

---

# Superlinear Speedup by Program Transformation

---

Neil D. Jones

DIKU, University of Copenhagen (prof. emeritus)

Joint work with Geoff Hamilton, Dublin City University

# CONTEXT

---

A number of **fully automatic program transformers** exist, including:

- ▶ classical compiler optimisations
- ▶ deforestation
- ▶ partial evaluation
- ▶ supercompilation and distillation

**Rules of the game:** to obtain

- ▶ Fully automatic transformation
- ▶ without human interaction, e.g.,
- ▶ not using a mathematician or even an interactive theorem-prover
- ▶ or unbounded search procedures

**A dream:** transformations

- ▶ as **well-systematised and reliable** as classical compiler techniques, but
- ▶ able to yield **superlinear program speedup**.

# OVERVIEW, A BIRD'S EYE VIEW

---

1. Partial evaluation (Jones et al; without partially static structures)
2. Supercompilation (Turchin, M.H. Sørensen, Glück, Jones, Klyuchnikov)
3. Distillation (Geoff Hamilton)

**Program speedups:**  $speedup_p(s, d) = time_p(s, d) / time_{p_s}(d)$

- ▶ 1 and 2: proven at most linear  $\forall s \exists c \forall d ( speedup_p(s, d) \leq c )$
- ▶ 3: superlinear on some programs.

Which ones? Where does the speedup come from?

**Bottom-up, mainly by some simple examples:**

- ▶ Classical compiler optimisations
- ▶ Søren Debois: Partial evaluation does classical compiler optimisations  
by interpreter specialisation.
- ▶ Functional program distillation: Fibonacci, naive reverse
- ▶ Imperative program distillation: nested loops, factorial sum

# SPEEDUP: LINEAR IN WHAT?

---

First idea: a transformation is

$$p \mapsto p'$$

Desirable:  $\llbracket p \rrbracket = \llbracket p' \rrbracket$ .

Though Turchin would allow  $\llbracket p \rrbracket \sqsubseteq \llbracket p' \rrbracket$ .

This is like **compiler optimisation**.

Natural first speedup measure

$$\text{speedup-optimize}_p(d) = \frac{\text{time}_p(d)}{\text{time}_{p'}(d)}$$

# CLASSICAL COMPILER OPTIMISATIONS

---

- ▶ Constant propagation
- ▶ Code motion (out of a loop)
- ▶ Detecting common subexpressions
- ▶ Dead code elimination
- ▶ Strength reduction, etc etc etc

## Some common characteristics:

- ▶ Based on program data flow analysis (a.k.a. abstract interpretation)
- ▶ Usually give

**useful but at most linear speedups**

- ▶ Minimal **changes to program structure** are made, eg **loop unrolling** is uncommon
- ▶ Most compile-time computations operate **within a single basic block**

# LINEAR SPEEDUP FOR MULTI-INPUT PROGRAMS?

---

Program specialisation: transformation is ( $s =$  static data)

$$(p, s) \mapsto p_s$$

Correctness:

$$\forall s \forall d ( \llbracket p \rrbracket (s, d) = \llbracket p'_s \rrbracket (d) )$$

**Partial evaluation** and **supercompilation** are examples

A speedup measure

$$speedup_s(d) = \frac{time_p(s, d)}{time_{p_s}(d)}$$

**Theorem** (Jones, Andersen, Gomard, Sørensen) on constant-limited speedup:

$$\forall s \exists c \forall d \text{ speedup}_s(d) \leq c$$

**Example: string-matching** pattern against subject.

- ▶ KMP gives “only” linear speedup.

**Note:**  $c$  can depend on  $s$ , e.g.,  $|pattern|$ .

- ▶ One can quibble, eg if  $|pattern| = O(|subject|)$  is it quadratic speedup?.

# MORE ABOUT THE LINEAR SPEEDUP BARRIER

---

Transformation by interpreter specialisation, NDJ, Science of Computer Programming, 2004.

The arguments

- ▶ that partial evaluation and supercompilation speedups are at most linear
- ▶ depend on an assumption:  $\exists$  an order-preserving mapping between
- ▶ forced operations in the original computations and specialised ones.

Other assumptions can yield superlinear speedup.

Example: Elimination of “semantically dead” code:

- ▶ Remove code whose execution does not influence the program’s final results
- ▶ Effect: there are source code operations with no correspondents in the transformed program.
- ▶ Extreme case: transformed program runs in constant time, but source program does not. Can give arbitrarily high speedup!

# WHERE CAN SPEEDUP COME FROM?

---

1. **Dead computations:** their result does **not enter into the program's final output**.  
To exploit: just remove dead code
2. **Repeated computations:** the same problem is **solved more than once**.  
(Some ways to exploit: **tupling, Cook's inear-time 2DPDA simulation, memoisation,...**)
3. **Similar computations:** several subproblems can be **solved by the same computation**.  
To exploit: many forms of **generalisation**.
4. **Compactifying the output representation.** Example: the Towers of Hanoi problem.
5. **Change of algorithm.** Examples: merge sort versus quick sort; variations on FFT.



# WAYS TO BREAK THE LINEAR SPEEDUP BARRIER

---

**Removing repeated subcomputations** is another way to achieve superlinear speedup, less trivial than just eliding useless computations (cf. tupling, memoization).

**Transformation-time detection of dead branches:**

A logic programming analogy: detection of branches in Prolog's SLD computation tree that are guaranteed to fail.

**Much more significant than in functional programming:**

- ▶ Time-consuming sequences of useless operations are most likely a sign of bad functional programming; **but**
- ▶ the ability to detect and prune fruitless branches of search trees is part of the essence of logic programming.

# ARE THERE LIMITS TO SPEEDUP?

---

▶ Lower bounds are hard to find!

(But some exist, e.g.,  $n \log n$  for sorting.)

▶ Complexity-theoretic lower bounds exist, e.g.,  $p\text{time} \subset \text{exptime}$ .

(But seem always to be **unnatural problems**, e.g., obtained by **diagonalisation...**)

▶ Expectations:

- There are limits to speedups obtained by **program transformation**
- Algorithm change can give more, but is probably uncomputable in general
- And **you can't defeat lower bounds**

# A SILLY EXAMPLE OF SUPERLINEAR SPEEDUP

---

## Program:

Time  $\Omega(2^{O(|xs|)})$

```
f xs ys [42] where
f xs ys zs = case xs of [] => ys
                | x : xs' => f xs' (1 : ys) (f xs' ys (zs : zs))
```

## Optimised program: just discard zs; It's dead!

Time  $O(|xs|)$

```
f1 xs ys where
f1 xs ys = case xs of [] => ys
                | x : xs' => f1 xs' (1 : ys)
```

## Well... this

- ▶ is an example of bad programming
- ▶ if we assume call-by-value semantics

## Odd remarks:

- ▶ This speedup wouldn't happen in call-by-name, since
- ▶ the `zs` code would just not be executed.

# PARTIAL EVALUATION: BIRD'S EYE VIEW (offline PE)

---

## Preprocess time:

- ▶ Divide **program inputs** into
  - Static: will be known at transformation time
  - dynamic: will be unknown
  - Note: each variable is totally static or totally dynamic
- ▶ BTA (binding-time analysis):  $p \Rightarrow p^{ann} =$  annotated program: **every operation and function call** is marked as either
  - static: do while transforming (evaluate, or unroll a function call); or
  - dynamic: generate residual code.

## Transformation time:

- ▶ **Given:** program  $p^{ann}$  and static data  $d$
- ▶ **Perform:** all statically annotated bits (compute or unroll function calls)
- ▶ **Generate residual code:** for all the dynamically annotated bits

**Well-automated:** partial evaluators exist for SCHEME, C, PROLOG, ...

# SUPERCOMPILATION: ESSENTIALLY online

---

**Given:** program  $e_0$  where  $f_1 = e_1 \dots f_n = e_n$  (call-by-name semantics)

1. **Driving:** unfold  $e_0$  (only at *needed* operations). Gives a **process tree**:
2. **Unfold** case  $ce_1 \dots e_n$  of  $pattern_1 \Rightarrow e'_1 \mid \dots \mid pattern_n \Rightarrow e'_n$
3. case  $x$  of  $pattern_1 \Rightarrow e_1 \mid \dots \mid pattern_n \Rightarrow e_n$ 
  - ▶ Generate a residual case expression
  - ▶ Drive each  $e_i$  in an extended environment  $env[x \mapsto pattern_i]$
4. Similar for function calls and constructor applications. Effect: *positive information propagation*
5. Expressions  $e_i$  may be mixed static and dynamic
6. “**Blow a whistle**” when **danger of nontermination** is detected.
  - ▶ **Homeomorphic embedding** is a well-quasi order on expressions.
  - ▶ **What?** To decide where to “tie a loop” in the residual program.
  - ▶ **How?** By **generalising** expressions (LSG operation is dual to MGU)
7. Homeomorphic embedding tests are **very expensive** (frequent and slow)

# DISTILLATION: MUCH LIKE SUPERCOMPILATION, BUT:

---

Generalisation is done

- ▶ not with respect to **expressions**, but
- ▶ with respect to **process trees**.  
(Something like matching one tree automaton against another, but complicated by bindings, calls, constructors and cases.)
- ▶ Payoff: more complex transformation; can give non-silly **superlinear speedup**.

**Supercompilation** speedup (Sørensen): for any expression  $e$  supercompiled into  $e' = \mathcal{C}[[e]]$ , there **is a constant  $c$**  s.t. for all ground substitutions  $\theta$ :

$$c \cdot \mathcal{C}[[e'\theta]] \geq \mathcal{C}[[e\theta]]$$

**Distillation speedup** (Hamilton): there exist expressions  $e$  distilled into  $e' = \mathcal{D}[[e]]$  such **there is no constant  $c$**  such that for all ground substitutions  $\theta$

$$c \cdot \mathcal{D}[[e'\theta]] \geq \mathcal{D}[[e\theta]]$$

For example:  $e$  has one free variable  $x$ , and:

$$time_{e'}(x) = (|x|) \text{ but } time_e(x) = (|x|)^2$$

# WHAT IS GOING ON?

---

- ▶ Which programs allow superlinear speedup?
- ▶ Where does the speedup come from ?

## Remarks:

- ▶ There is an interesting **phenomenon** here, not yet well understood.
- ▶ Alas, distillation is **too complex** to get an easy overview.
- ▶ An alternative: study the problem by means of examples
- ▶ Simplify the context, by **reducing some of assumptions** from distillation:
  - Higher-order functions
  - Call-by-name (both function calls and constructors)
  - Nested function calls, eg  $f(g(x), h(y, z))$(Aim: to “drive the problem into a corner”)
- ▶ A much simpler context is a classical **compiler intermediate language**:
  - First-order flowchart programs

Is **the phenomenon** still present?

# PARTIAL EVALUATION: A BIT MORE THAN COMPILERS

---

- ▶ Unlimited static computations
- ▶ Unlimited loop unrolling

**Søren Debois (PEPM 2004)**: partial evaluating a self-interpreter can achieve several classical compiler optimisations, without data flow analysis, eg

- ▶ code motion
- ▶ strength reduction

**Trick**: write a “smart self-interpreter”, e.g., maintain a (finite) memory of

- ▶ **assignments that have been seen before**, so the interpreter
- ▶ **never re-executes** an already-executed statement.

An effect is to

- ▶ unroll a loop when first encountered; and to
- ▶ generate loop code on the second time around, but
- ▶ without **first-time-around computations** that are still available



# SUPERLINEAR SPEEDUP OF FUNCTIONAL PROGRAMS: I

---

## Fibonacci function:

Time  $2^{O(n)}$

fib n where

fib n = case n of 0 => 1

| n'+1 => case n' of 0 => 1

| n''+1 => (fib n') + (fib n'')

## Distilled Fibonacci function:

Time  $O(n)$

f n 1 1 where

f n x y = case n of 0 => 1

| n'+1 => f n' (x+y)

**Source of speedup:** shared function calls (fib n makes 2 calls to fib n-2)

# SUPERLINEAR SPEEDUP OF FUNCT. PROGRAMS: II

---

## Naive reverse:

Time  $O(n^2)$

nrev xs where

```
nrev xs ys = case xs of [] => []
              | x :: xs' => append (nrev xs') [x]
```

```
append us vs = case us of [] => vs
```

```
              | w : ws => w : (append ws vs)
```

## Distilled reverse function:

Time  $O(n)$

arev zs where

```
arev zs      = arev' zs []
```

```
arev' zs acc = case zs of [] => acc
```

```
              | y' : ys' => arev' ys' (y' :: acc)
```

**Source of speedup:** semantically dead values. **Concretely:** nrev [1,2,...,n] makes calls to nrev [2,...,n] and nrev [3,...,n] and ... and nrev [n].

**A tricky point** : it is hard to see just where and when the produced intermediate values are consumed. (And even harder with call-by-name!)

# SUPERLINEAR SPEEDUP OF IMPERATIVE PROGRAMS

---

First conclusions from the previous slides:

- ▶ Nested function calls  $f(g(x), h(y, z))$  complicate things
- ▶ Call-by-name complicates things
- ▶ Natural question: **can similar phenomena occur with imperative programs**, i.e., with
  - Tail-recursive programs and
  - Call-by-value ?

This led to some experiments (by eye and by running the distiller).

- ▶ The answer was **yes**.
- ▶ Now we're trying to understand **why? and how?**.

# NESTED LOOP EXAMPLE

## Program:

Time  $O(m * n)$

`g u v m n` where

```
g u v x y = case x of 0      => Pair u v      -- output
                  1+x' => h 1+u 0 x' n      -- g calls h
```

```
h u v x y = case y of 0      => g u v x y      -- h calls g
                  1+y' => h u 1+v x y'        -- h calls h
```

## Analysis:

### 1. `g` calls `h` while

- ▶ **resetting** `v` and `y`, and
- ▶ **incrementing** `u` and **decrementing** `x`

### 2. Then `h` can call `g`; or it

- ▶ can call itself, **incrementing** `v` and **decrementing** `y`.

### 3. Output depends on **both** `m` and `n`; **but** `v` is recomputed again and again.

**Optimisation:** move the inside loop (either up or down). Time  $O(m + n)$

# NESTED LOOPS: OPTIMISED

## Original program:

Time  $O(m * n)$

```
g u v m n where
g u v x y = case x of 0    => Pair u v      -- output
                  1+x' => h 1+u 0 x' n    -- g calls h

h u v x y = case y of 0    => g u v x y    -- h calls g
                  1+y' => h u 1+v x y'    -- h calls h
```

## Distilled program:

Time  $O(m + n)$

```
case m of 0    => Pair u v      -- output
         | 1+x' => r 1+u x' n
r u x n = case x of 0    => s u 0 n
              1+x' => r 1+u x' n
s u v y = case y of 0    => Pair u v
              1+y' => s u 1+v y'
```

**Somehow...** this looks very elementary; but beyond an optimising compiler!

**Explanation try: idempotence:**  $[[code; code]] = [[code]]$

# SIMILAR, BUT OUTER LOOP AFFECTS INNER LOOP

---

## Original program:

Time  $O(m * n)$

```
g m n u v m n where
g m n u v x y =
  case x of 0    => Pair u v          -- output
            1+x' => h m 1+n 1+u 0 x' n -- g calls h, increases n
h m n u v x y = case y of 0    => g m n u v x y      -- h calls g
                        1+y' => h m n u 1+v x y'     -- h calls h
```

## Distilled program:

Time  $O(m + n)$

```
case n of 0 => Pair u v
         | 1+n' => h m n u 0 m n'      where
h m n u v x y = case y of
                 0    => g m n u v x
                 | 1+y' => h m n u 1+v x y'
g m n u v x = case x of
               0    => Pair u v
               | 1+x' => g m 1+n 1+u 1+v x'
```

**Second explanation try: absorption:**  $[[code1; code2]] = [[code2]]$

# FACTORIAL SUM: DISTILLABLE QUADRATIC SPEEDUP

---

## Original program:

Time  $O(n^2)$

```
loop1 n 1 where
loop1 n sum = case n of
    0 => sum
  | 1+n1 => loop2 n 1 n1 sum;
```

The part to add the factorials

```
loop2 i prod n sum = case i of
    0 => loop1 n (sum + prod)
  | 1+i1 => loop2 i1 (i * prod) n sum;
```

The part to compute  $n!$

## Distilled program:

Time  $O(n)$

```
f n 0 where
f n x =
case n of 0 => 1+x
  | 1+n' => f n' 1+(x+(n'*(1+x)));
```

or simplified: `f n' n*(1+x)`

**Third explanation try: Neither earlier explanation works!** The distiller did this automatically, but it looks like it should require induction or similar...

## SUMMING UP

---

1. The distillation algorithm achieves some **nontrivial superlinear speedups**.
2. It is too complex for cause-and-effect to be clearly visible.
3. To understand limits, powers better, we have resorted to experiments, using a **severely restricted input language**.
4. Some **unexpected and nontrivial superlinear speedups** have been seen.
5. The severely restricted input language amounts to traditional **compiler intermediate language**.
6. **BUT**: traditional compiler optimisations do not yield such superlinear speedups.
7. This suggests: a **“turbo” version of compiler optimisations** that can achieve substantially greater speedups.  
Ideally, one that can run in times acceptable to a compiler.
8. More...? **Who knows, it is indeed “work in progress”**.