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 supercompilation. Mostly these give only linear speedups. (There are, however, two Refal-based exceptions: the supercompiler work by Turchin, and Nemytykhs supercompiler SCP4.)

The limitation to linear time improvement has been proven in some cases, e.g., by Jones and colleagues for partial evaluation (using call-by-value), and by Sørensen for positive supercompilation (using call-by-name).

An instance: a goal for several years was automatically to achieve the speedup of the Knuth-Morris-Pratt string pattern matching algorithm. The KMP speedup is still linear though, although the constant speedup coefficient can be proportional to the length of the pattern being searched for.

What principles can lead to superlinear speedup? Some examples that suggest principles to be discovered and automated:

1. In functional programs:
   - finding shared subcomputations (e.g., the Fibonacci example)

   - finding unneeded computations (e.g., most of the computation done by "naive reverse")
2. In imperative programs:
   - finding unneeded computations (e.g., major speedups can result from generalising the usual compiler "dead code" analysis also to span over program loops)
   - 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, e.g., extending "value numbering"

In 2007 Hamilton showed that the "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). An example discovered just recently concerned computing $f(n) = 1!+2!+...+n!$. Distillation transforms the natural quadratic time factorial sum program into a linear time equivalent.

Even though distillation achieves many of these effects automatically, the principles above 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, one could extend classical compiler optimisations (normally only yielding small linear speedups) to obtain a well-understood and automated "turbo" version that achieves substantially greater speedups, and is efficient enough for daily use.

References

1. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

2. Søren Debois. Imperative program optimization by partial evaluation. In Heintze and Sestoft [5], pages 113–122.
3. Geoffrey William Hamilton and Neil D. Jones. Distillation with labelled transition systems. In *PEPM (ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation)*, pages 15–24. ACM, 2012.
4. Geoffrey William Hamilton and Neil D. Jones. Proving the correctness of unfold/fold program transformations using bisimulation. In *Proceedings of the 8th Andrei Ershov Informatics Conference*, volume 7162 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2012.
5. Nevin Heintze and Peter Sestoft, editors. *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004, Verona, Italy, August 24-25, 2004*. ACM, 2004.
6. Neil D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, 2004.
7. V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.