Supercompilation and the Reduceron

Jason S. Reich, Matthew Naylor and Colin Runciman

Department of Computer Science, University of York
{jason,mfn,colin}@cs.york.ac.uk

Abstract. This paper explores some of the performance-enhancing features of supercompilation in the context of the Reduceron — a special-purpose graph-reduction machine. Two small examples are discussed in detail, highlighting areas where the two technologies interact. A strategy is introduced for countering a situation where supercompilation adversely affects Reduceron execution time. Performance results and other metrics are presented across a range of nineteen benchmarks highlighting the synergistic properties of supercompilation on the Reduceron. This paper represents work in progress.

1 Introduction

Functional programming is a distinctive paradigm that has scope for exploiting non-standard technologies at every stage of computation. Supercompilation and the Reduceron are two such technologies.

Supercompilation [1,2] is a metaprogramming technique that, at compile-time, evaluates (drives) programs until an unknown is required and then proceeds by case analysis (residuates). Among other benefits, it can remove intermediate data structures and specialise higher order functions, with corresponding performance gains at execution time.

The Reduceron is an FPGA-based soft processor for executing lazy functional programs by graph reduction [3,4]. The special-purpose processor can perform in parallel many of the steps required for each reduction, whereas conventional architectures need to perform these steps serially.

Does a combination of these technologies lead to further improvements in performance? Are these techniques conflicting, compatible or even mutually beneficial? In this paper, we discuss how the two may interact and present preliminary findings from a prototype supercompiler for the Reduceron source language.

2 Our Source Language

Our source language [5] is close to subsets of both Haskell 98 [6] and Clean [7]. It supports algebraic data types, uniform pattern matching by construction, local variable bindings, and various primitive integer operations.

$$
\begin{aligned}
prog &:= \overline{f\ \overline{vs} = x} \\[1em]
exp &:= v\ (\mathit{variables}) \\
&\mid c\ (\mathit{constructors}) \\
&\mid f\ (\mathit{functions}) \\
&\mid n\ (\mathit{integers}) \\
&\mid x\ \overline{xs}\ (\mathit{applications}) \\
&\mid \textbf{case}\ x\ \textbf{of}\ \overline{c\ \overline{vs} \rightarrow x} \\
&\mid \textbf{let}\ \overline{v = x}\ \textbf{in}\ y
\end{aligned}
$$

Fig. 1. Abstract syntax for our source language.

Abstract syntax for our source language is given in Figure 1. In addition to the annotated symbols, x and y range over expressions. Overlining and pluralisation indicate sequences of productions. For example $\overline{vs}$ represents a sequence of variable names. All programs contain a function named `main` of arity zero.

Listing 1 shows an encoding of a program in our source language. This program (somewhat inefficiently) doubles each element in a range, calculates the sum and prints the result.

The basic compiler (before the introduction of supercompilation) first reduces all pattern matching to combinations of one-level case distinctions. A case-elimination phase then translates algebraic data constructors and case expressions to functions and function applications respectively, using a variation of the Scott encoding, e.g. after case elimination our `range` function is as displayed in Listing 2. In every function body, a bottom-up traversal inlines saturated applications of non-primitive functions. Finally, the compiler generates a compact encoding of function body.

The encoding phase must take into account the design parameters of the Reduceron. Encoded forms of function bodies are constrained by limits on the size of the top-level spine, the number and size of nested applications, and the number of case-table arguments in an application. Encoded bodies are split if necessary, with the introduction of auxiliary combinators.

It should be stressed that the compiler *is not* generating circuitry for the FPGA. Rather, it is generating a representation of the program, suitable for execution on a template instantiation machine [8]. In this case, the template instantiation machine, the Reduceron, has been realised on an FPGA.

Listing 1. An example program, in our source language.

```
{
    foldl f z xs = case xs of {
        Nil        -> z;
        Cons y ys -> foldl f (f z y) ys;
    };

    map f xs = case xs of {
        Nil        -> Nil;
        Cons y ys -> Cons (f y) (map f ys);
    };

    plus x y = (+) x y;
    sum = foldl plus 0;

    double x = (+) x x;

    sumDouble xs = sum (map double xs);

    range x y = case (<=) x y of {
        True  -> Cons x (range ((+) x 1) y);
        False -> Nil;
    };

    main = emitInt (sumDouble (range 0 10000)) 0;
}
```

3 The Reduceron Architecture

The Reduceron features broad memory channels to ‘widen the von Neumann bottleneck.’ [3] Many of the operations required to perform each graph reduction step are simultaneously performed in a single clock cycle.

Instantiation of a function body takes $\lceil n/2 \rceil$ clock ticks, where n is the number of applications in the body. Establishing the environment for function applications, updating the heap-graph to prevent repeated evaluations and applying primitive functions each take just one clock cycle. Dynamically maintained sharing information allows the Reduceron to avoid a high proportion of redundant updates where no sharing can occur. Constructor reductions (selection of the appropriate case alternative function from a case table) take place in zero clock cycles.

Listing 3 shows the evaluation of `range 0 10` to head normal form. Each reduction step is annotated with the operation that is being performed and how many cycles are required. This example takes four clock cycles, under the scheme outlined so far.

Listing 2. range function after case elimination.

```
range x y = (<=) x y [range#1,range#2] x y;
range#1 alts x y = Nil;
range#2 alts x y = Cons x (range ((+) x 1) y);
```

Listing 3. Reduction of range 0 10 without PRS.

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (<=) 0 10 [range#1,range#2] 0 10
= { Primitive application (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range ((+) 0 1) 10)
```

4 Primitive Redex Speculation

If primitive applications in body have fully evaluated arguments at instantiation time, the Reduceron can evaluate them speculatively during instantiation. 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.

Once again consider the reduction of the expression `range 0 10`, now with primitive redex speculation (PRS) enabled (Listing 4). One clock cycle is avoided for the comparison. Further clock cycles will be saved if the tail of the result is needed, as the addition to form the lower bound of the range has also been speculatively evaluated.

The beneficial effect of PRS is quite marked. For the example program in Listing 1, the Reduceron takes 230,029 clock cycles to execute *without* PRS. With PRS enabled the Reduceron only takes 150,024 clock cycles, a 35% reduction.

However, the number of PRS reductions in each instantiation is limited by a Reduceron design parameter. Currently, this limit is two.

5 Benchmark Programs

A selection of nineteen programs are used to test and benchmark the Reduceron platform. These programs range from small, toy examples that demonstrate specific effects, such as the `sum`– series, to significant computations like `Knuthbendix`.

Listing 4. Reduction of `range 0 10` with PRS.

```
  range 0 10
= { Instantiate function body (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range 1 10)
```

These programs are described below in terms of their purpose, characteristics and code size. Line counts are for sources including all required auxiliary functions.

Adjoxo An adjudicator for the game noughts and crosses, a.k.a. tic-tac-toe. The input is a game position, and the output is one of the three values — Win, Draw or Loss — indicating the outcome with best play for each of the players whose turn it might be. The method is the usual minimax recursive evaluation of completed game trees. *(106 lines)*

Braun A Braun tree is a balanced binary tree offering an efficient yet simple implementation of flexible arrays. The program tests the property that converting a list to a Braun tree and back again is equivalent to the identity function. *(51 lines)*

Cichelli Finds a perfect hash function for Haskell keywords [9]. It uses a backtracking search to find an assignment of natural-number values to each letter that starts or ends a keyword such that hash values for keywords, computed as start-value + end-value + length, are unique and occupy a small integer range without gaps. *(200 lines)*

Clausify Puts propositional formulae in clausal form using a multi-stage transformation of formula-trees [9]. Almost a purely symbolic application, with hardly any arithmetic. *(131 lines)*

Fib Computes the Nth number in the fibonacci sequence using a simple but naive doubly-recursive function definition. A purely arithmetic program involving no data structures at all. *(10 lines)*

Knuthbendix The Knuth-Bendix completion method tries to derive a convergent term-rewriting system for a given equational theory and symbol-weighting scheme. It is a typical symbolic computing application from computer algebra. The example input used in the program gives group-theoretic axioms from which ten rewriting rules are derived. *(533 lines)*

MSS Computes the maximum segment sum of a list of integers. Works by dividing the input list into all sub-lists, computing the sum of each, and returning the maximum. *(47 lines)*

Mate Solves chess end-game problems of the form "P to move and mate in N" [9]. The method is brute-force search in an explicit AND-OR game tree developing the given position to depth $2N-1$. Boards are represented by a square-piece assocation list for each player, where squares are coded as rank-file numeric pairs, so there is a fair amount of primitive arithmetic and comparison. *(393 lines)*

OrdList Checks the property that insertion of a number into an ordered list of numbers results in a list that is still ordered. Numbers are represented as Peano numerals, so this is a purely symbolic program. *(46 lines)*

Parts Computes a celebrated number-theoretic function, the number of partitions of n, where a partition is a bag of positive integers that sum to n. There is a sophisticated closed formula for this number, but the method here is to list and count partitions explicitly. *(54 lines)*

PermSort Enumerates the permutations of a list of numbers, and returns the first ordered permutation. *(39 lines)*

Queens Solves a programming problem made famous by Wirth: place N queens on an $N \times N$ chess board so that no two queens occupy a common rank, file or diagonal [9]. The solution involves backtracking, list processing and an inner recursive loop that tests the safety of each candidate position for a new queen by primitive arithmetic comparisons with the coded positions of queens already in place. *(47 lines)*

Queens2 A purely symbolic solution to the N-queens problem. Represents the board as a list of lists. Places a queen on one row at a time, maintaining a grid of threatened squares, and backtracks if a queen cannot be placed. *(62 lines)*

Sudoku A Sudoku solver due to Richard Bird [10]. Fills the blank cells on a Sudoku board with valid digits, pruning many possible choices that cannot possibly lead to a solution. *(209 lines)*

Taut A tautology checking program based on an example from Hutton's book. The method is a brute-force evaluation for all possible boolean assignments to variables. *(95 lines)*

While A structural operational semantics of Nielson and Nielson's While language [11] applied to a program that computes the number of divisors of given integer. *(96 lines)*

sumDouble Computes $\sum_{i=0}^{10000} 2i$ by generating the list of the numbers between 0 and 10,000, doubling each element and then computing the sum using the higher-order function, `foldl`. Contains intermediate data structures and primitive operations. The program in Listing 1. *(20 lines)*

sumSquares Computes $\sum_{i=0}^{100} i^2$. This is done in a similar fashion to the previous example. The square function consists of replicating its input n, n times and summing the result using `foldl`. Contains intermediate data structures, primitive operations and nested loops. *(23 lines)*

sumSumEnum Computes $\sum_{i=0}^{100} \sum_{j=0}^{i} j$ by generating the list of the numbers between 0 and 100, mapping a function, `sumEnum`, over the list and summing the resulting list. The `sumEnum` function sums the numbers between 0 and its input. Contains intermediate data structures, primitive operations and nested loops. *(22 lines)*

Listing 5. A supercompiled form of `sumDouble`.

```
sumDouble = sumDoubleAc 0;
sumDoubleAc z xs = xs [sumDoubleAc#1, sumDoubleAc#2] z;
sumDoubleAc#1 y ys alts z = sumDoubleAc ((+) z ((+) y y)) ys;
sumDoubleAc#2 alts z = z;
```

6 A Synergistic Effect of Supercompilation

The basic Reduceron compiler currently performs very little optimisation. We are developing a supercompiler targeted at the Reduceron platform. The starting point for our current prototype was a previous positive supercompiler for a core functional language by Mitchell [12].

Mitchell's design inserted a supercompilation phase between core generation and compilation by the optimising Glasgow Haskell Compiler (GHC). In all but one of the published benchmarks, Mitchell's supercompiler demonstrated at least equal and often significantly improved performance when compared with GHC alone.

One reason for the improvement is that supercompilation fuses away intermediate data structures. In its original form, the function `sumDouble` (Listing 1) maps `double` over its list input, only to apply `foldl plus` to the newly constructed list to calculate the sum. The supercompiler fuses this composition to a residual function that does not produce the intermediate list but performs the `double` operation as it sums. For both conventional implementations and the Reduceron, fewer reductions are needed to construct and deconstruct data structures.

Some effects of supercompilation particularly benefit the features of the Reduceron architecture. For example, the program in Listing 1 cannot, as it stands, benefit from PRS during the `sum` function because the primitive addition is not apparent in the body of the higher-order `foldl`. However, if supercompiled, `foldl plus` is specialised to a first-order equivalent. A considerable reduction in clock cycles is obtained because PRS now applies.

The original program evaluates 20,003 expressions by PRS, compared with the 29,995 for the supercompiled program. While it is possible in some cases, for this example, no more primitive reductions were performed overall than we performed originally. A further performance gain achieved on top of the fusion effects. Following supercompilation, the Reduceron takes 159,970 clock cycles to execute the program in Listing 1 without PRS and only 39,996 clock cycles with PRS. Compared with the original program executed without PRS, this is a 87% performance increase.

Listing 5 shows the combined effects of supercompilation fusion and specialisation on `sumDouble`. In `sumDoubleAc#1`, the dependency of the outer addition on the inner one means that PRS requires an extra clock cycle to evaluate the

Listing 6. Original and supercompiled piece safety in the n-queens problem.

```
and x y = case x of { True -> y; False -> False };

safe x d qs = case qs of {
  Nil      -> True;
  Cons q l ->
    and ((/=) x q) (
    and ((/=) x ((+) q d)) (
    and ((/=) x ((-) q d)) (
    safe x ((+) d 1) l)));
  };

safeSC x d qs = case qs of {
  Nil      -> True;
  Cons q l ->
    case (/=) x q of {
      True ->
         case (/=) x ((+) q d) of {
           True ->
             case (/=) x ((-) q d) of {
               True -> safeSC x ((+) d 1) l;
               False -> False
             };
           False -> False
        };
      False -> False
    }
  };
```

expression fully. However, overall cycles are still saved in comparison with the reduction of a separate function application.

7 A Potentially Obstructing Effect of Supercompilation

There are circumstances where the process of supercompilation might impede PRS. Consider Listing 6, an extract from the `Queens` example. The function `safe` computes whether it is 'safe' to place a queen in rank x, at a distance of d files away from the queens currently placed on the board. These queens are specified by their rank positions in the list qs.

Listing 7 shows the original and supercompiled definitions following case elimination and inlining. Notice that in `safe`, all of the primitive reducible expressions are in one case alternative. On the other hand, in the supercompiled version, `safeSC`, the expressions are split over separate case alternatives, and therefore, instantiations after Scott encoding.

Listing 7. Listing 6 after case elimination.

```
and v0 v1 = v0 [and#1,and#2] v1;
and#1 v0 v1 = False;
and#2 v0 v1 = v1;

safe v0 v1 v2 = v2 [safe#1,safe#2] v0 v1;
safe#1 v0 v1 v2 v3 v4 = let {
    v5 = (+) v4 1;
    v6 = (/=) v3 ((-) v0 v4);
    v7 = (/=) v3 ((+) v0 v4);
    v8 = (/=) v3 v0
  } in v8 [and#1,and#2]
        (v7 [and#1,and#2]
          (v6 [and#1,and#2]
            (v1 [safe#1,safe#2] v3 v5)));
safe#2 v0 v1 v2 = True;

safeSC v0 v1 v2 = v2 [safeSC#7,safeSC#8] v0 v1;
safeSC#1 v0 v1 v2 v3 = False;
safeSC#2 v0 v1 v2 v3
  = let { v4 = (+) v2 1 } in v3 [safeSC#7,safeSC#8] v1 v4;
safeSC#3 v0 v1 v2 v3 v4 = False;
safeSC#4 v0 v1 v2 v3 v4
  = (/=) v1 ((-) v2 v3) [safeSC#1,safeSC#2] v1 v3 v4;
safeSC#5 v0 v1 v2 v3 v4 = False;
safeSC#6 v0 v1 v2 v3 v4
  = (/=) v1 ((+) v2 v3) [safeSC#3,safeSC#4] v1 v2 v3 v4;
safeSC#7 v0 v1 v2 v3 v4
  = (/=) v3 v0 [safeSC#5,safeSC#6] v3 v0 v4 v1;
safeSC#8 v0 v1 v2 = True;
```

This leads to a situation where the execution of the original can speculatively evaluate a number of expressions simultaneously, whereas `safeSC` evaluates them separately at each function body instantiation.

To alleviate this issue, primitive expressions can be lifted as far as their variables are bound. The lifting process can take into account the maximum number of PRS reductions at instantiation and only lift to where there is spare capacity.

However, if we naively lift all primitive redex expressions, we may cause duplicate computation to occur. The supercompiler is permitted to duplicate *code* as long as it does not duplicate *computation*, under lazy evaluation. For example, our supercompiler may replicate bindings from outside a case expression down each case alternative. As only one alternative is evaluated, only one of the duplicate bindings will be evaluated under both lazy and speculative evaluation.

	Original		Supercompiled		SC + PRS lift	
	a (No PRS)	*b* (PRS)	*c* (No PRS)	*d* (PRS)	*e* (No PRS)	*f* (PRS)
`Adjoxo`	1.000	0.799	0.866	0.671	0.707	**0.405**
`Braun`	1.000	1.000	**0.769**	**0.769**	**0.769**	**0.769**
`Chichelli`	1.000	**0.990**	1.000	**0.990**	1.013	0.998
`Clausify`	1.000	1.000	1.050	1.050	1.051	**0.951**
`Fib`	1.000	0.445	1.000	0.445	0.907	**0.353**
`KnuthBendix`	1.000	0.900	0.896	0.833	0.876	**0.779**
`MSS`	1.000	0.864	0.995	**0.858**	0.997	0.861
`Mate`	1.000	0.867	0.912	0.838	0.916	**0.827**
`OrdList`	1.000	1.000	**0.662**	**0.662**	0.678	0.678
`Parts`	1.000	0.746	0.933	**0.679**	1.029	0.753
`PermSort`	1.000	0.962	0.861	0.861	0.759	**0.727**
`Queens`	1.000	0.421	0.850	0.489	0.811	**0.325**
`Queens2`	1.000	0.996	0.989	0.985	0.966	**0.961**
`Sudoku`	1.000	0.936	0.955	0.892	0.922	**0.815**
`Taut`	1.000	1.004	**0.700**	**0.700**	0.944	0.859
`While`	1.000	0.947	0.996	**0.942**	1.047	1.005
`sumDouble`	1.000	0.652	0.695	0.174	0.739	**0.130**
`sumSquares`	1.000	0.541	0.726	0.206	0.793	**0.205**
`sumSumEnum`	1.000	0.481	0.847	0.455	0.960	**0.454**
Geometric Mean	*1.000*	*0.788*	*0.871*	*0.647*	*0.881*	***0.598***

Table 1. Execution time as multiples of that for pipeline *a*, the program before supercompilation executed without PRS. *(Best results are in bold.)*

However, if replicated primitive redexes are lifted above a case distinction, they may be evaluated speculatively, taking away capacity that other primitive expressions could have used. A solution is to detect these replicated expressions and merge them into a single binding.

Our original supercompiler worsened PRS-enabled results for `Queens` by 16%. With the primitive redex lifting strategy applied, supercompilation improves results by 33%.

8 Performance Results

8.1 Compared to without PRS and Supercompilation

Each example described in Section 5, is compiled with six variations of compilation pipeline. These are; *a*) normal compilation, *b*) normal compilation with PRS, *c*) supercompilation and normal compilation, *d*) supercompilation and normal compilation with PRS, *e*) supercompilation, primitive redex lift and normal compilation, *f*) supercompilation, primitive redex lift and normal compilation with PRS.

The compiled output is executed on a Reduceron simulator. The simulator returns various profiling measurements such as total clock cycles, the number of PRS evaluated expressions and the proportion of time spent on individual functions and reduction operations. Table 1 presents the performance of our test programs relative to that of pipeline *a*, the original program executed without PRS.

PRS Only — Primitive redex speculation (pipeline *b*) shows an average performance increase of 21%. All but four of the examples achieve some improvement. The only example that suffers under PRS is `Taut`, likely due to the structure of the program. The drop in performance is only very slight, however.

Supercompilation Only — Supercompilation (pipeline *c*) shows an average performance boost of 13%. Only three examples do not benefit from supercompilation. While we do not expect much improvement on `Fib` due to its simple structure, the highly symbolic programs `Chichelli` and `Clausify` might have shown some fusion.

Supercompilation and PRS — The combination of PRS and supercompilation (pipeline *d*) largely gives results as expected (PRS factor × supercompilation factor) with a few notable exceptions. `Adjoxo`, `Parts`, `sumDouble` and `sumSquares` all show better than expected performance, mainly for the reasons described in Section 6. However, `Queens` and `sumSumEnum` show considerably worse than expected performance, likely for the reasons outlined in Section 7.

Supercompilation, Lifting and PRS — The results of PRS, supercompilation *and* primitive redex lifting (pipeline *f*) indicate that this strategy is effective. Every program except for `While` performs better than the original program without PRS. However, `Taut` performs significantly worse than under our original supercompiler strategy. Across all our test programs, this strategy gives an average performance boost of 40%.

8.2 Compared to without Supercompilation on a PRS-enabled Platform

Table 2 gives another view on the impact of the supercompiler. It compares the number of Reduceron combinators produced (size), time taken to execute (cycles), number of case tables evaluated (cases) and the proportion of primitive operations performed by PRS. These are recorded for pipeline *f* and shown as multiples of the results for pipeline *b*, to the original program executed under PRS.

	Relative			Primitives by PRS		
	Size	Cycles	Cases	Original	Residual	Increase
`Adjoxo`	1.500	0.507	0.692	16.3%	87.0%	+81.3%
`Braun`	2.526	0.769	0.810	99.2%	100.0%	+0.8%
`Chichelli`	1.494	1.008	0.989	2.7%	2.6%	-5.2%
`Clausify`	5.879	0.951	0.829	0.0%	94.0%	$+\infty$
`Fib`	1.588	0.792	1.000	77.9%	77.9%	0.0%
`KnuthBendix`	2.508	0.866	0.855	63.8%	72.4%	+11.8%
`MSS`	1.920	0.996	0.991	0.0%	0.0%	-0.4%
`Mate`	2.423	0.954	0.918	54.8%	55.1%	+0.6%
`OrdList`	3.887	0.678	0.771	0.0%	0.0%	0.0%
`Parts`	1.733	1.009	0.983	38.8%	56.0%	+30.8%
`PermSort`	2.000	0.756	0.853	100.0%	41.8%	-139.2%
`Queens`	2.535	0.773	0.988	99.5%	85.4%	-16.5%
`Queens2`	2.068	0.966	0.947	4.0%	4.0%	0.0%
`Sudoku`	1.793	0.870	0.911	30.3%	56.1%	+45.9%
`Taut`	2.057	0.856	0.718	0.0%	99.9%	$+\infty$
`While`	3.030	1.062	0.989	60.2%	55.7%	-8.1%
`sumDouble`	0.450	0.200	0.333	50.0%	100.0%	+50.0%
`sumSquares`	0.900	0.378	0.494	66.2%	98.7%	+32.9%
`sumSumEnum`	1.435	0.943	0.494	66.2%	66.9%	+0.9%
Geometric Mean	*1.936*	*0.759*	*0.790*	*7.3%*	*21.7%*	

Table 2. Various metrics for the benchmark programs. Relative values are against the original program executed with PRS (pipeline *b*).

As would be expected, supercompilation can greatly increase the size of the compiled program. There does not seem to be a relationship between relative execution time performance and relative code size. A reduction in the number of cases table evaluated would likely indicate that fusion has taken place, as fewer data structures have been consumed.

In three cases, pipeline *f* still produces programs that perform worse than under PRS alone (pipeline *b*). In comparison to the gains made by other programs, these are only very small performance loses. The reason for these loses is currently unclear. Both for `Chichelli` and for `While`, the proportion of primitive operations performed by PRS has actually fallen. It is currently unclear why `Parts` performs worse when it has a large increase in the number of PRS candidates and a small amount of fusion.

Despite these results, the current prototype of our supercompiler gives a geometric mean speed-up of 24% for programs executed under PRS.

9 Conclusions and Future Work

The `sumDouble` example was chosen to demonstrate the benefits of both PRS and supercompilation. However, we did not see the full magnitude of the combined effect of the two technologies.

Other examples, such as `Queens`, did not benefit from supercompilation. This led to the development of the primitive redex lifting strategy that has largely permitted these examples to benefit from the same effects as `sumDouble`. This strategy does not seem to produce benefits for all results. Further investigation is required to discover why some results still do not improve and a small number get worse.

Still, based on the evidence detailed in this paper, it would appear that PRS and supercompilation can be synergistic, once certain primitive redexes are relocated to maximise design constraints.

There is further scope to exploit the Reduceron design characteristics with the supercompiler. A final inlining phase is required after supercompilation to reduce instantiations at run-time. The current method for selecting candidates for inlining is simplistic. Further performance gains can be made by an improved inlining strategy that considers the constraints on function bodies imposed by the design parameters of the Reduceron.

Future designs of the Reduceron will also permit even more primitive redexes to be speculatively evaluated in parallel. This will likely enable even further performance gains from supercompilation targeted at the Reduceron platform.

Acknowledgements The first author would like to thank Michael Banks and Chris Poskitt for their invaluable proof reading. Neil Mitchell is to be acknowledged for his helpful discussions on the topic. Many thanks also go to Richard Paige for his support and advice. This research was supported, in part, by the EPSRC through the Large-Scale Complex IT Systems project, EP/F001096/1.

References

1. Turchin, V.: A supercompiler system based on the language Refal. ACM SIGPLAN Notices **14**(2) (1979) 46–54
2. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming **6**(06) (2008) 811–838
3. Naylor, M., Runciman, C.: The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In: Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers), Springer LNCS 5083 (2008) 129–146
4. Naylor, M., Runciman, C.: The Reduceron Reconfigured. Available online at `http://www.cs.york.ac.uk/fp/reduceron/` (2010)

5. Naylor, M.: F-lite: a core subset of Haskell. Available online at `http://www.cs.york.ac.uk/fp/reduceron/memos/Memo9.txt` (2008)
6. Peyton Jones, S.L., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming **13**(1) (Jan 2003)
7. van Eekelen, M., Plasmeijer, R.: Concurrent Clean Language Report (version 2.0). University of Nijmegen (2001)
8. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
9. Partain, W.: The nofib Benchmark Suite of Haskell Programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Springer-Verlag (1992) 202
10. Bird, R.: A program to solve Sudoku. Journal of Functional Programming **16**(6) (2006) 671
11. Nielson, H.R., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons, Inc., New York, NY, USA (1992)
12. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: IFL 2007. Volume 5083 of LNCS., Springer-Verlag (May 2008) 147–164