Supercompilation and the Reduceron

Jason S. Reich, Matthew Naylor & Colin Runciman
<jason,mfn,colin@cs.york.ac.uk>

THE UNIVERSITY of York

3rd July 2010

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

Simon Marlow, 2009

The Reduceron

- Special-purpose graph-reduction machine. (Naylor and Runciman, 2007 & 2010)
- Implemented on a Field Programmable Gate Array. (FPGA)
- Evaluates a lazy functional language;
  - Close to subsets of Haskell 98 and Clean.
  - Algebraic data types.
  - Uniform pattern matching by construction.
  - Local recursive variable bindings.
  - Primitive integer operations. ($+$, $-$, $=$, $\leq$, $\neq$, *emit*, *emitInt*)
- Exploits low-level parallelism and wide memory channels in reductions.
- *See ICFP'10 paper "The Reduceron Reconfigured".*

Our source language

$$prog := \overline{f\ \overline{vs} = x} \quad (declarations)$$

$$
\begin{aligned}
exp := {} & v && (variables) \\
\mid {} & c && (constructors) \\
\mid {} & f && (functions) \\
\mid {} & f^P && (primitive\ function) \\
\mid {} & n && (integers) \\
\mid {} & x\ \overline{xs} && (applications) \\
\mid {} & \textbf{case}\ x\ \textbf{of}\ \overline{c\ \overline{vs} \rightarrow y} && \\
\mid {} & \textbf{let}\ \overline{v = x}\ \textbf{in}\ y &&
\end{aligned}
$$

An example

```
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;
```

After case elimination

```
foldl f z xs = xs [foldl#1,foldl#2] f z;
foldl#1 y ys t f z = foldl f (f z y) ys;
foldl#2 t f z = z;

map f xs = xs [map#1,map#2] f;
map#1 y ys t f = Cons (f y) (map f ys);
map#2 t f = Nil;

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

double x = (+) x x;
sumDouble xs = sum (map double xs);

range x y = (≤) x y [range#1,range#2] x y;
range#1 t x y = Nil;
range#2 t x y = Cons x (range ((+) x 1) y);

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

Reduction of an expression

```
range 0 10
```

Reduction of an expression

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (≤) 0 10 [range#1,range#2] 0 10
```

Reduction of an expression

```
  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
```

Reduction of an expression

```
  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
```

Reduction of an expression

```
  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)
```

Four cycles to reduce to HNF.

Reduceron performance

- The Reduceron is running on a Xilinx Virtex-5 FPGA clocking at 96 MHz.
- Compare with an Intel Core 2 Duo E8400 clocking at 3 GHz.
- Sixteen benchmark programs.

Reduceron performance

- The Reduceron is running on a Xilinx Virtex-5 FPGA clocking at 96 MHz.
- Compare with an Intel Core 2 Duo E8400 clocking at 3 GHz.
- Sixteen benchmark programs.
- On average, 4.1x slower than GHC -O2.

Reduceron performance

- The Reduceron is running on a Xilinx Virtex-5 FPGA clocking at 96 MHz.
- Compare with an Intel Core 2 Duo E8400 clocking at 3 GHz.
- Sixteen benchmark programs.
- On average, 4.1x slower than GHC -O2.
- On average, 5.1x slower than Clean.

Primitive redex speculation

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (≤) 0 10 [range#1,range#2] 0 10
```

Primitive redex speculation

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (≤) 0 10 [range#1,range#2] 0 10
```

- If tracing reduction by hand, *you* would evaluate the primitive.
- Why not the Reduceron?
- Primitive redex speculation (PRS) (*currently*) evaluates up to two primitives as the body is instantiated.
- Breaks laziness but as we are only dealing with reducible. primitives, always terminates.
- Low cycle cost, often zero!

Reduction using PRS

```
range 0 10
```

Reduction using PRS

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (≤) 0 10 [range#1,range#2] 0 10
= { Primitive redex speculation (0 cycle) }
  True [range#1,range#2] 0 10
```

Reduction using PRS

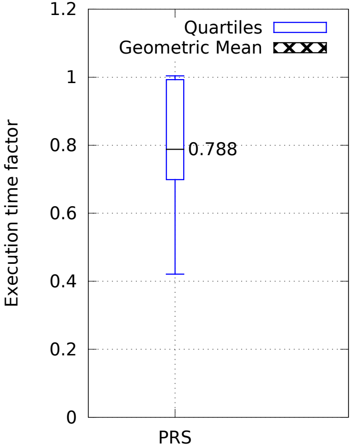
```
   range 0 10

 = { Instantiate function body (1 cycle) }
   (≤) 0 10 [range#1,range#2] 0 10
 = { Primitive redex speculation (0 cycle) }
   True [range#1,range#2] 0 10

 = { Constructor reduction (0 cycle) }
   range#2 [range#1,range#2] 0 10
```

Reduction using PRS

```
  range 0 10

= { Instantiate function body (1 cycle) }
  (≤) 0 10 [range#1,range#2] 0 10
= { Primitive redex speculation (0 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)
= { Primitive redex speculation (0 cycle) }
  Cons 0 (range 1 10)
```

Three cycles to reduce further than HNF.

Performance using PRS

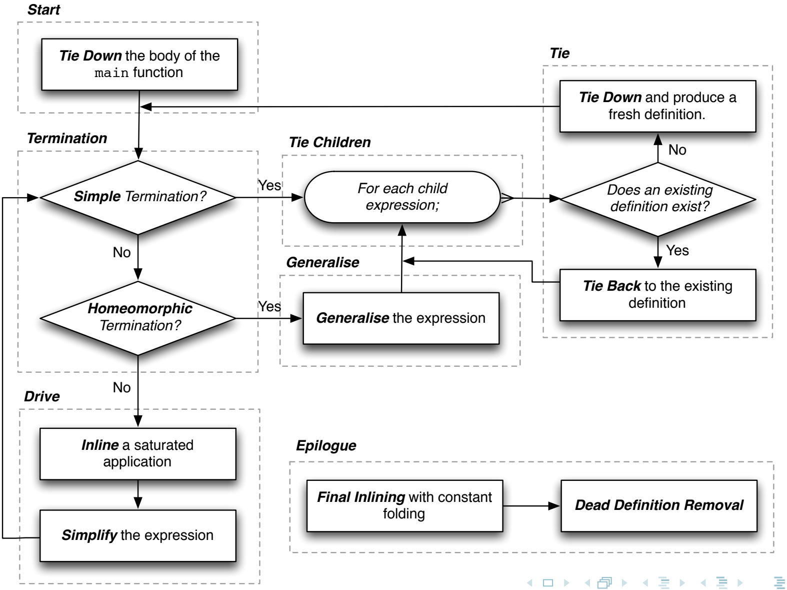


- Best speed-up — `Queens` by 2.4x.
- `Taut` has a marginal performance hit but is the only one.
- Nine out of nineteen examples see a speed-up of 1.1x or better.

Supercompilation

- A source-to-source compilation time optimisation
- Reduces the program as far as possible at compile-time.
- Where an unknown is required, proceeds by case analysis as far as possible.
- Can remove intermediate data structures and specialise higher-order functions.
- Our supercompiler is similar in design to that of Mitchell and Runciman. (2008)

Supercompilation



Drive

1. Inline the first saturated non-primitive application that does not cause driving to terminate. If all inlines cause termination, inline the first anyway.
2. Simplify the resulting expression using the twelve applicable simplifications listed in Peyton Jones and Santos (1994) and Mitchell and Runciman. (2008)

Terminal Forms

Simple termination

Terminate if expression is a;

- v (*free variable*)
- c (*constructor*)
- n (*integer*)
- $v\ \overline{xs}$ (*app. to free*)
- $f^P\ \overline{xs}$ (*prim. app.*)
- **case** v **of** $\overline{c\ \overline{vs} \rightarrow x}$
- **case** $v\ \overline{xs}$ **of** $\overline{c\ \overline{vs} \rightarrow x}$
- **case** $f^P\ \overline{xs}$ **of** $\overline{c\ \overline{vs} \rightarrow x}$

Homeomorphic termination

Terminate if the expression homeomorphically embeds a previous derivation.

$$
\begin{aligned}
x \trianglelefteq y &= \mathit{dive}\ x\ y \lor \mathit{couple}\ x\ y \\
\mathit{dive}\ x\ y &= \mathit{all}\ ((\trianglelefteq)\ x)\ (\mathit{children}\ y) \\
\mathit{couple}\ x\ y &= x \approx y \\
&\quad \land\ \mathit{and}\ (\mathit{zipWith}\ (\trianglelefteq) \\
&\quad (\mathit{children}\ x)(\mathit{children}\ y))
\end{aligned}
$$

Generalisation

If a homeomorphic embedding is detected, attempt to *generalise* the current expression.

1. If expressions are related by coupling, use most specific generalisation. (Sørensen and Glück, 1995)
2. Otherwise, if the expression does not depend on any local bindings, lift the subexpression that is coupled with the embedding. (Adapted from Mitchell and Runciman for a lambda-less language.)

Generalisation

If a homeomorphic embedding is detected, attempt to *generalise* the current expression.

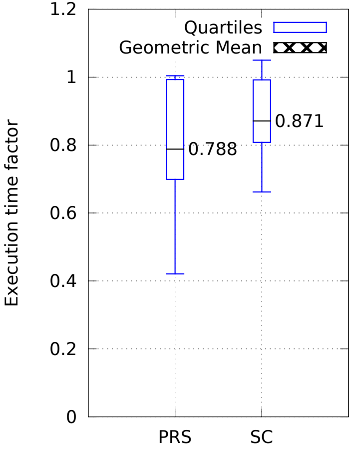
1. If expressions are related by coupling, use most specific generalisation. (Sørensen and Glück, 1995)
2. Otherwise, if the expression does not depend on any local bindings, lift the subexpression that is coupled with the embedding. (Adapted from Mitchell and Runciman for a lambda-less language.)

Tie

For each child expression;

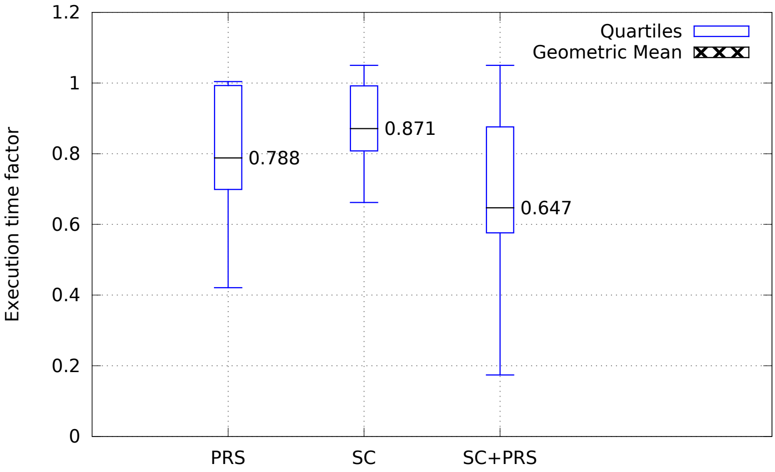
1. Tie back *(fold)* — Where possible, replace the expression with an equivalent application of a previously derived definition.
2. Tie down *(residuate)* — Otherwise, replace the expression with an equivalent application of a newly produced definition and drive the new definition.

Performance using Supercompilation



- Best speed-up — `Ordlist` by 1.5x.
- `Taut` speeds up by 1.4x!
- `Clausify` gets marginally worse.
- Ten out of nineteen examples see a performance increase of more than 1.1%.

Performance through combined SC and PRS



Why does sumDouble do so well?

sumDouble supercompiled

```
h4 v v1 = case ((≤) v1 10000) of {
     False → v;
     True  → h4 ((+) v ((+) v1 v1)) ((+) v1 1) };

main = emitInt (h4 6 3) 0
```

- Gone from eight definitions to just two.
- Benefits from the removal of intermediate data structures.
- More PRS as the `foldl plus` expression has been specialised.
- Speed-up by a factor of 5.8x!

Why is Queens disappointing?

- Speed-up factor of 2.38x under PRS.
- Only 2.04x under SC+PRS.
- Supercompiler splits primitive redexes across case alternatives.
- The original program evaluated some primitives speculatively and in parallel.
- Supercompiled program does not utilise this feature.
- Not a one off, can happen to any program. Just particularly noticeable in `Queens`.

Primitive Lifting

- PRS can evaluate up two primitive redexes for free with each Reduceron body instantiation.
- Reduceron bodies map to source language;
  1. Function definitions.
  2. Case alternatives.
- Move the primitive redexes to maximise utilisation of this feature.
- Extract things that are potential primitive redexes as let-bindings.
- Lift the binding to the highest valid body root that has spare capacity, prioritising the expressions coming through less case distinctions.

Return to sumDouble

```
h4 v v1 = case ((≤) v1 10000) of {
    False → v;
    True  → h4 ((+) v ((+) v1 v1)) ((+) v1 1) };
```

Return to sumDouble

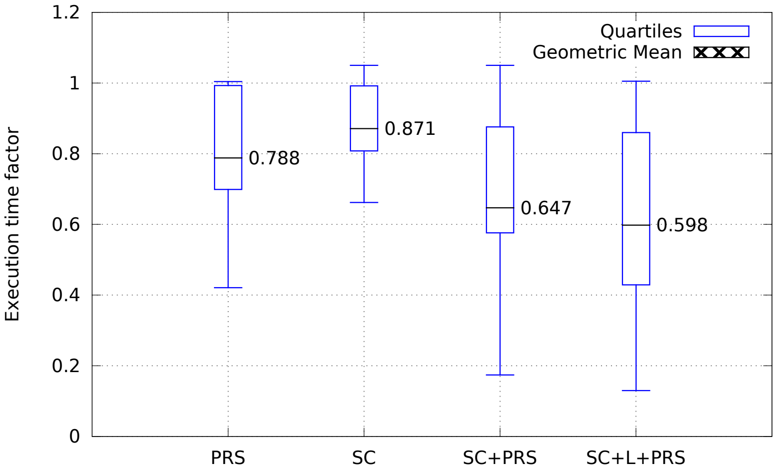
```
h4 v v1 = case ((≤) v1 10000) of {
    False → v;
    True  → h4 ((+) v ((+) v1 v1)) ((+) v1 1) };
```

```
h4 v v1 = let {
    prs = (+) v1 v1;
    prs1 = (≤) v1 10000
  } in (case prs1 of {
        False → v;
        True  → let {
            prs2 = (+) v1 1;
            prs3 = (+) v prs
            } in (h4 prs3 prs2)
        });
```

Laziness vs. Speculation

- Supercompilation simplifications are permitted to duplicate code as long as they do not duplicate computation. e.g. Let-bindings down case alternatives.
- Lifting primitive expressions will bring the duplicate code above case distinctions.
- Doesn't matter under lazy evaluation.
- Wastes resources under speculative evaluation.
- Solution: Merge duplicate expressions into a single binding.

Performance using PRS, SC and Lifting



Summary

- Primitive-heavy programs can benefit from PRS.
- Supercompilation can speed up programs by removing intermediate data structures and specialising higher-order functions.
- Supercompilation aids PRS by making primitive redexes apparent where they were not previously.
- Further transformation is required to maximise utility of PRS.
- Results in an average combined speed-up by 1.7x.
- With SC, PRS and lifting, the Reduceron is now only 2.5x slower than GHC -O2 on Intel.

Conclusions

- x86 processors aren't the only way to execute functional code.

Conclusions

- x86 processors aren't the only way to execute functional code.
- If we rethink our execution, we have to rethink our optimisations.

Conclusions

- x86 processors aren't the only way to execute functional code.
- If we rethink our execution, we have to rethink our optimisations.
- PRS and Supercompilation are not just complementary but synergistic.

Conclusions

- x86 processors aren't the only way to execute functional code.
- If we rethink our execution, we have to rethink our optimisations.
- PRS and Supercompilation are not just complementary but synergistic.
- Must always ensure that we consider execution model when developing transformations.

Further Work

- Further investigation of disappointing examples.

Further Work

- Further investigation of disappointing examples.
- Availability analysis;
  - Better detection of potential primitive redex.
  - Static PRS. More efficient, raises limit to eight primitive reductions.

Further Work

- Further investigation of disappointing examples.
- Availability analysis;
  - Better detection of potential primitive redex.
  - Static PRS. More efficient, raises limit to eight primitive reductions.
- Push on to 2.0x as slow as GHC -O2 on Intel.

Further Work

- Further investigation of disappointing examples.
- Availability analysis;
  - Better detection of potential primitive redex.
  - Static PRS. More efficient, raises limit to eight primitive reductions.
- Push on to 1.5x as slow as GHC -O2 on Intel.

Further Work

- Further investigation of disappointing examples.
- Availability analysis;
    - Better detection of potential primitive redex.
    - Static PRS. More efficient, raises limit to eight primitive reductions.
- Push on to same speed as GHC -O2 on Intel.

Further Work

- Further investigation of disappointing examples.
- Availability analysis;
  - Better detection of potential primitive redex.
  - Static PRS. More efficient, raises limit to eight primitive reductions.
- Push on to 2.0x as fast as GHC -O2 on Intel.