The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA

Matthew Naylor and Colin Runciman

University of York, UK
{mfn,colin}@cs.york.ac.uk

Abstract. For the memory intensive task of graph reduction, modern PCs are limited not by processor speed, but by the rate that data can travel between processor and memory. This limitation is known as the *von Neumann bottleneck*. We explore the effect of *widening* this bottleneck using a special-purpose graph reduction machine with wide, parallel memories. Our prototype machine – the Reduceron – is implemented using an *FPGA*, and is based on a *simple* template-instantiation evaluator. Running at only 91.5MHz on an FPGA, the Reduceron is faster than mature bytecode implementations of Haskell running on a 2.8GHz PC.

1 Introduction

The processing power of PCs has risen astonishingly over the past few decades, and this trend looks set to continue with the introduction of multi-core CPUs. However, increased processing power does not necessarily mean faster programs! Many programs, particularly *memory intensive* ones, are limited by the rate that data can travel between the CPU and the memory, not by the rate that the CPU can process data.

A prime example of a memory intensive application is *graph reduction* [14], the operational basis of standard lazy functional language implementations. The core operation of graph reduction is *function unfolding*, whereby a function application $f\ a_1 \cdots a_n$ is reduced to a fresh copy of f's body with its free variables replaced by the arguments $a_1 \cdots a_n$. On a PC, unfolding a single function in this way requires the sequential execution of *many* machine instructions. This sequentialisation is merely a consequence of the PC's von Neumann architecture, *not* of any data dependencies in the reduction process.

In an attempt to improve upon the PC's overly-sequential approach to function unfolding, we develop a special-purpose graph reduction machine – the Reduceron – using an FPGA. Modern FPGAs contain hundreds of independent memory units called *block RAMs*, each of which can be accessed in parallel. The Reduceron *cascades* these block RAMs to form separate dual-port, quad-word memories for stack, heap and combinator storage, meaning that up to eight words can be transferred between two memories in a single clock cycle. Together

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 129–146, 2008.

with vectorised processing logic, the wide, parallel memories allow the Reduceron to rapidly execute the block read-modify-write memory operations that lie at the heart of function unfolding.

The Reduceron is only a *prototype* machine, based on a *simple* template-instantiation evaluator, and the Reduceron compiler performs *no optimisation*. Yet the results are promising. The *wide* implementation of the Reduceron runs six times faster than a single-memory, one-word-at-a-time version. And running at 91.5MHz on a Xilinx Virtex-II FPGA, the Reduceron is faster than mature bytecode implementations of Haskell running on a Pentium-4 2.8GHz PC.

This paper is structured as follows. Section 2 defines the bytecode that the Reduceron executes, and describes how Haskell programs are compiled down to it. Section 3 presents a small-step operational semantics of the Reduceron, pinpointing the parts of the evaluator that can be executed in parallel. Section 4 describes the FPGA implementation of the Reduceron. Section 5 presents performance measurements, comparisons and possible enhancements. Section 6 discusses related work, and section 7 concludes.

The source code for the Reduceron implementation is publicly available from `http://www.cs.york.ac.uk/fp/darcs/reduceron2`.

2 Compilation from Haskell to Reduceron Bytecode

There are two main goals of our compilation scheme. First, it should allow the Reduceron to be *simple*, so that the implementation can be constructed in good time. To this aim, we adopt the idea of Jansen to encode data constructors as functions and case expressions as function applications [9]. The result is that all data constructors and case expressions are eliminated, meaning fewer language constructs for the machine to deal with. One might expect to pay a price for this simplicity, yet Jansen's interpreter is rather fast in practice. It is believed that one reason for this good performance is that having fewer language constructs permits a simpler interpreter with less interpretive overhead.

The second goal of our compiler is to expose the *parallelism* present in sequential graph reduction. An earlier version of the Reduceron was based on Turner's combinators, so it performed only a small amount of work in each clock cycle. Our aim is to do lots of work in each clock cycle, so the coarser-grained *supercombinator* [8] approach to graph reduction is taken here.

Like Jansen's interpreter, the graph reduction technique used by the Reduceron is similar to what Peyton Jones calls *template instantiation* [14]. Peyton Jones introduces template instantiation as a "simple" first step towards a more sophisticated machine – the G-machine. In this light, the Reduceron might be seen as being too far from a "real" functional language implementation to produce meaningful results. But Jansen's positive results give good reason to be open-minded about this.

The remainder of this section describes in more detail the stages of compilation to get from Haskell programs to Reduceron bytecode. As a running example we use the following function for computing the factorial of a given integer:

```
fact :: Int -> Int
fact n = if n == 1 then 1 else n * fact (n-1)
```

We end this section by defining the Reduceron bytecode and showing how `fact` looks at the bytecode level.

2.1 Desugaring and Compilation to Supercombinators

The first stage of compilation is to translate the input program to *Yhc Core* [6] using the York Haskell Compiler [17]. The result is an equivalent but simplified program in which expressions contain only function names, function applications, variables, data constructions, case expressions, let expressions, and literals. All function definitions are *supercombinator* definitions. In particular, they do not contain any lambda abstractions. In our example, `fact` is already a supercombinator, but in Yhc Core its definition becomes:

```
fact n = case (==) n 1 of
           True -> 1
           False -> (*) n (fact ((-) n 1))
```

Here, infix applications have been made prefix, and the `if` expression has been desugared to a `case`.

2.2 Eliminating Data Constructors and Cases

The second stage eliminates all data constructions and case expressions from the program. First, each data type d of the form

$$\texttt{data}\ d\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n$$

is replaced by a set of function definitions, one for each data constructor c_i, of the form

$$c_i\ v_1\ \cdots\ v_{\#c_i}\ w_1\ \cdots\ w_n\ \texttt{=}\ w_i\ v_1\ \cdots\ v_{\#c_i}$$

where $\#c$ denotes the number of arguments taken by the constructor c. In words, each original data constructor c_i is encoded as a function that takes as arguments the $\#c_i$ arguments of c_i *and* n continuations stating how to proceed depending on the constructor's value.

Next, all default alternatives in case expressions are removed. Case expressions in Yhc Core already have the property that the pattern in each alternative is at most one constructor deep. So removing case defaults is simply a matter of enumerating all unmentioned constructors. Now each case expression has the form

$$\texttt{case}\ e\ \texttt{of}\ \{c_1\ v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1\ ;\ \cdots\ ;\ c_n\ v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n\}$$

and can be straightforwardly translated to a function application

$$e\ (\lambda v_1\ \cdots\ v_{\#c_1}\ \texttt{->}\ e_1)\ \cdots\ (\lambda v_1\ \cdots\ v_{\#c_n}\ \texttt{->}\ e_n)$$

Since this transformation reintroduces lambda abstractions, the lambda lifter is reapplied to make all function definitions supercombinators once again. After this stage of compilation, our factorial example looks as follows:

```
fact n = (==) n 1 1 ((*) n (fact ((-) n 1)))
```

2.3 Dealing with Strict Primitives

Further to user-defined algebraic data types, the Reduceron also supports, as *primitives*, machine integers and associated arithmetic operators. Under lazy evaluation, primitive functions, such as integer multiplication, need special treatment because their arguments must be fully evaluated before *before* they can be applied. Peyton Jones and Jansen both solve this problem by making their evaluators recursively evaluate each argument to a primitive. This is an elegant approach when the evaluator is written in a programming language like Miranda or C, where the presence of an implicit call stack may be assumed. But FPGAs have no such implicit call stack, so an alternative solution must be found.

Our solution is to treat primitive values in the same way as nullary constructors of an algebraic data type: they become functions that take a continuation as an argument. The idea is that the continuation states what to do once the integer has been evaluated, and it takes the fully evaluated integer as its argument. Transforming the program to obtain this behaviour is straightforward. Each two-argument primitive function application is rewritten by the rule

$$p\ n\ m \ \rightarrow \ m\ (n\ p)$$

The factorial function is now:

```
fact n = 1 (n (==)) 1 (fact (1 (n (-))) (n (*)))
```

2.4 Reduceron Bytecode

In the final stage of compilation, programs are turned into Reduceron bytecode. The bytecode for a program is defined to be a sequence of *nodes*, and the syntax of a node is defined in Figure 1. In the syntax definition, the meta-variables i and p range over integers and primitive function names respectively.

An n-ary application node (`Ap` i) in Reduceron bytecode is a pointer i to a sequence of n consecutive nodes in memory whose final node is wrapped in an `End` marker. To permit sharing in over-saturated applications, the nodes in an application sequence are stored in *reverse* order, e.g. $f\ x\ y$ would be stored as $y\ x$ (`End` f), and if $f\ x$ evaluates to z then the application can simply be updated to y (`End` z) without relocating it in memory. To illustrate, Figure 2 shows the bytecode of the `fact` function, as it would appear relative to some address a in program memory. Each application node in the bytecode is an *offset* address, relative to a, the address of the first node of the function's bytecode. This first node is always a `Start` node, and defines the arity and size (number of words) of the function's body. The bytecode for a whole program is simply the concatenation of the bytecodes for each individual function. Each `Fun` node is then adjusted to point to the final location of the function in the program.

node	::=	`Start` i i	(first node of a function body: arity and size of function)
	|	`Int` i	(primitive integer)
	|	`Ap` i	(application node: a pointer to a sequence of nodes)
	|	`End` *node*	(the final node in a node sequence)
	|	`Prim` p	(primitive function name)
	|	`Fun` i	(pointer to a function body)
	|	`Var` i	(variable representing a function argument)

Fig. 1. The syntax of nodes in Reduceron Bytecode

a	$+1$	$+2$	$+3$
`Start 1 15`	`Ap 7`	`Int 1`	`Ap 5`
$+4$	$+5$	$+6$	$+7$
`End (Int 1)`	`Prim (==)`	`End (Var 0)`	`Ap 12`
$+8$	$+9$	$+10$	$+11$
`Ap 10`	`End (Fun` a`)`	`Ap 14`	`End (Int 1)`
$+12$	$+13$	$+14$	$+15$
`Prim (*)`	`End (Var 0)`	`Prim (-)`	`End (Var 0)`

Fig. 2. The bytecode for `fact`, as it would appear relative to address a in memory

3 An Operational Semantics for the Reduceron

In this section, a semantics for the Reduceron is defined. The are two reasons for presenting a semantics: first to define precisely how the Reduceron works, and second to highlight the parts of the reduction process that can be assisted by *special-purpose hardware*. The semantics is given as a binary small-step state transition relation, $\Rightarrow$, between triples of the form $\langle h,\ s,\ a\rangle$, where h is the heap, s is the node stack, and a is the address stack.

In defining the semantics, we model the heap and stacks as *lists*, and assume the availability of several common functions on lists. In addition, we write $\#xs$ to denote the *length* of the list xs and $xs[i \mapsto x]$ to denote xs with its i^{th} element replaced by x.

Initially, the heap contains the bytecode of the program, the node stack contains the node `Fun 0`, where 0 is the address of the function `main :: Int`, and the address stack contains the address 0. The final result of a program p is defined to be r where

$$\langle p,\ [\texttt{Fun 0}],\ [0]\rangle\ \Rightarrow^{\star}\ \langle \_,\ [\texttt{Int}\ r],\ \_\rangle$$

Notice that the `main` function is a pure (non-monadic) function with no arguments. Currently, Reduceron programs take no input. Furthermore, their output is a single integer – `main` is of type `Int`.

3.1 A Primitive Evaluator

We assume a function $\mathcal{P}$ that takes a primitive function name p and two integers, i and j, and returns a *node* representing the value of $p\ i\ j$. For example,

$$\mathcal{P}\ \texttt{(+)}\ 5\ 10\ =\ \texttt{Int}\ 15$$

and

$$\mathcal{P}\ \texttt{(==)}\ 1\ 1\ =\ \texttt{Fun}\ \mathit{true}$$

where *true* is the address of the function `True` in the bytecode of the program.

3.2 Semantic Definition

The small-step transition relation $\Rightarrow$ is defined in Figure 3 and the helper functions *inst* and *unwind* are defined in Figure 4. There is one transition rule for each possible type of node that can appear on top of the stack, as described by the following paragraphs.

$$\langle h,\ \texttt{Int}\ i : x : s,\ a\rangle \ \Rightarrow\ \langle h,\ x : \texttt{Int}\ i : s,\ a\rangle$$

$$\langle h,\ \texttt{Prim}\ p : \texttt{Int}\ x : \texttt{Int}\ y : s,\ \_ : \_ : r : a\rangle \ \Rightarrow\ \langle h[r \mapsto \texttt{End}\ z],\ z : s,\ r : a\rangle$$

where

$$z\ =\ \mathcal{P}\ p\ x\ y$$

$$\langle h,\ \texttt{Ap}\ i : s,\ \_ : a\rangle \ \Rightarrow\ \mathit{unwind}\ i\ \langle h,\ s,\ a\rangle$$

$$\langle h,\ \texttt{Fun}\ i : s,\ a\rangle \ \Rightarrow\ \mathit{unwind}\ \#h\ \langle h',\ s',\ a'\rangle$$

where

$$\begin{aligned}
\texttt{Start}\ \mathit{arity}\ \mathit{size} &= h\ !!\ i\\
\mathit{body} &= \mathit{take}\ \mathit{size}\ (\mathit{drop}\ (i+1)\ h)\\
s' &= \mathit{drop}\ \mathit{arity}\ s\\
r : a' &= \mathit{drop}\ \mathit{arity}\ a\\
h' &= h[r \mapsto \texttt{End}\ (\texttt{Ap}\ \#h)]\\
&\qquad +\!\!+\ \mathit{map}\ (\mathit{inst}\ s\ \#h)\ \mathit{body}
\end{aligned}$$

Fig. 3. Transition rules for the Reduceron

Primitives. Recall from section 2.3 that primitive applications of the form $p\ a\ b$, where a and b are unevaluated integers, are transformed to $b\ (a\ p)$. Clearly, to evaluate such an application, b must be evaluated first. This results in the *value* of b, of the form `Int` i, appearing on top of the stack. To deal with such a situation, the evaluator simply *swaps* the top two stack elements, resulting in $(a\ p)\ b$ on the stack. Further evaluation yields $a\ p\ b$ on top of the stack and then, after another swap, $p\ a\ b$, where a and b are now fully evaluated, and evaluation of the primitive application is straightforward.

Once the result of the primitive application has been computed, it must be written onto the heap, overwriting the the contents of the original application node $b\ (a\ p)$, so that other references to it do not repeat the computation. This is possible because, as will be explained shortly, a pointer to the original application is sitting on the *address stack*.

Applications. When an application node of the form `Ap` i appears on top of the stack, it is replaced by the `End`-terminated sequence of nodes starting at address i on the heap. Furthermore, the addresses of the nodes in the sequence are pushed on the address stack, to permit updating the sequence after reduction. Following Peyton Jones's terminology, we collectively call these two tasks *unwinding*.

In an implementation of the Reduceron on a standard PC architecture, each node in an application sequence is read, one at a time, from the heap and written, one at a time, to the stack. Furthermore, each node address is computed and written, again one at a time, onto the address stack.

The definition of the *unwind* function in Figure 4 highlights the first main opportunities for hardware-assisted graph reduction. First, the uses of *getAp* and $+\!\!+$ illustrate that the nodes being copied are *contiguous*, so the copying can be achieved by block transfers in a machine with a wider data bus. Second, the

$$
\begin{array}{rcl}
inst\ s\ b\ (\texttt{Var}\ i) & = & s\ !!\ i \\
inst\ s\ b\ (\texttt{Ap}\ i) & = & \texttt{Ap}\ (b+i-1) \\
inst\ s\ b\ (\texttt{End}\ n) & = & \texttt{End}\ (inst\ s\ b\ n) \\
inst\ s\ b\ n & = & n \\
\\
unwind\ i\ \langle h,\ s,\ a\rangle & = & \langle h,\ reverse\ ap +\!\!+ s,\ reverse\ as +\!\!+ a\rangle \\
\textbf{where} & & \\
ap & = & getAp\ (drop\ i\ h) \\
as & = & map\ (i+)\ [0 \ldots \#ap - 1] \\
\\
getAp\ (\texttt{End}\ n : ns) & = & [n] \\
getAp\ (n : ns) & = & n : getAp\ ns
\end{array}
$$

Fig. 4. Definitions of *inst* and *unwind*

use of *map* to compute the node addresses indicates that they can be computed in parallel. And third, there is no dependency between writing to the node and address stacks, so the two can be done at the same time in a machine with parallel memories.

Functions. When a node of the form `Fun i` is at the top of the stack, the bytecode starting at address $i + 1$ on the heap is

1. *copied* onto the end of the heap (say at address *hp*),
2. with each variable `Var` j *substituted* with the j^{th} argument on the stack,
3. and with each application node, `Ap` k, *relocated* to an absolute address, `Ap` $(hp + k - 1)$, on the heap.

Subsequently, n nodes are popped off the node and address stacks, where n is the arity of the function that has just been instantiated. The address r which is n places from the top of the address stack represents the *root* of the redex. The value at r is overwritten with `End` (`Ap` *hp*), so that the reduction is never repeated. Finally, the node sequence beginning at the address *hp* is unwound onto the stack. We refer to this whole collection of operations as *function unfolding*.

Just as for unwinding, function unfolding on a standard PC architecture requires execution of many sequential instructions to carry out all the necessary memory manipulations. And again the semantics shows great scope for parallelism. In particular, the use of ++ to copy a potentially large contiguous block of nodes onto the end of heap, and the use of *map* to instantiate each node independently, opens up the possibility for parallelisation on a machine with wide memory and vectorised processing logic. Since instantiation of a node requires access to the stack, a parallel evaluator would need to be able to read the stack and heap at the same time. Further, because the nodes are being copied from one portion of memory to another, sequentialisation can be reduced by separating program memory and application node memory, permitting parallel access.

Notice in the semantics that the `Fun` rule calls *unwind*. Immediately after a combinator body is instantiated on the heap, the *spine* of that body is unwound from the heap onto the stack. It is much more efficient to instantiate the spine of the combinator on the heap and the stack *in parallel*. This idea is related to the *spineless* G-machine [2], which, by keeping track of which nodes are not *shared*, can often completely bypass construction of the combinator spine on the heap. So our idea gives the speed benefit of the spineless G-machine without introducing any complexity, but not the space benefit.

4 Implementation on FPGA

The semantics presented in the previous section suggests that an efficient implementation of the Reduceron can be obtained if *wide, parallel memories* and *vectorised processing logic* are available. A suitable architecture on which to explore this possibility is the FPGA. FPGA devices are ideal for constructing custom processing logic and typically contain large arrays of independent block

RAMs. This section describes our implementation of Reduceron on the FPGA device available to us, a Xilinx Virtex-II.

To measure of the effect of our proposed optimisations, we implement two versions of the Reduceron on the Virtex-II: *Baseline* and *Wide*. The Wide version exploits wide, parallel memories and the Baseline version does not.

4.1 Block RAMs

The Virtex-II contains 56 independent 1024 by 18-bit dual-port block RAMs. Being "dual-port" means that a RAM has two address busses, two data busses and two write enable signals. Thus two different locations in RAM can be accessed in a single clock cycle. Furthermore, each port has separate busses for data input and data output. Thus a value may be written to and read from a single location on a single RAM port at the same time. The *Wide* Reduceron exploits both the dual-port and separate data bus features of block RAMs, and the *Baseline* version does not. Both versions of the Reduceron encode bytecode nodes as 18 bit words, so each RAM location has capacity for a single node.

4.2 Constructing Large Memories from Small Ones

The Baseline Reduceron *cascades* 48 block RAMs to form a single 48k word memory; 32k is used as heap and stack memory and 16k is used solely by the garbage collector (described in section 4.5). Block RAMs are cascaded in the standard way using a multiplexor to combine the outputs of several memories into a single output. When cascading a large number of block RAMs the multiplexor becomes rather large and its delay becomes significant. To overcome this inefficiency, a register is placed on the output of the multiplexor. This means that two clock cycles are needed between writing an address to the address-bus and reading the resulting value off the data-bus. This overhead is alleviated by *pipelining*, whereby a new memory access is scheduled while waiting for the previous one to complete. But keeping the pipeline primed at all times is difficult, so some overhead is inevitable. Such overhead is present in both versions of Reduceron, as we always buffer RAM outputs in a register.

4.3 Quad-Word Memory

To permit wider memory transfers, the Wide Reduceron uses *quad-word* memories allowing any four consecutive locations to be read or written in a single clock cycle. This is *not* the same as saying that memory locations store 72 bits rather than 18 – that would imply that only blocks of words beginning at a four word boundary could be accessed in one cycle. A 72 bit wide memory is easier to build on the Virtex-II, but a quad-word memory facilitates implementation of graph reduction since word alignment issues can be ignored. As the method to implement quad-word memories on FPGA is neither standard nor obvious, and they cannot be synthesised automatically by existing FPGA design tools, we give details. Quad-word memories are built out of four separate memories. If each

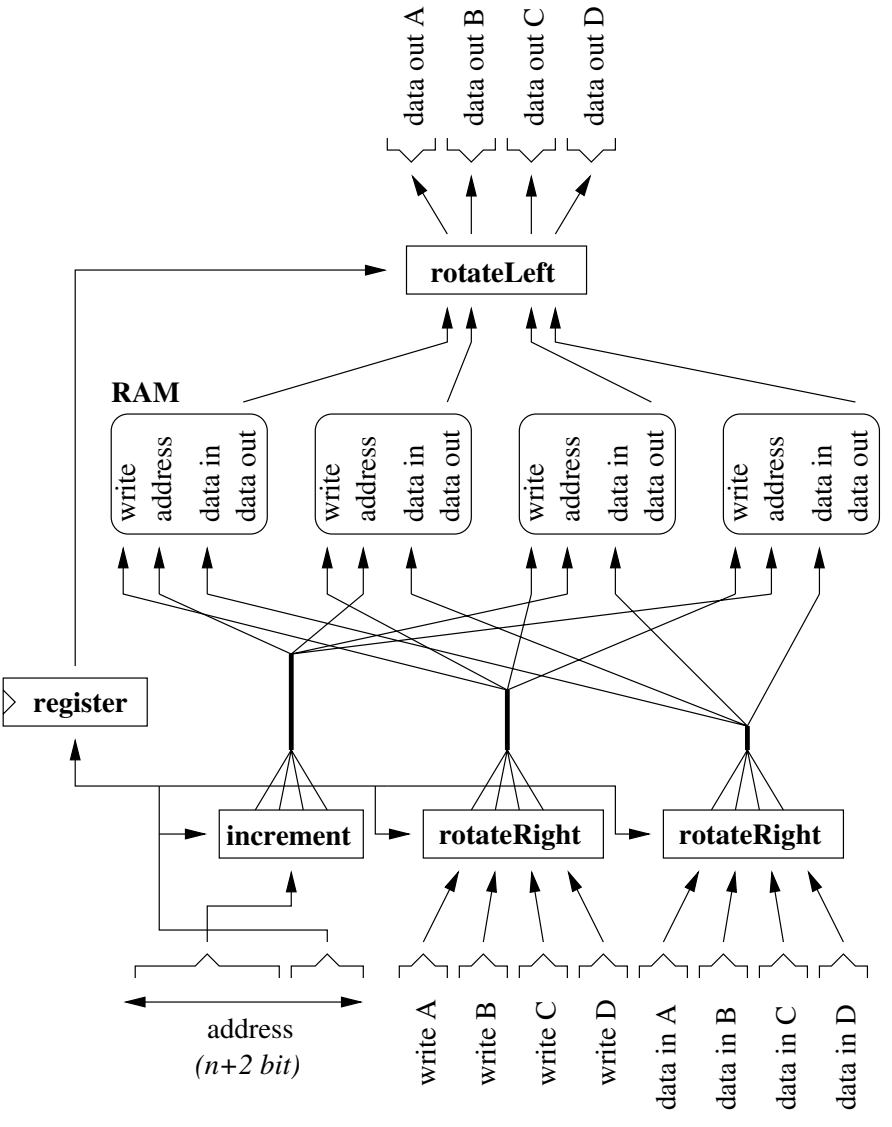

Fig. 5. Circuit diagram for a quad-word memory

internal memory is numbered i where i is drawn from $[0, 1, 2, 3]$ then memory i is used to store locations $[i, i+4, i+8, \ldots]$ of the quad-word memory. Accessing four consecutive locations beginning at an address a is then straightforward if a is a multiple of four, but awkward if it is not. Awkward, but not impossible, because each of the four consecutive locations, beginning at *any* address, *must* be stored in a different internal memory. The problem is then one of rotating

Table 1. The parallel, dual-port, quad-word memories of the Wide Reduceron

Memory	**Capacity** (words)
Combinator (program) store	4k
Heap	32k
Node stack	4k
Address stack	4k
Garbage collector scratch-pad	12k

the quad-word input and output data busses so they line up with those of the internal memories. See Figure 5. The *rotateLeft* and *rotateRight* circuits rotate a given list of inputs by a given number of positions. The *increment* circuit takes an address a and a number n, and produces 4 copies of a, the first n of which are incremented by 1.

Finally, the Wide Reduceron uses quad-word memories that are also *dual-port*, so up to eight words to be accessed together.

4.4 Parallel Memories

As well as widening memory, our semantics also suggested that parallel memories are beneficial, allowing, for example, the stack, heap and combinators to be accessed at the same time during function unfolding. For this reason, the Wide Reduceron has five separate memories, each of which is shown in Table 1 alongside its capacity.

4.5 Garbage Collection

For any serious computations to be performed in such a small amount of memory, a garbage collector is essential. Both versions of the Reduceron use a simple stop-and-copy two-space garbage collector [5]. In this algorithm, active nodes in the heap are copied onto an empty scratch-pad. The scratch-pad, which then contains a compacted copy of the heap, is copied back to the heap again before reduction continues. Although not the cleverest collector, it has the advantage of being extremely simple. Furthermore, the algorithm is easily defined to be *iterative* so no recursive call stack is needed. Our focus is on optimising the reduction process rather than exploring advanced garbage collectors.

4.6 Clock-Level Timing Breakdown

Table 2 shows the number of clock cycles taken to execute each transition rule of the Reduceron. The variable n represents the number of nodes in the body of the supercombinator being unfolded. Each transition rule requires at least two clock cycles because memory is buffered to shorten the critical path (see section 4.2).

Unwinding an application always takes two clock cycles as applications are limited to be a maximum of eight nodes long. The compiler hides this limitation

Table 2. Clock cycles taken by each Reduceron instruction

Operation	Clock cycles
Swap	2
Primitive	3
Unwind	2
Unfold	$3 + \lfloor \frac{n}{8} \rfloor$

by splitting large applications into smaller, nested ones – e.g. $f\ a\ b\ c\ d$ is equivalent to $(f\ a\ b)\ c\ d$.

Another restriction of the implementation is that functions have a maximum of eight parameters. This is because only eight elements of the stack can be accessed simultaneously while instantiating a function body. Again, the limitation can be hidden by the compiler, though our current implementation does not yet do so.

4.7 Description Language

Both versions of the Reduceron are implemented in Haskell using the Lava library [4]. Lava allows circuits to be described by normal Haskell functions over structures of bits (booleans), and can turn such functions into VHDL netlists of FPGA components that can be synthesised by the Xilinx tool set.

We view the *description* of the Reduceron circuit as an interesting aspect of our work. Lava's functional approach has been found to be suitable, despite the Reduceron being an irregular, stateful circuit, non-typical of many Lava applications found in the literature. In particular, we were surprised by how much of the Reduceron could actually be described by *pure* (non-monadic) Lava functions. For example, pure functions are all that are needed to describe the circuit in Figure 5 and the *inst* function in Figure 4. To express the stateful aspects of the circuit we developed a register-transfer monad, similar to the Recipe monad we define in [11]. Having pure functions as the *default* description method is quite appealing: pure functions are typically easy to test and verify; they result in concise, highly parameterised descriptions; and they naturally express circuit parallelism. Only when one needs to express intricate control flow and timing is monadic (sequential) code required.

Unfortunately, although the Reduceron descriptions are quite short, space does not permit presenting them in this paper.

4.8 Resource Usage

The results of synthesising each version of the Reduceron for the Virtex-II (`XC2V2000-6BF957`) using Xilinx ISE 9.1 are shown in Table 3. Concerning clock frequency: a small, carefully optimised 8-bit processor designed by Xilinx (the PicoBlaze) can be clocked at 173.6 MHz on the same device. For the Reduceron to be clocking within a factor of two is acceptable, but suggests room for improvement. One problem with our tool flow is that there is no traceability from

Table 3. Reduceron synthesis results on the Virtex-II (`XC2V2000-6BF957`)

	Baseline	**Wide**	**Maximum**
Number of slices	1530	4874	10752
Number of block RAMs	48	56	56
Clock frequency	94.7 MHz	91.5 MHz	(see text)

code written in Lava to the generated netlist, so it is hard to identify the critical path in the Lava program.

5 Performance

In this section, the impact of the wide memory optimisations is measured by comparing the Baseline and Wide Reducerons running a range of Haskell programs. In addition, the *potential* for special-purpose graph reduction machines is explored by running the same programs using several Haskell implementations on a Pentium-4 2.8GHz PC. The PC Haskell implementations are: Hugs (version May 2006), GHCi (version 6.6), Yhc (latest), Nhc98 (version 1.20), a C implementation of the Reduceron, and the GHC native code compiler (version 6.6) with and without optimisations.

5.1 Programs

Due to the restrictions on the Reduceron, the Haskell programs used in our experiments must: (1) have a maximum heap residency and stack size less than 32k words and 4k words respectively; (2) not take any external input; and (3) produce a single integer as a result. The programs used are:

1. **OrdList.** A program to check the property that insertion into a list preserves ordering for all boolean lists of depth n, applied to $n = 11$.
2. **Perm.** A program to find the smallest number in a list of numbers using a permutation sort, applied to the list containing the numbers 9 down to 1.
3. **MSS.** A program to compute the maximum segment sum of a list of integers applied to the list `[-150..150]`.
4. **Queens.** A function to compute the number of queens that can be placed on an n-by-n chess board such that no queen attacks any other queen, applied to $n = 10$.
5. **Adjoxo.** An adjudicator for noughts and crosses that determines if one side can force victory given a partially complete board. The adjudicator is applied to the empty board.
6. **SumPuz.** A solver for general cryptarithmetic problems. It is applied to a range of problems and outputs the total number of solutions to all of them. (Integer division is not supported on the Reduceron so is implemented by repeated subtraction.)
7. **Sem.** A structural operational semantics of the *While* language [12] applied to a program that naively computes the number of divisors of 1000. (Divisor testing is implemented by repeated subtraction.)

Table 4. Timings of a range programs running on various Haskell implementations

	OrdList	Perm	MSS	Queens	Adjoxo	SumPuz	Sem
Hugs	3.68s	2.70s	3.85s	6.50s	14.81s	4.11s	5.49s
Baseline Red.	13.19s	4.15s	7.41s	7.65s	15.92s	8.98s	15.87s
GHCi	4.26s	2.42s	3.24s	6.35s	7.09s	3.39s	5.36s
Yhc	3.59s	1.76s	1.22s	3.06s	3.85s	2.51s	3.81s
PC Red.	3.65s	1.16s	1.96s	2.33s	5.00s	2.77s	4.50s
Nhc98	3.60s	1.46s	1.38s	2.32s	3.12s	2.28s	3.21s
Wide Red.	1.88s	0.58s	2.01s	1.57s	2.70s	1.67s	2.04s
GHC	0.71s	0.28s	0.38s	0.66s	0.86s	0.47s	0.41s
GHC -O2	0.57s	0.19s	0.28s	0.09s	0.30s	0.27s	0.34s

Table 5. Profiles of programs running on the Wide Reduceron

	OrdList	Perm	MSS	Queens	Adjoxo	SumPuz	Sem
Unwind	31.4%	33.0%	41.7%	32.1%	37.7%	36.8%	28.9%
Unfold	64.0%	54.7%	24.1%	27.8%	37.8%	37.2%	55.8%
Swap	0.0%	4.7%	5.1%	10.8%	7.8%	6.0%	5.8%
Prim.	0.0%	3.5%	7.6%	12.2%	6.8%	5.2%	4.6%
GC	4.6%	4.1%	21.5%	17.0%	9.9%	14.8%	4.9%

5.2 Observations

See tables 4 and 5 for run times and instruction profiles. On average, the Wide Reduceron outperforms the Baseline Reduceron by a factor of six. On heavily arithmetic programs (Queens and MSS) the factor is between three and five, whereas on heavily applicative programs (OrdList and Sem) it is between seven and eight. Unfolding benefits most from wider memory. The average factor of six improvement is significant, but we might have hoped for more considering that eight consecutive locations can be accessed together on each of the five parallel memories. Some suggestions to utilise the parallel memory more fully are given in section 5.4.

On average, the Wide Reduceron (on FPGA) outperforms the Reduceron, Yhc, and Nhc98 bytecode interpreters (on PC). All of these implementations share a common frontend, so each interpreter runs the same core Haskell programs. One of the potential advantages of a bytecode interpreter is that the bytecode can be made sufficiently abstract to have a concise formal semantics, offering hope for a mechanically verified Haskell implementation. However, there is a tension between defining a simple, high-level bytecode and one that is similar enough to the target machine so as to be *efficient*. The Reduceron approach appears to relax this tension; a simple bytecode can be designed without concern for the target machine, and then a machine can be designed to efficiently execute this bytecode. Interestingly, the PC version of the Reduceron performs surprisingly well in comparison to Yhc and Nhc98, considering that it is based

on template instantiation and that Yhc and Nhc98 are G-machine variants. This reinforces the findings of Jansen [9].

The leading native-code compiler GHC performs many advanced optimisations. For example, GHC spots that the critical `safe` function in Queens is strict, so need not be instantiated on the heap. Similar optimisations might be used in a future Reduceron implementation, but architectural changes would be required, e.g. moving from a template instantiation evaluator to an instruction sequence approach. Excluding Queens and Adjoxo, which both involve significant integer operations in a critical loop, and which GHC's optimisations speed up by over a factor of two, the Reduceron (on FPGA) runs, on average, 4.85 times slower than GHC -O2 (on PC).

5.3 Increasing Memory Capacity and Clock Frequency

One of the main limitations of the Reduceron is that it only has 32k words of heap space. This is enough to make an interesting experiment, but too small for any serious application. However, the limitation might be overcome with improved hardware, *without* affecting the existing design significantly. For example, the Computer Architecture group at York have built the PRESENCE-3 FPGA board [13] containing a Virtex-5 FPGA and five large, fast RAMs. Since these RAMs are all accessible in parallel, a wide heap could be obtained using off-chip storage. Further, the Virtex-5 would offer many more block RAMs, permitting larger stack and combinator memories on-chip and therefore to be accessed in parallel as in the existing design.

Another benefit of the Virtex-5 over the Virtex-II is higher performance. The Xilinx synthesis tool states that our current Reduceron design will run at 160 MHz on the Virtex-5. Identifying and reducing the critical path would yield further improvements.

5.4 Possible Design Improvements

Currently, the compiler does not attempt to modify the program to take advantage of the Wide Reduceron's features. In particular, lambda lifting after encoding data types as functions usually introduces a new function definition for each case alternative, breaking function bodies into smaller pieces. While this is desirable for the PC and Baseline Reducerons, larger function bodies should play to the strengths of the Wide Reduceron, and might justify direct support for case expressions and lambda abstractions.

Another limiting factor for memory utilisation is that application nodes are typically only one to five words in size. In particular, the indirections used to achieve sharing are only one node wide. Possible solutions include building combinator spines directly on top of redex roots, and the use of one-level deep trees instead of flat sequences for representing applications.

Eventually, multiple Reducerons could be put on a single FPGA to perform parallel evaluation [7]. The hope is that the flexibility of the FPGA would allow

for a simple yet effective means of parallel reduction. Another avenue of exploration would be to develop a variant of the `ByteString` library [3] which exploits wide memories and vectorised processing logic on the FPGA.

6 Related Work

In the FPCA series of international conferences held between 1981 and 1995, several papers presented designs of exotic new machines to execute functional programs efficiently. Some special-purpose, sequential graph reduction machines were indeed built, including SKIM [16] and NORMA [15]. Unfortunately, at the time, building such machines was a slow and expensive process, and any performance benefit obtained was nullified by the next advancement in stock hardware. Nowadays, the situation is different: FPGA technology has significantly reduced the time and expense required to build custom hardware, and is a widespread, advancing technology in its own right. Furthermore, it appears that users of stock hardware can no longer expect automatic advances in sequential computing speed. Another difference compared with the Reduceron is that both SKIM and NORMA were based on Turner's combinators and did not attempt to use wide, parallel memories to increase performance.

A piece of work similar in spirit to the Reduceron is Augustsson's Big Word Machine (BWM) [1], although the two have been independently. The BWM is a graph reduction machine with a wide word size, specifically four pointers long, allowing wide applications to be quickly built on, and fetched from, the heap. 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 encodes constructors and case expressions using functions and applications respectively. Unlike the Reduceron, the BWM works on an explicit, sequential instruction stream rather than by template instantiation, and it avoids updating the heap in some cases where a computation cannot be shared, thus saving unnecessary heap accesses. Features of the Reduceron not present in the BWM include (1) separate code and heap memories; (2) machine integer support; (3) less memory wastage as data need not be aligned on four-pointer boundaries; and (4) support for building multiple different function applications on the heap simultaneously. 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" [1].

7 Conclusion

In the introduction we argued that the von Neumann bottleneck impedes the performance of graph reduction on standard computers, and suggested that the problem could be overcome by building a special-purpose machine with wide, parallel memory units. We have explored this very possibility by building a prototype of such a machine – the Reduceron – using an FPGA. The combination of wide, parallel memory units and vectorised processing logic on the Reduceron

gives a factor of six speed-up on average across a range of benchmark programs. Furthermore, running at 91.5MHz on a Xilinx Virtex-II FPGA, the Reduceron performs better than interpreted bytecode and often within a small factor of optimised native-code running on a 2.8GHz Pentium-4 PC. Considering the large performance advantage of conventional *hard* processors over *soft*, FPGA-based ones for executing C programs [10], and the *simplicity* of the Reduceron, it would certainly be an interesting result if, after further work on the Reduceron, Haskell programs were found to run at comparable speeds on both.

FPGAs have, to a large extent, eliminated the effort and expertise needed to build custom hardware. They may be viewed as an advancing technology that continues to offer higher performance, perhaps one day approaching the clock rates of modern PCs. Or, alternatively, as a tool for rapidly prototyping designs before they are manufactured as efficient, non-programmable ASICs. Both views, along with the results obtained in this paper, motivate further experiments in the design of special-purpose graph reduction machines using FPGAs. The hope is that researchers can find *simple* and *elegant* yet *fast* and *parallel* reduction methods by side-stepping the constraints and intricacies of standard, von Neumann, computers.

Acknowledgements

The first author is supported by an award from the Engineering and Physical Sciences Research Council of the United Kingdom. We thank Jack Whitham and Ian Gray for their help in the remote lab and the Real Time Systems group for providing the Virtex-II FPGA. We also thank Emil Axelsson for comments on a draft, and the anonymous IFL reviewers for both encouraging and critical comments! Most of all, we thank Neil Mitchell for many useful discussions, for making Yhc Core such a pleasure to use, and for providing several transformations including the data type encoder and the lambda lifter.

References

1. Augustsson, L.: BWM: A Concrete Machine for Graph Reduction. In: Proceedings of the 1991 Glasgow Workshop on Functional Programming, London, UK, pp. 36–50. Springer, Heidelberg (1992)
2. Burn, G.L., Peyton Jones, S.L., Robson, J.D.: The spineless G-machine. In: LFP 1988: Proceedings of the 1988 ACM conference on LISP and functional programming, pp. 244–258. ACM, New York (1988)
3. Coutts, D., Stewart, D., Leshchinskiy, R.: Rewriting haskell strings. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 50–64. Springer, Heidelberg (2006)
4. Claessen, K.: Embedded Languages for Describing and Verifying Hardware. PhD Thesis, Chalmers University of Technology (2001)
5. Fenichel, R.R., Yochelson, J.C.: A LISP garbage-collector for virtual-memory computer systems. Commun. ACM 12(11), 611–612 (1969)
6. Golubovsky, D., Mitchell, N., Naylor, M.: Yhc.Core - from Haskell to Core. The Monad.Reader (7), 45–61 (2007)

7. Hammond, K., Michelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer, London (2000)
8. Hughes, R.J.M.: Super Combinators—A New Implementation Method for Applicative Languages. In: Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, pp. 1–10 (1982)
9. Jansen, J.M., Koopman, P., Plasmeijer, R.: Efficient interpretation by transforming data types and patterns to functions. In: Trends in Functional Programming, vol. 7, Intellect (2007)
10. Lysecky, R., Vahid, F.: A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In: DATE 2005: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, pp. 18–23. IEEE Computer Society, Los Alamitos (2005)
11. Naylor, M.: A Recipe for Controlling Lego using Lava. The Monad.Reader (7), 5–21 (2007)
12. Riis Nielson, H., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley, Chichester (1992)
13. University of York. PRESENCE-3 Development, `http://www.cs.york.ac.uk/arch/neural/hardware/presence-3/`
14. Peyton Jones, S.: The Implementation of Functional Programming Languages. Computer Science. Prentice-Hall, Englewood Cliffs (1987)
15. Scheevel, M.: NORMA: a graph reduction processor. In: LFP 1986: Proceedings of the 1986 ACM conference on LISP and functional programming, pp. 212–219. ACM, New York (1986)
16. Stoye, W.: The Implementation of Functional Languages using Custom Hardware. PhD Thesis, University of Cambridge (1985)
17. The Yhc Team. The York Haskell Compiler - user's guide (February 2007), `http://www.haskell.org/haskellwiki/Yhc`