Superlinear Speedup by Program Transformation (Extended Abstract)

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
e-mail: neil@diku.dk

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
hamilton@computing.dcu.ie

There seems to be, at least in practice, a fundamental conflict within program transformations. One way: *hand transformations* can yield dramatic speedups, but seem to require human insight. They are thus are only suited to small programs and have not been successfully automated. On the other hand, there exist a number of well-known *automatic program transformations*; but these have been proven to give at most linear speedups.

This work in progress addresses this apparent conflict, and concerns the principles and practice of superlinear program speedup. A disclaimer: we work in a simple sequential program context: no caches, parallelism, etc.

Many interesting program transformations (by Burstall-Darlington, Bird, Pettorossi, and many others) have been published that give superlinear program speedups on some program examples. However, these techniques all seem to require a "Eureka step" where the transformer understands some essential property relevant to the problem being solved (e.g., associativity, commutativity, occurrence of repeated subproblems, etc.). Such transformations have proven to be very difficult to automate.

On the other hand a number of fully automatic transformers exist, including: classical compiler optimisations, deforestation, partial evaluation and positive supercompilation. However these can be seen (and have been formally proven, e.g., by Jones for partial evaluation, and by Sørensen for positive supercompilation) only to yield linear time improvements.

For example, a limit in automatic string pattern matching was for several years to achieve the speedup of the Knuth-Morris-Pratt algorithm. The KMP speedup is still linear though, although its constant coefficient can be proportional to the length of the pattern being searched for.

In 2007 Hamilton showed that his "distillation" transformation (a further development of positive supercompilation) can sometimes yield superlinear speedups. Distillation has automatically transformed the quadratic-time "naive reverse" program, and the exponential-time "Fibonacci" program, each into a linear-time equivalent program that uses accumulating parameters.

On the other hand, there are subtleties, e.g., distillation works with a higher-order call-by-name source language. Further, distillation is a very complex algorithm, involving positive information propagation, homeomorphic embedding, generalisation by tree matching, and folding. A lot of the complexity in the algorithm arises from the use of potentially infinite data structures and the need to process these in a finite way. It is not yet clear which programs can be sped up so dramatically, and when and why this speedup occurs. It is as yet also unclear whether the approach can be scaled up to use in practical, industrial-strength contexts, as can classical compiler optimisations.

The aim of this work in progress is to discover an essential "inner core" to distillation. Our approach is to study a simpler language, seeking programs that still allow superlinear speedup. Surprisingly, it turns out that asymptotic speedups can be obtained even for first-order tail recursive call-by-value programs (in other words, imperative flowchart programs). The most natural example (discovered just recently) transforms the natural but factorial sum program for f(n) = 1! + 2! +...+ n! from quadratic time to linear time.

Some examples that suggest principles to be discovered and automated:

- In functional programs:
  - finding shared subcomputations (e.g., the Fibonacci example)
  - finding unneeded computations (e.g., most of the computation done by "naive reverse")
- In imperative programs:
  - finding unneeded computations (e.g., generalising the usual compiler "dead code" analysis to also span over program loops can give quadratic speedups)

- finding shared subcomputations (e.g., the factorial sum example)
- code motion to move an entire nested loop outside an enclosing loop
- strength reduction
- common subexression elimination across loop boundaries, eg extending "value numbering"

Alas, these principles seem to be buried in the complexities of the distillation algorithm and the subtleties of its input language. One goal of our current work is to extract the essential transformations involved, ideally to be able to extend classical compiler optimisations (currently able only to yield small linear speedups) to obtain a well-understood and automated "turbo" version that can achieve substantially greater speedups.