cycle.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles—High-level language architectures; D.3.4 [*Programming Languages*]: Processors—Run-time environments; I.1.3 [*Symbolic and Algebraic Manipulation*]: Languages and Systems—Special-Purpose Hardware

General Terms Design, Experimentation, Performance

Keywords Graph Reduction, Reconfigurable Hardware

1. Introduction

Efficient evaluation of high-level functional programs on conventional computers is a big challenge. Sophisticated techniques are needed to exploit architectural features designed for low-level imperative execution. Furthermore, conventional computers have limitations when it comes to running functional programs. For example, memory bandwidth is limited to *serial* communication in *small units*. Evaluators based on graph reduction perform intensive construction and deconstruction of expressions in memory. Each such operation requires sequential execution of *many* machine instructions, not because of any inherent data dependencies, but because of architectural constraints in conventional computers.

All this motivates the idea of computers specially designed to meet the needs of high-level functional languages - much as GPUs are designed to meet needs in graphics. This is not a new idea. In the '80s and '90s there was a 15-year ACM conference series *Functional Programming Languages and Computer Architecture*. In separate initiatives, there was an entire workshop concerned with graph-reduction machines alone [Fasel and Keller 1987], and a major computer manufacturer built a graph-reduction prototype [Scheevel 1986]. But the process of constructing exotic new hardware was slow and uncertain. With major advances in compilation for ever bigger, faster and cheaper mass-market machines, the idea of specialised hardware for functional languages went out of fashion.

Reconfigurable Hardware Today, the situation is quite different. *Field-programmable gate arrays* (FPGAs) have greatly reduced the effort and expertise needed to develop special-purpose hardware. They contain thousands of parallel logic blocks that can be configured at will by software tools. They are widely-available and are an advancing technology that continues to offer improved performance and capacity.

The downside of FPGA applications is that they typically have much lower maximum clocking frequencies than corresponding directly-fabricated circuits – this is the price to pay for *reconfigurability*. To obtain good performance using an FPGA, it is therefore necessary to exploit significant parallelism.

The Reduceron In this paper, we present a special-purpose machine for sequential graph reduction – the Reduceron – implemented on an FPGA. We build upon our previous work on the same topic [Naylor and Runciman 2007] by presenting a new design that exhibits a factor of five performance improvement.

A notable feature of our new design is that each of its six semantic reduction rules is performed in a *a single-clock cycle*. All the necessary memory transactions required to perform a reduction are done *in parallel*. The Reduceron performs on average 0.55 *hand-reductions* per clock-cycle. A hand-reduction is a reduction that programmer would perform in *by-hand* evaluation trace of a program; it includes function application and case analysis, but not machine-level reductions such as updating and unwinding.

Another notable development in our new design is the use of two *dynamic analyses* enabling *update avoidance* and *speculative evaluation of primitive redexes*, both of which lead to significant performance improvements. On conventional computers, the runtime overhead of these dynamic analyses would be prohibitive, but on FPGA they are cheap and simple to implement.

Contributions In summary, we give:

- §**2** a precise description of the Reduceron compiler, including refinements to the Scott encoding of constructors, used for compiling case expressions, addressing various efficiency concerns;
- §**3** an operational semantics of the template instantiation machine underpinning the Reduceron implementation;
- §**4** a detailed description of how each semantic reduction rule is implemented in a single clock-cycle using an FPGA;
- §**5** extensions to the semantics to support (1) dynamic sharing analysis, used to avoid unnecessary heap updates, and (2) dynamic detection of primitive redexes, enabling speculative reduction of such expressions during function-body instantiation;
- §**6** a comparative evaluation of the Reduceron implementation against other functional language implementations.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

e	::=	$\vec{e}$	(Application)
	|	`case` e `of {` $\vec{a}$ `}`	(Case Expression)
	|	`let {` $\vec{b}$ `} in` e	(Let Expression)
	|	n	(Integer)
	|	x	(Variable)
	|	p	(Primitive)
	|	f	(Function)
	|	C	(Constructor)
	|	`<`$\vec{f}$`>`	(Case Table)
a	::=	$C\ \vec{x}$ `->` e	(Case Alternative)
b	::=	x `=` e	(Let Binding)
d	::=	$f\ \vec{x}$ `=` e	(Function Definition)

Figure 1. Core syntax of F-lite.

2. Compilation

This section defines a series of refinements that take programs written in a lazy functional language called *F-lite* to a form known as *template code* which the Reduceron can execute.

2.1 Source Language

F-lite is a core lazy functional language, close to subsets of both Haskell and Clean. The syntax of F-lite is presented in Figure 1.

Case Expressions Case expressions are in a simplified form that can be produced by a pattern match compiler such as that defined in [Peyton Jones 1987]. Patterns in case alternatives are constructors applied to zero or more variables. All case expressions contain an alternative for every constructor of the case subject's type.

Primitives The meta-variable p denotes a primitive function symbol. All applications of primitive functions are fully saturated. The Reduceron implements only a small set of primitive operations, not the full set of a conventional processor – e.g. we have no floating-point operations. Primitives used in this paper include: `(+)`, `(-)` and `(<=)`.

Main Every program contains a definition `main` $=$ e where e is an expression that evaluates to an integer n; the result of the program is the value n.

Case Tables Notice the unusual case-table construct `<`$\vec{f}$`>`. Case tables are introduced during compilation – see §2.4.

Examples Here are two example function definitions. The first concatenates two lists and the second computes triangular numbers.

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

tri n = case (<=) n 1 of
  { False -> (+) (tri ((-) n 1)) n ; True -> 1 }
```

2.2 Terminology

Application Length The *length* of an application $e_1 \cdots e_n$ is n. For example, the length of the application `append xs ys` is three.

Compound and Atomic Expressions Applications, case expressions and let expressions are *compound* expressions. All other expressions are *atomic*.

Flat Expression A *flat* expression is an atomic expression or an application $e_1 \cdots e_n$ in which each e_i for i in $1 \cdots n$ is an atomic expression. For example, `append xs ys` is a flat expression, but `tri ((-) n 1)` is not.

Expression Graph A let expression

$$\texttt{let \{}\ x_1 \texttt{=}\ e_1\ \texttt{;}\ \cdots\ \texttt{;}\ x_n \texttt{=}\ e_n\ \texttt{\}\ in}\ e$$

is an *expression graph* exactly if e is a flat expression and each e_i for i in $1 \cdots n$ is a flat expression. Expression graphs are restricted A-normal forms [Flanagan et al. 1993].

Constructor Index and Arity Each constructor C of a data type with m constructors is associated with a unique *index* in the range $1 \cdots m$. More precisely, the index of a constructor is its position in the *alphabetically sorted* list of all constructors of that data type. For example, the standard list data type has two constructors: `Cons` has index 1 and `Nil` has index 2. A constructor with index i is denoted C_i, and the arity of a constructor C is denoted $\#C$.

2.3 Primitive Applications

In a lazy language, an application of a primitive function such as `(+)`, `(-)` or `(<=)` requires special treatment: the integer arguments must be fully evaluated before the application can be reduced. One simple approach is to transform binary primitive applications by the rule

$$p\ e_0\ e_1 \ \rightarrow\ e_1\ (e_0\ p) \tag{1}$$

with the run-time reduction rule

$$n\ e \ \rightarrow\ e\ n \tag{2}$$

for any fully evaluated integer literal n. To illustrate this approach, consider the expression `(+) (tri 1) (tri 2)`. By compile-time application of rule (1), the expression is transformed to `tri 2 ((tri 1) (+))`. At run-time, reduction is as follows.

```
    tri 2 ((tri 1) (+))    { tri 2 evaluates to 3 }
=   3 ((tri 1) (+))        { Rule (2) }
=   (tri 1) (+) 3          { tri 1 evaluates to 1 }
=   1 (+) 3                { Rule (2) }
=   (+) 1 3
```

After transformation by rule (1), `tri` looks as follows.

```
tri n = case 1 (n (<=)) of
  { False -> n (tri (1 (n (-))) (+)) ; True -> 1 }
```

In §5, we present more efficient techniques for dealing with primitive applications.

2.4 Case Expressions

This section describes how case expressions are compiled. First we recall the Scott encoding recently rediscovered by Jansen [Jansen et al. 2007]. Then we make a number of refinements to this encoding.

The Scott/Jansen Encoding The first step of the encoding is to generate, for each constructor C_i of a data type with m constructors, a function definition

$$C_i\ x_1 \cdots x_{\#C_i}\ k_1 \cdots k_m\ \texttt{=}\ k_i\ x_1 \cdots x_{\#C_i}$$

The idea is that each data constructor C_i is encoded as a function that takes as arguments the $\#C_i$ arguments of the constructor and m continuations.The function encoding constructor C_i passes the constructor arguments to the i^{th} continuation. For example, the list constructors are transformed to the following functions.

```
Cons x xs c n = c x xs
Nil       c n = n
```

Now case expressions of the form

$$\texttt{case}\ e\ \texttt{of \{}\ C_1\ \vec{x}_1\ \texttt{->}\ e_1\ \texttt{;} \cdots \texttt{;}\ C_m\ \vec{x}_m\ \texttt{->}\ e_m\ \texttt{\}}$$

are transformed to

$$e\ (alt_1\ \vec{v}_1\ \vec{x}_1) \cdots (alt_m\ \vec{v}_m\ \vec{x}_m)$$

where $\vec{v}_i$ are the free variables occurring in the i^{th} case alternative and each alt_i for i in $1 \cdots m$ has the definition

$$alt_i\ \vec{v}_i\ \vec{x}_i\ \texttt{=}\ e_i$$

For example, the append function is transformed to

```
append xs ys = xs (consCase ys) (nilCase ys)
consCase ys x xs = Cons x (append xs ys)
nilCase  ys      = ys
```

Notice that the application of `nilCase` could be reduced at compile time. This is a consequence of constructor `Nil` having arity 0.

Larger Example Now let us look at a slightly larger example: an evaluator for basic arithmetic expressions.

```
eval x y e = case e of {
  Add n m -> (+) (eval x y n) (eval x y m);
  Neg n   -> (-) 0 (eval x y n);
  Sub n m -> (-) (eval x y n) (eval x y m);
  X       -> x;
  Y       -> y;
}
```

After transformation, and in-lining the nullary cases, we have:

```
eval x y e = e (add x y) (neg x y) (sub x y) x y
add  x y n m = (+) (eval x y n) (eval x y m)
neg  x y n   = (-) 0 (eval x y n)
sub  x y n m = (-) (eval x y n) (eval x y m)
```

Look at the large body of `eval`: it contains three nested function applications and several repeated references to `x` and `y`. In typical functional language implementations, large function bodies are more expensive to construct than small ones.

Refinement 1 Rather than partially apply each case-alternative function to the free variables it refers to, we can define every alternative function alike to take *all* free variables occurring in *any* alternative. A case alternative can simply ignore variables that it does not need. So, let us instead transform case expressions to

$$e\ alt_1 \cdots\ alt_m\ \vec{v}$$

where $\vec{v}$ is the union of the free variables in each case alternative, and each alt_i for i in $1 \cdots m$ has the definition

$$alt_i\ \vec{x}_i\ \vec{v}\ \texttt{=}\ e_i$$

Each case-alternative function now takes the constructor arguments followed by the free variables, rather than the other way around. To illustrate, `append` now looks as follows.

```
append xs ys = xs consCase nilCase ys
consCase x xs ys = Cons x (append xs ys)
nilCase       ys = ys
```

And `eval` becomes

```
eval x y e = e add neg sub xCase yCase x y
add   n m x y = (+) (eval x y n) (eval x y m)
neg   n   x y = (-) 0 (eval x y n)
sub   n m x y = (-) (eval x y n) (eval x y m)
xCase     x y = x
yCase     x y = y
```

The new bodies of `append` and `eval` contain *no* nested function applications and *no* repeated references. An apparent disadvantage is that we have had to introduce functions for the 0-arity constructor cases `nilCase`, `xCase`, and `yCase`. But our next refinement prepares the way to recover the cost of applying these functions.

Refinement 2 We now have a large row of contiguous constants in the body of `eval`. To allow these constants to be represented efficiently (see §2.7) we place them in a *case table*. Case expressions are transformed to

$$e\ \texttt{<}alt_1\ \texttt{,}\ \cdots\ \texttt{,}\ alt_m\texttt{>}\ \vec{v}$$

and each constructor C_i is encoded as

$$C_i\ x_1 \cdots x_{\#C_i}\ t\ \texttt{=}\ (t\ \texttt{!}\ i)\ \ x_1 \cdots x_{\#C_i}$$

where $t!i$ returns the i^{th} element of case table t.

Refinement 3 An evaluator can handle constructors more efficiently than general function definitions. We could introduce the following reduction rule for constructors.

$$C_i\ e_1 \cdots e_{\#C_i}\ t\ \ \rightarrow\ \ (t\ \texttt{!}\ i)\ \ e_1 \cdots e_{\#C_i}$$

This rule replaces a constructor with a case-alternative function by looking up the case table using the constructor's index. However, the rule *also* drops the t argument. As a result, an implementation would have to slide the constructor arguments down the stack. A reduction rule that does not require argument sliding is

$$C_i\ e_1 \cdots e_{\#C_i}\ t\ \ \rightarrow\ \ (t\ \texttt{!}\ i)\ \ e_1 \cdots e_{\#C_i}\ t \qquad (3)$$

To account for the fact that t has *not* been dropped, the case-alternative functions take the form:

$$alt_i\ \vec{x}_i\ t\ \vec{v}\ \texttt{=}\ e_i$$

The final version of `append` is

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

The `t` argument is simply ignored by the case alternatives. The final version of `tri` is

```
tri n = 1 (n (<=)) <falseCase, trueCase> n
falseCase t n = n (tri (1 (n (-))) (+))
trueCase  t n = 1
```

In §3.3 and §4.5 we will see how these refinements enable efficient choices to be made at the implementation level.

2.5 In-lining

Our definition of `append` is no longer directly recursive. This is a consequence of splitting the case alternatives off as new function definitions. However, direct recursion is easily recovered: simply in-line the definition of `append` in the body of `consCase`.

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

This transformation motivates the following general in-lining rule: *in-line saturated applications of functions that have flat bodies*. In-lining a flat expression e is often a big win because it eliminates a reduction and e is often no larger than the application it replaces.

2.6 Expression Graphs

It is convenient for implementation purposes to make the graph structure of function bodies explicit by transforming them to expression graphs (§2.2). This is achieved by three rewrite rules. (1) Lift nested applications into let bindings

$$e_1 \cdots (e_i) \cdots e_n\ \ \rightarrow\ \ \texttt{let \{}\ x\ \texttt{=}\ e_i\ \texttt{\} in}\ e_1 \cdots x \cdots e_n$$

where e_i is an application or a let expression, and x is a fresh variable. (2) Lift let expressions out of let bodies.

$$\texttt{let \{}\ \vec{b}_0\ \texttt{\} in (let \{}\ \vec{b}_1\ \texttt{\} in}\ e) \rightarrow \texttt{let \{}\ \vec{b}_0\ \texttt{;}\ \vec{b}_1\ \texttt{\} in}\ e$$

```
> data Atom =
>    FUN Arity Int  -- Function with arity and address
>  | ARG Int        -- Reference to a function argument
>  | PTR Int        -- Pointer to an application
>  | CON Arity Int  -- Constructor with arity and index
>  | INT Int        -- Integer literal
>  | PRI String     -- Primitive function name
>  | TAB Int        -- Case table
```

Figure 2. Syntax of atoms in template code.

(3) Lift let expressions out of let bindings.

$$\texttt{let \{ } \cdots \texttt{; x = let \{ } \vec{b} \texttt{ \} in } e_0 \texttt{ ; } \cdots \texttt{ \} in } e_1 \rightarrow \texttt{let \{ } \cdots \texttt{; } \vec{b} \texttt{ ; x = } e_0 \texttt{ ; } \cdots \texttt{ \} in } e_1$$

These rules assume no variable shadowing. To illustrate, the definition of `falseCase` becomes:

```
falseCase t n =
  let {x0 = tri x1 (+); x1 = 1 x2; x2 = n (-)} in n x0
```

It is easy to see the number and length of applications in an expression graph. For example, `falseCase` contains four applications and its longest application, `tri x1 (+)`, has length three.

2.7 Template Code

We are now very close to the *template code* that can be executed by the Reduceron. We shall define template code as a Haskell data type, paving the way for an executable semantics to be defined in the next section. To highlight the semantics, each semantic definition is prefixed with a '`>`' symbol.

In template code, a program is defined to be a list of templates.

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

A template represents a function definition. It contains an *arity*, a *spine application* and a list of *nested applications*.

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

The spine application holds the let-body of a definition's expression graph and the nested applications hold the let-bindings. Applications are flat and are represented as a list of *atoms*.

```
> type App = [Atom]
```

An atom is a small, tagged piece of non-recursive data, defined in Figure 2. The following paragraphs define how programs are translated to template code.

Functions Given a list of function definitions

$$f_0\ \vec{x}_0\ \texttt{=}\ e_0,\ \cdots,\ f_n\ \vec{x}_n\ \texttt{=}\ e_n$$

each function identifier f_i occurring in $e_0 \cdots e_n$ is translated to an atom `FUN` $\#f\ i$ where $\#f$ is the arity of function f.

Arguments In each definition $f\ x_0 \cdots x_n\ \texttt{=}\ e$, each variable x_i occurring in e is translated to an atom `ARG` i.

Let-Bound Variables In each expression graph

$$\texttt{let \{ } x_0 \texttt{ = } e_0 \texttt{ ; } \cdots \texttt{ ; } x_n \texttt{ = } e_n \texttt{ \} in } e$$

each x_i occurring in $e, e_0 \cdots e_n$ is translated to an atom `PTR` i.

Integers, Primitives, and Constructors An integer literal n, a primitive p, and a constructor C_i are translated to atoms `INT` n, `PRI` p, and `CON` $\#C_i\ i$ respectively.

Case Tables Given a list of function definitions

$$f_0\ \vec{x}_0\ \texttt{=}\ e_0,\ \cdots,\ f_n\ \vec{x}_n\ \texttt{=}\ e_n$$

each case table `<`f_i`,` $\cdots f_j$`>` occurring in $e_0 \cdots e_n$ is translated to an atom `TAB` i. We assume that the functions in each case table are defined contiguously in the program.

Example The template code for the program

```
main = tri 5
tri n = let x = n (<=) in 1 x <falseCase, trueCase> n
falseCase t n =
  let {x0 = tri x1 (+); x1 = 1 x2; x2 = n (-)} in n x0
trueCase t n = 1
```

is as follows.

```
> tri5 :: Prog
> tri5 = [ (0, [FUN 1 1, INT 5], [])
>        , (1, [INT 1, PTR 0, TAB 2, ARG 0],
>              [[ARG 0, PRI "(<=)"]])
>        , (2, [ARG 1, PTR 0],
>              [[FUN 1 1, PTR 1, PRI "(+)"],
>               [INT 1, PTR 2],
>               [ARG 1, PRI "(-)"]])
>        , (2, [INT 1], []) ]
```

3. Operational Semantics

This section defines a *small-step operational semantics* for the Reduceron. There are two main reasons for presenting a semantics: (1) to define precisely how the Reduceron works; and (2) to highlight the low-level parallelism present in graph reduction that is exploited by the Reduceron. We have found it very useful to encode the semantics directly in Haskell. Before we commit to a low-level implementation, we can assess the complexity and performance of different design decisions and optimisations.

At the heart of the semantic definition is the small-step state transition function

```
> step :: State -> State
```

where the state is a 4-tuple comprising a program, a heap, a reduction stack, and an update stack.

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

The heap is modelled as a list of applications, and can be indexed by a heap-address.

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

An element on the heap can be modified using the update function.

```
> update :: HeapAddr -> App -> Heap -> Heap
> update i a as = take i as ++ [a] ++ drop (i+1) as
```

The reduction stack is also modelled as a list of nodes, with the top stack element coming first and the bottom element coming last.

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

There is also an update stack.

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

The meaning of a program p is defined by `run` p where

```
> run :: Prog -> Int
> run p = eval initialState
>   where initialState = (p, [], [FUN 0 0], [])

> eval (p, h, [INT i], u) = i
> eval s = eval (step s)
```

The initial state of the evaluator comprises a program, an empty heap, a singleton stack containing a call to `main`, and an empty update stack. The `main` template has arity 0 and is assumed to be the template at address 0. To illustrate, `run tri5` yields `15`. In the following sections, the central `step` function is defined.

3.1 Primitive Reduction

The `prim` function applies a primitive function to two arguments supplied as fully-evaluated integers.

```
> prim :: String -> Atom -> Atom -> Atom
> prim "(+)" (INT n) (INT m) = INT (n+m)
> prim "(-)" (INT n) (INT m) = INT (n-m)
> prim "(<=)" (INT n) (INT m) = bool (n<=m)
```

The comparison primitive returns a boolean value. Both boolean constructors have arity 0; `False` has index 0 and `True` has index 1.

```
> bool :: Bool -> Atom
> bool False = CON 0 0
> bool True = CON 0 1
```

3.2 Normal Forms

The number of arguments demanded by an atom on top of the reduction stack is defined by the `arity` function.

```
> arity :: Atom -> Arity
> arity (FUN n i) = n
> arity (INT i) = 1
> arity (CON n i) = n+1
> arity (PRI p) = 2
```

To reduce an integer, the evaluator demands one argument as shown in rewrite rule (2). And to reduce a constructor of arity n, the evaluator requires $n + 1$ arguments (the constructor's arguments *and* the case table) as shown in rewrite rule (3).

The arity of an atom is only used to detect when a *normal form* is reached. A normal form is an application of length n whose first atom has arity $\geq n$.

Some functions, such as case-alternative functions, are statically known *never* to be partially-applied, so they cannot occur as the first atom of a normal form. Such a function, say with address n, can be represented by the atom `FUN` 0 n.

3.3 Step-by-Step Reduction

There is one reduction rule for each possible type of atom that can appear on top of the reduction stack.

Unwinding If the top of the reduction stack is a pointer x to an application on the heap, evaluation proceeds by *unwinding*: copying the application from the heap to the reduction stack where it can be reduced. We must also ensure that when evaluation of the application is complete, the location x on the heap can be updated with the result. So we push onto the update stack the heap address x and the current size of the reduction stack.

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

Updating Evaluation of an application is known to be complete when an argument is demanded whose index is larger than n, the difference between the current size of the reduction stack and the stack address of the top update. If this condition is met, then a normal form of arity n is on top of the reduction stack and must be written to the heap.

```
> 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
```

Integers and Primitives Integer literals and primitive functions are reduced as described in §2.3.

```
> 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)
```

Constructors Constructors are reduced by indexing a case table, as described in §2.4.

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

There is insufficient information available to compute the arity of the case-alternative function at address `i+j`. However, an arity of zero can be used because a case-alternative function is statically known not to be partially applied (§3.2).

Function Application To apply a 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, and the remaining applications are instantiated and appended to the heap.

```
> 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
```

Instantiating a function body involves replacing the formal parameters with arguments from the reduction stack and turning relative pointers into absolute ones.

```
> instApp :: Stack -> Heap -> App -> App
> instApp s h = map (inst s (length h))
```

```
> inst :: Stack -> HeapAddr -> Atom -> Atom
> inst s base (PTR p) = PTR (base + p)
> inst s base (ARG i) = s !! i
> inst s base a = a
```

4. Implementation

We now refine the semantic definition to an actual implementation that runs on an FPGA. Specifically, our target is a mid-range Xilinx Virtex-5 released in 2008. Our guiding design principle is to perform as much reduction as possible in each clock-cycle. Our implementation performs each semantic reduction rule in *a single clock-cycle*, and clocks at a modest but respectable frequency for processor-like FPGA designs.

4.1 Low-Level Parallelism

Below we motivate three main opportunities for parallelism that we exploit in our implementation.

Parallel Memories The state of the reduction machine comprises four independent memory regions: the program, the heap, the reduction stack and the update stack. Most reduction rules refer to and modify more than one memory region. For example, the reduction rule for unwinding writes to both the reduction stack and the update stack. If the four memory regions are implemented as four separate memory units then they can be accessed in *parallel*, avoiding contentions that would arise if they were all stored in a single memory unit.

Wide Memories Many of the reduction rules involve transferring applications to and from memory. If a memory only allows one atom to be accessed at a time, transferring a single application involves *several* memory accesses. If memories are wide enough to allow a whole application to be accessed at a time, transferring an application needs only a *single* memory access.

Parallel Instantiation The reduction rule for function application involves instantiating each application in a function body and appending it to the heap. Each atom in an application can be instantiated in parallel, as indicated by the use of `map` in the definition of `instApp`. The wide heap then allows the instantiated application to be written in one memory access. Further, each application in a function body can also be instantiated in parallel, as indicated by the use of `map` in semantic rule for function application. If more than one application can be appended to the heap at a time, *parallel* instantiation of applications is possible.

4.2 Bounded Template Instantiation

Maximum Application Length Ideally, we would have a wide enough data bus to transfer *any* entire application in one go. However, this is an impossibility without some upper bound on the length of an application. Therefore, we introduce a bound, *MaxAppLen*, on the number atoms that can occur in an application.

To deal with an application whose length is larger than *MaxAppLen*, we split it into two or more smaller ones. For example, if *MaxAppLen* is 3 the application $f\ a\ b\ c\ d\ e$ can be bracketed $((f\ a\ b)\ c\ d)\ e$ resulting in three applications rather than one.

An alternative way to bound application length is to split applications into chunks that are aligned contiguously in memory, with the final chunk especially tagged by an end-marker. This approach [Naylor and Runciman 2007] is more efficient in some cases, but it cannot be expressed as a core-language transformation.

Maximum Spine Length Spine applications are special because, during function application, they are written to the stack, not the heap. So it is fine for spine applications to have a different maximum length: *MaxSpineLen*.

Maximum Applications per Template Ideally, all applications in a template would be instantiated in parallel. To allow for such an implementation, we introduce a bound, *MaxAppsPerBody*, on the maximum number of applications that can occur in a template body. To deal with templates containing more applications than *MaxAppsPerBody*, we employ a technique called *template splitting*.

Template Splitting We explain template splitting by example. Consider the following template, representing the `falseCase` function occurring in the `tri5` program defined in §2.7.

```
(2, [ARG 1, PTR 0]                    -- Spine
  , [ [FUN 1 1, PTR 1, PRI "(+)"]     -- Application 1
    , [INT 1, PTR 2]                  -- Application 2
    , [ARG 1, PRI "(-)"] ] )          -- Application 3
```

It contains one spine application and three nested applications. If *MaxAppsPerBody* is two then this template is split into two sub-templates. The first sub-template

```
(0, [FUN 0 4]                    -- Intermediate spine
  , [ [FUN 1 1, PTR 1, PRI "(+)"]   -- Application 1
    , [INT 1, PTR 2] ] )            -- Application 2
```

replaces the original template in the `tri5` program. The second sub-template is appended to the program at the next free program address: address four in the case of the `tri5` program.

```
(2, [ARG 1, PTR (-2)]                 -- Spine
  , [ [ARG 1, PRI "(-)"] ])           -- Application 3
```

The spine of the first sub-template is simply a call to the second sub-template. There are three important points to note:

- The first sub-template contains three applications, which is still larger than *MaxAppsPerBody*. However, at the implementation level, we do not count a spine application of the form `[FUN 0` f`]` as an application: it can be interpreted simply as "jump to template f", and does not entail any heap or stack accesses.
- In the second sub-template, each atom of the form `PTR` n is replaced by `PTR` $(n-2)$ to account for the fact that instantiation of the first sub-template will have increased the size of the heap by two.
- The arity of the first sub-template is set to zero: no elements are popped from the stack since they may be required by the second sub-template.

Choosing the Bounds We must choose the values of the bounds *MaxAppLen*, *MaxSpineLen*, and *MaxAppsPerBody* carefully: making them too low prevents useful parallelism; making them too high wastes resources. Our choices are informed by experiment. Table 1 shows the performance effect of varying each parameter in turn – non-varying parameters are effectively defined as infinity. The reduction count and heap usage figures are normalised across the varying parameter and averaged across a range of benchmark programs (see §6.1). The measurements are obtained using a PC implementation of the operational semantics. The reduction count represents the number of times that the `step` function is applied in the definition of `eval`[1].

The chosen bounds are: *MaxAppLen* $= 4$, *MaxSpineLen* $= 6$, and *MaxAppsPerBody* $= 2$. The measurements suggest a *MaxAppLen* of three is preferable to four due to better heap usage; the choice of four is motivated by another implementation parameter – the arity limit – introduced in §4.3. A *MaxSpineLen* of five would not be much worse than six, but the choice of six does not cost much extra at the implementation level. A *MaxAppsPerBody* of two is motivated by the fact that three would not be much better and that two fits nicely with the dual-port memories available on the FPGA.

Memory Unit	Element	Bits/Element	Elements
Program	`Template`	234	1k
Heap	`App`	77	32k
Reduction Stack	`Atom`	18	8k
Update Stack	`Update`	28	4k
Case-Table Stack	`Atom`	18	4k
Copy Space	`App`	77	16k

Table 2. Size and type of each parallel memory unit.

4.3 Memory Layout

Our Xilinx Virtex-5 FPGA contains 296 dual-port block RAMs each with a capacity of 18 kilobits giving a total on-chip RAM capacity of 5,328 kilobits. Each block RAM is dual-port allowing two independent accesses per clock-cycle. The data-bus and address-bus widths of each block-RAM are configurable. Possible configurations include 1k-by-18bit and 16k-by-1bit, and a range of possibilities in-between. Two 18 kilobit block RAMs can be merged to give further possible configurations ranging from 1k-by-36bit to 32k-by-1bit.

For simplicity, our implementation uses FPGA block RAMs only; no off-chip RAMs are used. This represents a tight constraint on the amount of memory available to the implementation. (The possibility of introducing off-chip memories is discussed in §7.)

Memory Structure The parallel memory units, each built out of block RAMs, are listed in Table 2 along with their capacities and the type of element stored at every addressable location. Note that there are uniform sizes for every program template and for every heap application. The two memory units at the bottom of the table are introduced in §4.5 and §4.6 respectively.

Wide Memories The wide heap memory is implemented by concatenating the data-busses of 77 32k-by-1bit block RAMs and merging their address-busses. This is done on both ports of each block RAM, making a *dual-port heap*. Similarly, the wide program memory is implemented using 13 1k-by-18bit block RAMs, but this time the dual-port capability is not needed.

Stack Memories We store the top N stack elements in special-purpose stack registers. In any given clock-cycle, the stack implementation allows: the top N elements to be observed; *and* up to N elements to be popped off; *and* up to N elements to be pushed on. If pushing and popping occur the same clock-cycle, the pop is

[1] Constructor reductions are not counted, anticipating the optimisation presented in §4.5.

MaxAppLen	Reductions	Heap	*MaxSpineLen*	Reductions	Heap	*MaxAppsPerBody*	Reductions
2	1.00	1.00	2	1.00	1.00	1	1.00
3	0.84	1.00	3	0.82	0.76	2	0.89
4	0.83	1.30	4	0.76	0.67	3	0.85
5	0.82	1.57	5	0.71	0.60	4	0.85
6	0.82	1.89	6	0.70	0.57		

Table 1. Effect of application-length, spine-length, and applications-per-template bounds on reduction count and heap usage.

performed before the push. Simultaneous access to the top N elements of the stack is achieved by a crossbar switch. It requires over 2,000 logic gates, but this is less than 1% of our FPGA's logic-gate capacity. There is a lot of parallelism in a crossbar, so the investment is worth it. Further hardware-level implementation details of the stack implementation are available in [Memo 27].

Arity Limit The stack implementation is parameterised by N, but requires N to be a power of two. For the update stack, N is defined to be 1 since reading and writing multiple values is of no benefit. For the reduction stack, there are three considerations to take into account, bearing in mind the aim of single-cycle reduction: (1) only the top N stack elements are available in any clock-cycle, hence the maximum number of arguments that be taken by a function is $N-1$; (2) the maximum length of a partially-applied function application, or normal-form, is therefore $N-1$; and (3) the choice of N should allow a normal form of length $N-1$ to be written onto the heap in a single clock cycle. As two applications of length $MaxAppLen$ can be written to the dual-port heap per clock-cycle, and $MaxAppLen$ is four, a sensible choice for N is eight since a normal form of length seven can be bracketed perfectly into two applications of length four.

To deal with functions taking more than $N-1$ arguments, an abstraction algorithm can be used [Turner 1979]. We have developed a minor variant [Memo 12] of an abstraction algorithm based on director strings [Dijkstra 1980, Kennaway and Sleep 1988] which uses a more coarse-grained combinator set than Turner's algorithm.

4.4 One Reduction per Clock-Cycle

Heap and program memory units have the following two properties.

- If a memory location x is read in clock-cycle n, the value at address x becomes available on the memory's data bus on clock-cycle $n+1$.
- If a value is written to memory location x in clock-cycle n, the new value at address x is not apparent until clock-cycle $n+1$.

The top stack elements are always observable without any clock-cycle delay. Now we show how each reduction rule in the semantics can be performed in a single clock-cycle, with reference to the following two invariants.

> **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.

Unwinding The top of the reduction stack has the form `PTR` x. So the application currently on the heap's data bus, say *app*, is the application at heap address x (Invariant 1). The following memory transactions are performed in parallel in a single clock-cycle:

- the application *app* is pushed onto the reduction stack;
- an update (n, x) is pushed onto the update stack where n is the size of the reduction stack before modification; and
- the first atom of *app* is the new top of the reduction stack and is used to lookup heap and program memory in order to maintain Invariants 1 and 2.

Updating The update stack's data bus is used to determine if an update is required, and if so, at what heap address x. If an update is required, then a normal form is available on the reduction stack's data bus. The following memory transactions are performed in parallel in a single clock-cycle:

- if the normal form has length less than or equal to four it is written to the heap at address x;
- if the normal form has length larger than four, it is bracketed into two applications of maximum length four one of which is written to the heap at address x, and the other of which is appended to the heap;
- the top element of the update stack is popped; and
- a program lookup is performed to preserve Invariant 2.

A heap lookup to preserve Invariant 1 is not necessary since the top of the reduction stack cannot possibly be of the form `PTR` x if an update is being performed. So updating requires at most two heap accesses, which can be done parallel thanks to dual-port memory.

Integers, Primitives, and Constructors Each of these reduction rules involves a pure stack manipulation, and each straightforwardly consumes a single clock-cycle.

Function Application The top of the reduction stack has the form `FUN` n f. So the template of f, say t, is available on the data bus (Invariant 2). There are two cases to consider.

Case 1: If t contains a spine application of the form [`FUN` 0 f], then:

- *up to two* nested applications in t are instantiated and appended to the heap;
- the atom `FUN` 0 f is written to the top of the reduction stack; and
- function f is looked-up in program memory in order to preserve Invariant 2.

Case 2: If t is of some other form, then:

- *zero or one* nested applications in t are instantiated and appended to the heap;
- the spine application in t is instantiated and written to the reduction stack; and
- the first element of the instantiated spine is used to lookup heap and program memory to preserve Invariants 1 and 2.

In Case 1, a heap lookup to preserve Invariant 1 is not required: the top of the stack is known to be a `FUN`, not a `PTR`. Thus in each case, at most two heap access are required.

4.5 The Case-Table Stack

Constructor reduction modifies only the top element of the reduction stack by adding the index of the constructor to the address of a case table. This addition is *almost* cheap enough to be implemented

in combinatorial logic (i.e. in zero clock-cycles) without affecting the critical path delay of the circuit. The problem is that the case table must be fetched from *a variable position* on the stack. This requires a multiplexer, making the combinatorial logic more expensive.

To solve this problem, we introduce a new stack memory to store case tables. When unwinding an application containing a case table, the case table is pushed onto the *case-table stack*. When performing constructor reduction, the case table of interest is always in the same position: the *top* of the case-table stack.

Table 3 shows the impact of various optimisations on clock-cycle count and heap usage across a range of benchmark programs. The in-lining strategy defined in §2.5 and the case-table optimisation both result in significant performance gains on average. The other optimisations in Table 3 are introduced in §5.

4.6 Garbage Collection

Our implementation employs a simple two-space stop-and-copy garbage collector [Jones and Lins 1996]. Although a two-space collector may not make the best use of limited memory resources, it does have the attraction of being easy to implement. In particular, the algorithm is easily defined iteratively so that no recursive call stack is needed.

4.7 Hardware Description

The Reduceron is described entirely in around 2,000 lines of York Lava [Naylor et al. 2009], a hardware description language embedded in Haskell. A large proportion of the description deals with garbage collection and the bit-level encoding of template code; the actual reduction rules account for less than 400 lines.

The Reduceron description is quite different to other reported Lava applications. It combines *structural and behavioural* description styles. Behavioural description brings improved *modularity* to our description. We associate each reduction rule with the memory transactions it performs, rather than associating each memory unit with all the memory transactions performed on it. So each reduction rule can be expressed in isolation.

The behavioural description language, called Recipe and included with York Lava, takes the form of a 300 line Lava library. It provides mutable variables, assignment statements, sequential and parallel composition, conditional and looping constructs, and shared procedure calls. In addition, it uses the results of a simple timing analysis, implemented by abstract interpretation, to enable optimisations.

4.8 Synthesis Results

Synthesising our implementation on a Xilinx Virtex-5 LX110T (speed-grade 1) yields an FPGA design using 14% of available logic slices and 90% of available block RAMs. The maximum clock frequency after place-and-route is 96MHz. By comparison, Xilinx distributes a hand-optimised RISC soft-processor called the MicroBlaze that clocks at 210MHz on the same FPGA. However, as the Reduceron performs a lot of computation per clock-cycle, 96MHz seems respectable. Furthermore, the MicroBlaze supports up to five pipeline stages, whereas the Reduceron is *not pipelined*.

5. Optimisations

This section presents several optimisations, defined by a series of progressive modifications to the semantics defined in §3. A theme of this section is the use of *cheap dynamic analyses* to improve performance.

5.1 Update Avoidance

Recall that when evaluation of an application on the heap is complete, the heap is updated with the result to prevent repeated evaluation. There are two cases in which such an update is unnecessary: (1) the application is *already evaluated*, and (2) the application is *not shared* so its result will never be needed again.

We identify non-shared applications at *run-time*, by *dynamic* analysis. Argument and pointer atoms are extended to contain an extra boolean field.

```
> data Atom = ··· | ARG Bool Int | PTR Bool Int | ···
```

An argument is tagged with `True` exactly if it is referenced more than once in the body of a function. A pointer is tagged with `False` exactly if it is a *unique pointer*; that is, it points to an application that is not pointed to directly by any other atom on the heap or reduction stack. There may be multiple pointers to an application containing a unique pointer, so the fact that a pointer is unique is, on its own, not enough to infer that it points to a non-shared application. To identify non-shared applications, we maintain the invariant:

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

A pointer that is not unique is referred to as *possibly-shared*.

Unwinding The reduction rule for unwinding becomes

```
> step (p, h, PTR sh x:s, u) = (p, h, app++s, upd++u)
>   where
>     app = map (dashIf sh) (h!!x)
>     upd = [(1+length s, x) | sh && red (h!!x)]
```

If the pointer on top of the stack is possibly-shared, then the application is *dashed* before being copied onto the stack by marking each atom it contains as possibly-shared. This has the effect of propagating sharing information through an application.

```
> dashIf sh a = if sh then dash a else a

> dash (PTR sh s) = PTR True s
> dash a = a
```

If the pointer on top of the stack is unique, the application it points to must be non-shared according to Invariant 3. An update is only pushed onto the update stack if the pointer is possibly-shared and the application is *reducible*. An application is reducible if it is saturated or its first atom is a pointer.

```
> red :: App -> Bool
> red (PTR sh i:xs) = True
> red (x:xs) = arity x <= length xs
```

Updating When an update occurs, the normal-form on the stack is written to the heap. The normal-form may contain a unique pointer, but the process of writing it to the heap will duplicate it. Hence the normal-form on the stack is dashed.

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

> dashN n s = map dash (take n s) ++ drop n s
```

It is unnecessary to dash the normal-form that is written to the heap, but there is no harm in doing so: the application being updated is possibly-shared, and a possibly-shared application will anyway be dashed when it is unwound onto the stack.

Function Application When instantiating a function body, shared arguments must be dashed as they are fetched from the stack.

```
> inst s base (PTR sh p) = PTR sh (base + p)
> inst s base (ARG sh i) = dashIf sh (s!!i)
> inst s base a = a
```

Program	Baseline		+In-lining		+Case Stack		+Update Avoid.		+Infix Prims.		+PRS	
	Time	Heap	Time	Heap	Time	Heap	Time	Heap	Time	Heap	Time	Heap
Adjoxo	1.00	1.00	0.85	0.80	0.71	0.80	0.54	0.80	0.43	0.49	0.36	0.41
Braun	1.00	1.00	0.84	0.93	0.63	0.93	0.46	0.93	0.43	0.88	0.42	0.88
Cichelli	1.00	1.00	0.93	0.97	0.77	0.97	0.56	0.97	0.42	0.36	0.41	0.33
Clausify	1.00	1.00	0.79	0.59	0.59	0.59	0.48	0.59	0.41	0.42	0.41	0.42
CountDown	1.00	1.00	0.95	0.97	0.86	0.97	0.70	0.97	0.49	0.53	0.31	0.33
Fib	1.00	1.00	1.28	2.33	1.21	2.33	0.96	2.33	0.75	2.00	0.35	0.33
KnuthBendix	1.00	1.00	0.81	0.66	0.63	0.66	0.48	0.66	0.43	0.58	0.40	0.49
Mate	1.00	1.00	0.83	0.45	0.67	0.45	0.50	0.45	0.43	0.29	0.40	0.25
MSS	1.00	1.00	0.92	1.00	0.84	1.00	0.61	1.00	0.38	0.51	0.24	0.03
OrdList	1.00	1.00	0.73	0.67	0.55	0.67	0.42	0.67	0.42	0.67	0.42	0.67
PermSort	1.00	1.00	0.77	0.77	0.62	0.77	0.48	0.77	0.42	0.69	0.42	0.69
Queens	1.00	1.00	0.75	0.54	0.68	0.54	0.51	0.54	0.40	0.39	0.21	0.11
Queens_2	1.00	1.00	0.82	0.92	0.67	0.92	0.55	0.92	0.50	0.77	0.50	0.73
SumPuz	1.00	1.00	0.95	1.06	0.80	1.06	0.60	1.05	0.50	0.74	0.48	0.63
Taut	1.00	1.00	0.90	0.99	0.70	0.99	0.56	0.99	0.51	0.90	0.50	0.87
While	1.00	1.00	0.93	0.95	0.77	0.95	0.58	0.95	0.50	0.81	0.49	0.80
Average	1.00	1.00	0.88	0.92	0.74	0.92	0.57	0.92	0.47	0.69	0.40	0.50

Table 3. Impact of optimisations on clock-cycle count and heap usage across a range of programs.

Performance Table 3 shows that, overall, update avoidance offers a significant run-time improvement. On average, 88% of all updates are avoided across the 16 benchmark programs. Just over half of these are avoided due to non-reducible applications, and just under half of them are avoided due to non-shared reducible applications. The average maximum update-stack usage drops from 406 to 11.

5.2 Infix Primitive Applications

For every binary primitive function p, we introduce a new primitive $*p$, a version of p that expects its arguments flipped.

```
> prim ('*':p) n m = prim p m n
```

Any primitive function p can be flipped.

```
> flip ('*':p) = p
> flip p = '*':p
```

Now we translate binary primitive applications by the rule

$$p\ m\ n \ \rightarrow\ m\ p\ n \tag{4}$$

In place of the existing reduction rules for primitives and integers, we define:

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

If both arguments are already evaluated, the primitive is applied. If only the first argument is evaluated, then the arguments are swapped and the primitive is flipped.

Note that compilation rule (4) could just as sensibly be

$$p\ m\ n \ \rightarrow\ n\ *p\ m \tag{5}$$

In the interest of efficiency, the choice between (4) and (5) is informed for each primitive application by compile-time knowledge of whether m or n is expected to be already-evaluated.

Example Consider the steps needed to evaluate `(+)` e_0 e_1. Using the approach to primitive reduction of §3.1, the application is translated to e_1 (e_0 `(+)`) at compile time. At run-time, in four successive clock-cycles:

- an integer reduction is required after e_1 is evaluated;
- an unwinding is required to fetch argument e_0 `(+)` from heap;
- an integer reduction is required after e_0 is evaluated; and
- a primitive reduction is required.

With the new approach, the application is translated to e_0 `(+)` e_1 at compile-time. At run-time, in two successive clock-cycles:

- an integer reduction is required after e_0 is evaluated; and
- a primitive reduction is required after e_1 is evaluated.

Also note that e_0 `(+)` e_1 comprises one application whereas e_1 (e_0 `(+)`) comprises two, so the former is cheaper to instantiate. Table 3 shows run-time and heap-usage improvements brought by the new approach.

5.3 Speculative Evaluation of Primitive Redexes

Consider evaluation of the expression `tri 5`. Application of `tri` yields the expression

```
case (<=) 5 1 of
  { False -> (+) (tri ((-) 5 1)) 5 ; True -> 1 }
```

which contains two *primitive redexes*: `(<=) 5 1` and `(-) 5 1`. This section introduces a technique called primitive-redex speculation (PRS) in which such redexes are evaluated during function body instantiation. For example, application of `tri` instead yields

```
case False of { False -> (+) (tri 4) 5 ; True -> 1 }
```

The benefit is that primitive redexes need not be constructed in memory, nor fetched again when needed. Even if the result of a primitive redex is *not* needed, reducing it is no more costly than constructing it. We identify primitive redexes at *run-time*, by *dynamic* analysis.

Register File To support PRS, we introduce a *register file* to the reduction machine, for storing the results of speculative reductions.

```
> type RegFile = [Atom]
```

The body of a function may refer to these results as required.

```
> data Atom = ··· | REG Bool Int
```

An atom of the form `REG` b i contains a reference i to a register, and a boolean field b that is true exactly if there is more than one reference to the register in the body of the function.

The instantiation functions `inst` and `instApp` are modified to take the register file r as an argument, and the following equation is added to the definition of `inst`.

```
> inst s r base (REG sh i) = dashIf sh (r !! i)
```

Waves The primitive redexes in a function body are evaluated in a series of *waves*. To illustrate, consider `(+) 1 ((+) 2 3)`. In the first wave of speculative evaluation, `(+) 2 3` would be reduced to `5`; in the second wave, `(+) 1 5` would be reduced to `6`.

More specifically, a wave is a list of *independent* primitive redex *candidates*. A primitive redex candidate is an application which may turn out at run-time to be a primitive redex. Specifically, it is an application of the form `[`a_0`, PRI` p`,` a_1`]` where a_0 and a_1 are `INT`, `ARG` or `REG` atoms.

```
> type Wave = [App]
```

Templates are extended to contain a list of waves in which no application in a wave depends on the result of an application in the same or a later wave.

```
> type Template = (Arity, App, [App], [Wave])
```

Given the reduction stack, the heap, and a series of waves, PRS produces a possibly-modified heap, and one result for each application in each wave.

```
> prs :: Stack -> Heap -> [Wave] -> (Heap, RegFile)
> prs s h = foldl (wave s) (h, [])

> wave s (h,r) = foldl spec (h,r) . map (instApp s r h)
```

If a primitive redex candidate turns out to be a primitive redex at run-time, it is reduced, and its result is appended to the register file. Otherwise, the candidate application is constructed on the heap, and a pointer to this application is appended to the register file.

```
> spec (h,r) [INT m,PRI p,INT n] = (h, r++[prim p m n])
> spec (h,r) app = (h++[app], r++[PTR False (length h)])
```

Function Application Since applications in a function body may refer to the results in the PRS register file, PRS is performed before instantiation of the body. The new rule is:

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

The template splitting technique outlined in §4.2 is modified to deal with waves of primitive redex candidates. Each wave is split into a separate template. If a wave contains more than *MaxAppsPerBody* applications, it is further split in order to satisfy the constraint.

Strictness Analysis PRS works well when recursive call sites sustain *unboxed* arguments[2]. For example, if a call to `tri` is passed an unboxed integer then, thanks to PRS, so too is the recursive call. However, if the initial call is passed a boxed expression, primitive redexes never arise, e.g. the outer call in `tri (tri 5)` is passed a pointer to an application, inhibiting PRS.

A basic strictness analyser in combination with the worker-wrapper transformation [Gill and Hutton 2009] alleviates this problem. Each initial call to a recursive function is replaced with a call to a wrapper function. The wrapper applies a special primitive to force evaluation of any strict integer arguments before passing them on to the recursive worker.

Performance Table 3 shows how PRS cuts run-time and heap-usage over the range of benchmark programs. On average, the maximum stack usage drops from 811 to 104, and 85% of primitive redex candidates turn out to be primitive redexes.

[2] An unboxed integer is an integer literal `INT` n as opposed to a pointer `PTR` x to an expression of type integer.

Program	Lines	GHC -O2 Run-time	Clean Run-time	Hand reds. per Cycle
Adjoxo	108	0.18	0.26	0.65
Braun	51	0.35	0.29	0.46
Cichelli	200	0.17	0.12	0.52
Clausify	132	0.33	0.27	0.54
CountDown	120	0.42	0.26	0.62
Fib	10	0.14	0.14	0.90
KnuthBendix	551	0.37	0.21	0.47
Mate	393	0.10	0.16	0.52
MSS	47	0.17	0.11	0.65
OrdList	46	0.64	–	0.43
PermSort	39	0.43	0.37	0.43
Queens	49	0.17	0.38	0.72
Queens_2	62	0.34	0.31	0.40
SumPuz	158	0.23	0.21	0.46
Taut	97	0.32	0.16	0.46
While	96	0.27	0.19	0.49
Average	135	0.29	0.23	0.55

Table 4. Normalised run-times of GHC and Clean compiled code running on an Intel Core 2 Duo E8400 PC clocking at 3GHz. Run-times are relative to 1.00, the run-time of Reduceron running on a Xilinx Virtex-5 FPGA clocking at 96MHz (over 30x slower).

6. Comparative Evaluation

This section evaluates the Reduceron in the context of previous and current work on functional language implementation.

6.1 Benchmark programs

The performance of the Reduceron is measured using a set of 16 benchmark programs named in Table 4. The programs, though small (the largest is 551 lines), are diverse and fairly representative of functional programs in general. For details of the programs, including source code, see [Naylor et al. 2009].

6.2 Previous work on the Reduceron

Compared to our previous work on the Reduceron presented in [Naylor and Runciman 2007], the implementation described in this paper reduces the number of clock-cycles required to run the benchmark programs by an average factor of 6.4. As the previous implementation clocks at 111MHz, and the new one at 96MHz on the same FPGA, the raw speed-up factor is 5.5. The gains are mainly due to the combined impact of improved case-expression compilation, single-cycle reduction, and the optimisations listed in Table 3. But another factor is that the new implementation performs *spineless* evaluation [Burn et al. 1988]. During function application, the spine of a function body is only written onto the stack, reducing heap contention and heap usage. The spine is only ever written to the heap during updating, and even then, only if it is a possibly-shared normal-form. Spineless evaluation also avoids the problem of indirection chains, and is more modular in the sense that it allows function application to be conceptually separated from updating.

6.3 State of the Art

A run-time performance comparison of the Reduceron against state-of-the-art functional language implementations running on a 3GHz Intel Core 2 Duo PC is shown in Table 4[3].

Given the speed-up over our previous implementation of the Reduceron, we had hoped that the performance of our new implemen-

[3] The Clean-compiled version of the OrdList program does not terminate due to a bug in the Clean compiler.

tation would approach that of the PC implementations. However, new GHC optimisations and the use of a 3GHz Core 2 Duo instead of a 2.8GHz Pentium-4 have significantly boosted the PC results. (The Dhrystone MIPS (DMIPS) per MHz of the Core 2 Duo is almost twice that of the Pentium-4 [Longbottom 2009].)

It would be interesting to compare the Reduceron against GHC or Clean compiled programs running on an FPGA soft-processor such as the Xilinx MicroBlaze. Unfortunately, this experiment would be quite an undertaking since the run-time system of GHC or Clean would need to be ported to the FPGA environment. We can, however, point out that the Core 2 Duo achieves almost three times as many DMIPs per MHz as the Xilinx MicroBlaze [MicroBlaze], and clocks 14 times faster. So the performance ratio for this conventional benchmarking is around 42. The performance ratio between the Reduceron and the PC is an order of magnitude less.

Table 4 also shows *hand-reductions per clock-cycle*. A hand-reduction is the application of a function or a primitive function; it includes applications of functions introduced by case compilation, but does *not* include updating, unwinding, integer reduction, constructor reduction, or applications of functions introduced by template splitting.

6.4 Modern Processors

Modern microprocessors are the product of almost half a century of intensive engineering. Instruction pipelines with tens of stages have helped achieve clock frequencies in the region of 3-4GHz. Techniques such as dynamic branch prediction, out of order execution, and caching have enabled high utilisation of such deep pipelines.

The Reduceron represents a different kind of processor: a *vector* processor. Rather than process one word at a time, it processes several in parallel. It is *not* pipelined, so the sophisticated techniques needed to keep rapidly-clocked pipelines busy are not needed.

6.5 The G-machine

In [Peyton Jones 1987], template instantiation is presented as a "simple" first step towards a more sophisticated approach to graph reduction based on the *G-machine*. So why is the Reduceron based on template instantiation and not the G-machine?

The G-machine approach aims to generate good code for *conventional hardware*, exploiting its strengths and avoiding its weaknesses. We base the Reduceron on template instantiation precisely because it does *not* make assumptions about the target hardware. The G-machine executes a sequential stream of fine-grained instructions, many of which could in fact be executed in parallel. The FPGA negates the assumption that such a sequential stream of instructions is necessary to avoid interpretive overhead.

6.6 Manipulating Basic Values

One aspect of reduction that the G-machine approach aims to optimise is the processing of basic values such as integers. In particular, avoiding construction of strictly-needed primitive applications in memory can lead to large performance gains. For example, if a function body has the form `f ((+) x 1)` and `f` is strict then construction of `(+) x 1` on the heap can be avoided and instead reduced immediately.

The Reduceron can also avoid construction of primitive applications to good effect (§5.3). However, it discovers suitable primitive applications *at run-time* and evaluates them *speculatively*. The Reduceron allows construction of `(+) x 1` to be avoided regardless of whether or not `f` is strict, but only if `x`, at run-time, takes the form `INT` i.

So the conditions under which construction of primitive applications can be avoided are quite different between the two approaches. As discussed in §5.3, strictness analysis can aid PRS. But strictness analysis alone, without some mechanism for reducing primitive redexes cheaply, is of little use to the Reduceron. PRS provides such a mechanism.

6.7 The SKIM Machine

SKIM is a microcoded processor designed specifically to perform combinator reduction [Stoye 1985]. Stoye writes that "a combinator reducer coded on an 8-MHz 68000 goes at about one thirtieth of the speed of SKIM, and was considerably harder to write than SKIM's microcode".

One interesting aspect of SKIM is its use of *one-bit reference counts*. Stoye observes that such reference counts can be stored in the *pointer* to an application rather than in the application itself, making useful information about an application available without the expense of dereferencing a pointer. A reference count bit indicates whether the pointer is a *unique application pointer* or *multiple application pointer*. This information is used to good effect in SKIM by allowing the space pointed to by a unique pointer to be reused during reduction rather than discarded. On average about 70% of discarded cells are immediately reused.

SKIM's successful use of reference-count bits partly motivated the development of the dynamic sharing analysis presented in §5.1. We have precisely specified the modifications needed to implement dynamic sharing analysis in a general graph reduction machine. We also discuss two important details not mentioned by Stoye: (1) the subtle case in which an update can cause a unique pointer to become non-unique; and (2) Invariant 3, an important key to understanding why the technique actually works. We use the results of the analysis not for storage reclamation (which would complicate the machinery for template instantiation), but for update avoidance.

6.8 Static versus Dynamic Analysis

Sharing Analysis The idea to avoid updates by identifying non-shared applications is discussed in [Burn et al. 1988], including trade-offs between static and dynamic sharing analysis. The authors write that dynamic sharing analysis has the advantage of greater precision but that "in general we strongly suspect that the cost of dashing greatly outweighs the advantages of precision when compared to [static analysis]". In the Reduceron, dynamic sharing analysis (dashing) has *no time cost*: it is implemented in combinatorial logic that is not on the Reduceron's critical path. It is precise and simple to implement, requiring only minor modifications to three of the Reduceron's reduction rules.

Primitive Redex Analysis Primitive redexes can also be detected by static or dynamic analysis. In our experience, a dynamic approach is *simple* and *cheap* to implement in hardware, and works quite well. As an alternative, we are currently trying a static analysis to determine expressions whose every instance at run-time will be a primitive redex. The analysis can be combined with specialisation to increase the incidence of such expressions. Eliminating the logic and memory capacity needed to handle failed PRS candidates could significantly boost performance.

6.9 The Big Word Machine

A prototype machine similar in spirit to the Reduceron is Augustsson's Big Word Machine (BWM) [Augustsson 1992]. The BWM is a graph reduction machine with a wide word size, four pointers long, allowing wide applications to be quickly built on, and fetched from, the heap. Augustsson likens the BWM to a VLIW (very long instruction word) machine [Hennessy and Patterson 1992], designed for functional languages rather than scientific computing. Like the Reduceron, the BWM has a crossbar switch attached to the stack allowing complex rearrangements to be done in a single clock-cycle. The BWM also uses the Scott encoding to implement case expressions and constructors. Unlike the Reduceron,

the BWM works on an explicit instruction stream rather than by template instantiation. The BWM was never actually built. Some simulations were performed but Augustsson writes "the absolute performance of the machine is hard to determine at this point".

7. Conclusions and Future Work

Considering their relatively low clocking frequencies, FPGA applications must exploit significant parallelism to achieve high performance. In the context of sequential graph-reduction, we have taken this idea to its natural limit: each reduction rule is performed within one clock-cycle. Furthermore, upon synthesis our design achieves a respectable clock frequency compared to similar FPGA designs for the same device. It is therefore quite hard to see how the Reduceron's reduction rules could be performed more quickly.

On the other hand, there is a lot of scope to reduce the *number* of reductions performed in a given program run. To this end, update avoidance and speculative evaluation of primitive redexes are both effective, making use of simple and precise dynamic analyses. These dynamic analyses would have a prohibitive run-time overhead on a PC, but have no such overhead on an FPGA.

Compared to state-of-the-art functional language implementations running on a PC, the Reduceron implemented on a FPGA is on average around a factor of four slower. This difference may be disappointing, but it is an order of magnitude smaller than the typical performance gap between PC-based hard-processors and FPGA-based soft-processors.

Future Work The main limitation of the current Reduceron implementation is the small amount of heap memory it provides. Could the heap be implemented using a larger, off-chip memory unit? We believe it could, without loss of performance, and without significant modification to the existing design. Two possible options are: (1) the use of low-latency memory technologies such as RLDRAM, ZBT RAM, and QDR SRAM, commonly used by FPGA applications that require access to large amounts of memory; and (2) the use of buffers or caches, implemented using on-chip block RAM.

Functional languages offer much scope for *parallel* evaluation of expressions. On conventional architectures there is a high cost for operations such as locking and releasing expressions under evaluation, so the benefits of parallel evaluation are offset by significant communication overheads. It would be interesting to see if special-purpose hardware could be used to overcome such overheads. Multiple Reducerons could be synthesised to FPGA, coordinated for parallel graph reduction [Clack 1999].

One of the main features of FPGAs that we are not exploiting is that they can be configured on a *per-program* basis. One option would be to allow programmers to express, *as part of their program*, custom FPGA logic that accelerates execution of that program. Such logic would act as *co-processor* to the Reduceron, and could itself be suitably described in the functional source language.

The future development and competitiveness of special-purpose processors for graph reduction remains questionable. But within a few years, just as plug-in GPU cards are already used for high-performance graphics, we'd like to see FPU cards for high-performance applications of functional languages. We hope our work on the Reduceron makes a small advance in that direction.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council of the UK under grant EP/G011052/1. Thanks to Xilinx for donating the FPGA used in this work, and to Satnam Singh, Gabor Greif, and the anonymous ICFP reviewers for their helpful suggestions.

References

[Augustsson 1992] L. Augustsson. BWM: A Concrete Machine for Graph Reduction. In Proceedings of the 1991 Glasgow Workshop on Functional Programming, pages 36–50, Springer, 1992.

[Burn et al. 1988] G. L. Burn, S. L. Peyton Jones and J. D. Robson. The Spineless G-machine. In Proceedings of the 1988 Conference on Lisp and Functional Programming, pages 244-258, ACM, 1988.

[Clack 1999] C. Clack. Realisations for non-strict languages. In Research Directions in Parallel Functional Programming, pages 149187. Springer, 1999.

[Dijkstra 1980] E. W. Dijkstra. A mild variant of Combinatory Logic. EWD735, 1980.

[Fasel and Keller 1987] J. H. Fasel and R. M. Keller, editors. Graph Reduction, Proceedings of a Workshop. Springer LNCS 279, 1987.

[Flanagan et al. 1993] C. Flanagan, A. Sabry, and B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pages 237–247, ACM, 1993.

[Gill and Hutton 2009] A. Gill and G. Hutton. The Worker/Wrapper Transformation. JFP, volume 18, part 2, pages 227–251, 2009.

[Hennessy and Patterson 1992] J. Hennessy and D. Patterson. Computer Architecture; A Quantitative Approach. Morgan Kaufmann, 1992.

[Jansen et al. 2007] J. M. Jansen, P. Koopman, R. Plasmeijer. Efficient Interpretation by Transforming Data Types and Patterns to Functions. In Trends in Functional Programming, volume 7, pages 157–172, 2007.

[Jones and Lins 1996] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, 1996.

[Kennaway and Sleep 1988] R. Kennaway and R. Sleep. Director strings as combinators. ACM Transactions on Programming Languages and Systems, volume 10, number 4, pages 602–626, 1988.

[Longbottom 2009] R. Longbottom. Dhrystone Benchmark Results On PCs, November 2009. (`http://www.roylongbottom.org.uk/dhrystone%20results.htm`)

[Naylor and Runciman 2007] M. Naylor and C. Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In IFL'07, pages 129–146. Springer LNCS 5083, 2008.

[Naylor et al. 2009] M. Naylor, C. Runciman, and J. Reich. Reduceron home page. (`http://www.cs.york.ac.uk/fp/reduceron/`)

[Memo 9] M. Naylor. F-lite: a core subset of Haskell, 2008. (`http://www.cs.york.ac.uk/fp/reduceron/memos/Memo9.txt`)

[Memo 12] M. Naylor. An algorithm for arity-reduction, 2008. (`http://www.cs.york.ac.uk/fp/reduceron/memos/Memo12.lhs`)

[Memo 27] M. Naylor. Design of the Octostack, 2009. (`http://www.cs.york.ac.uk/fp/reduceron/memos/Memo27.lhs`)

[MicroBlaze] Xilinx. MicroBlaze Soft Processor v7.20, April 2009. (`http://www.xilinx.com/tools/microblaze.htm`)

[Peyton Jones 1987] S. L. Peyton Jones. The Implementation of Functional Programming Languages, Prentice Hall, 1987.

[Peyton Jones 1992] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. Journal of Functional Programming, volume 2, pages 127-202, 1992.

[Scheevel 1986] M. Scheevel. NORMA: a graph reduction processor. In Proceedings of the 1986 Conference on LISP and Functional Programming, pages 212-219. ACM, 1986.

[Stoye 1985] W. Stoye. The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge, 1985.

[Turner 1979] D. A. Turner. A New Implementation Technique for Applicative Languages. Software – Practice and Experience, volume 9, number 1, pages 31–49, 1979.

[Weicker 1984] R. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. Communications of the ACM, volume 27, number 10, pages 1013-1030, 1984.