

Global Optimization for a Coagulating Code Generator: Final Technical Report*

Michael Karr, Walter G. Morris, Steve Rozen

<http://jura.wi.mit.edu/rozen>

June 20, 1991

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

Abstract

The Coagulating Code Generator (CCG) is designed to generate highly efficient code in terms of instruction selection, register usage, and procedure calls. These are not the only determinants of code efficiency, however. Optimizing compilers also try to improve code by a number of well-known transformations that we refer to as *global optimizations*; examples are global common subexpression elimination and moving invariant computations out of loops.

We report here our studies of the interaction between global optimizations and CCG based on experiments with selectively enabling specific optimizations and analysis of emitted code. We also summarize a comparison between CCG and a production optimizing compiler.

*This work was supported in part with funds provided by the Defense Advanced Research Projects Agency and the Naval Ocean Systems Center under contract N00014-85-C-0710 with the Office of Naval Research. The views and conclusions contained in this report are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Naval Ocean Systems Center, the Office of Naval Research, or the U.S. Government.

1 Introduction

The Coagulating Code Generator (CCG) performs register allocation and instruction selection in an interleaved fashion, following a strategy of *local optimality*. This means it starts by compiling the most frequently executed parts of a program, and locally allocates to them optimal resources [Kar84a, Kar84b, Mor91]. Then, when two already-compiled parts of the program are connected, adjustments may have to be made to ensure the availability of resources previously assumed to be available. However, these adjustments typically occur at less frequently executed parts of the program.

With this strategy CCG avoids any dichotomy between instruction selection and register allocation, and there is no phase analogous to peephole optimization. The strategy of local optimality extends to procedure calls, with the result that arguments and results are usually passed in registers, and local variables are placed on a stack only if absolutely necessary, i.e. when they are live across a recursive call. As suggested by the strategy of local optimality, and in contrast with more conventional code generators, CCG relies heavily on an execution profile and a detailed, explicit cost model of the target.

Despite the advantages of the strategy of local optimality, it does not detect possibilities for global transformations. We therefore considered global optimizations for CCG with two objectives in mind:

1. To transform the input to CCG sufficiently to enable an even-handed comparison with other optimizing compilers.
2. To understand the interactions between CCG and global optimizations.

2 Approach

Initially we inspected ten standard benchmark programs to determine those global optimizations most likely to be beneficial in conjunction with the CCG. The benchmarks we chose were the integer benchmarks (not requiring dynamic allocation) from the Stanford (Hennessy) suite, the Tak and Fib functions from [Gab85], plus a version of Sieve (of Eratosthenes). The flow-graph input for the CCG was prepared using E-L, a new wide-spectrum language and program development system [E-L86, Tow].

We considered the following global optimizations (described in [ASU86]).

- code motion (removal of loop-invariant expressions),
- reduction in strength,
- induction variable elimination,
- global common subexpression elimination,
- constant folding, and
- dead code elimination

2.1 Reduction in Strength (Finite Differencing)

Analysis of the probable effect of these optimizations on the benchmarks indicated that reduction in strength was the single most important optimization.

The term “reduction in strength” can be somewhat misleading. The central idea is to replace expensive operations by cheaper ones, for example $(4 * i)$ is usually more cheaply implemented as a shift. Presumably, then, “strong” operators are more expensive than “weak” ones, whence “reduction in strength”. In this case the optimization is quite local, and in fact can be provided by CCG.

However, in other cases reduction in strength involves a more global strategy of (cheaply) maintaining the value of an expensive expression as the operands of the expression change in some predictable way. The classical example is an array indexing expression in a loop over one of the indexes. Suppose the array indexing expression translates to, e.g.

$$\text{base} + (i * 3) + j,$$

where `base` is the address of the first element in the array, `i` is the row index and `j` is the column index. If this expression occurs in a loop as `i` goes from, say, 0 to 20, it can be replaced by `k`, where `k` is initialized to $(\text{base} + j)$ prior to the loop, and incremented by 3 every time `i` is incremented by 1.

In this context we prefer to call the transformation *finite differencing* [PK82, Pai84]. In our implementation finite differencing subsumes code motion since invariant expressions can be trivially maintained. Finite differencing was implemented to operate on base E-L, the λ -calculus-like kernel language for E-L. The design and implementation of finite differencing are fully described in [KRa].

Another optimization often provided in addition to and in conjunction with finite differencing is induction variable elimination. In the example above, assuming the termination condition for the loop involves `i` and that the only other use of `i` in the loop is at the increment, we could replace the termination condition with one involving `k`, and remove the increment of `i`. In conventional compilers the main consequence of failing to eliminate induction variables is increased register demand and the cost of the avoidable update instructions in the loop. We initially judged that for the benchmarks we used neither expense would be significant and did not implement induction variable elimination.

2.2 Simplification

In our initial experiments with CCG and finite differencing we observed that sometimes as a result of finite differencing several variables were created with the same constant value. In an effort to improve this and to reduce the size of the flow graph submitted to CCG we implemented a mechanism for partial β - and δ -reduction of the base E-L prior to the generation of the flow graph. The effect of these transformations was some constant propagation and constant folding. We called this transformation *simplification*. The code for simplification is found in [KRb], Subsection 4.8, (*β - and δ -Reduction*).

By itself, simplification marginally improves some programs, but degrades others, especially Sieve. This is because in the MC68020 immediate (literal) arguments may be significantly more expensive than register arguments [Mot85]. We solved this problem by

Benchmark	CCG	gcc	CCG/gcc	fd?
Bubble	116	124	0.94	yes
Fib	516	960	0.54	
IntMM	182	178	1.02	yes
Perm	100	126	0.79	no
Puzzle	98	112	0.88	yes
Queens	108	120	0.90	no
Quick	102	114	0.89	no
Sieve	104	136	0.76	no
Tak	130	258	0.50	
Towers	166	248	0.67	yes
Geometric Mean			0.77	

Figure 1: Benchmark Timings for the CCG and gcc.

augmenting CCG with the ability to place constants in registers as part of the normal coagulation process if cost estimates indicate this is beneficial. We call this “registerizing”. Registerizing appears to obviate any need for a switch to control simplification for a target architecture that can move a literal into a register as quickly as it can perform arithmetic involving literals, i.e. for any reasonable architecture.

3 Benchmark Results

We let finite-differencing transformations be controlled by a compiler switch, and compared CCG and another optimizing compiler, the GNU C compiler (gcc) [Sta]. The results are summarized in Figure 1. The second column shows the best time for CCG over both settings of the finite-differencing switch. Columns two and three show the run times in host-dependent units, with smaller numbers indicating faster times. Column four is the ratio of CCG’s to gcc’s time, in effect normalizing the timing to gcc. The geometric mean provides an average ratio between CCG and gcc times [FW86]. The fifth column, “fd?”, indicates whether the finite differencing transformation was enabled or disabled to achieve the CCG timing; there is no entry for programs that finite differencing does not affect. The timing for gcc was the best obtained in an exhaustive search over all combinations of four compiler switches controlling “optimizations” in addition to the basic `-O` switch.

The source code and a full discussion of benchmarking methodology are presented in [MR91].

3.1 Discussion of Comparison with gcc

A salient advantage of CCG is its efficient procedure call mechanism, and this shows clearly in the heavily recursive Fib and Tak benchmarks, and also in Perm, Queens, Quick and

Towers. (Puzzle is also recursive, but there is a two-dimensional array access in a loop in the most frequent procedure. As a result the efficiency of procedure calls is less important.) For the recursive benchmarks CCG is 39% faster than `gcc`.

In the non-recursive Bubble and IntMM the code emitted by CCG is approximately as good as that of `gcc`. The inferior performance of CCG on IntMM is due to an additional memory access in the inner loop. An attractive solution to this problem is to enhance the existing coagulation machinery to manage the staging of array slots between memory and registers in the way the CCG now handles virtual registers. This should allow CCG to eliminate two memory accesses in the inner loop, and result in a net speedup over `gcc`.

Finally, CCG is much faster than `gcc` on Sieve primarily because `gcc` fails to registerize a constant in the program, and secondarily because `gcc` places an important variable in memory rather than in a register (a placement apparently forced by the C language’s inter-module scoping rules).

Even for the non-recursive benchmarks CCG is 11% faster than `gcc` and for the geometric means of all ten programs CCG is 30% faster than `gcc`.

3.2 Discussion of the Effect of Finite Differencing

As Figure 1 suggests, finite differencing is an improvement only in some of the benchmarks, specifically Bubble, IntMM, and Puzzle. In IntMM finite differencing is especially important; without finite differencing the timing is 412 and with, 184.

On the other hand, in Perm, Queens, Sieve, and Towers finite differencing produces a less efficient program. In the case of Perm and Queens the problem is that finite differencing generates additional variables that are live across a recursive call and consequently must be saved in memory. This effect would likely not be noticed in a conventional run-time implementation with activation-based procedure calls, but in the CCG where variables are stacked only if necessary, the effect is dramatic. In Queens, where four induction variables must be introduced, finite differencing slows the timing from 114 to 156 units.

For Sieve the problem is that finite differencing adds complexity to a loop header that is in fact executed quite frequently. In addition, there is only a single array access in the innermost loop, and savings on that access do not counterbalance the additional update instruction in the loop. For Towers there is a slight degradation with finite differencing, from 160 to 162 units; the authors have not been able to identify the cause.

As noted above, no induction variable elimination was performed in conjunction with the finite differencing transformation. We hypothesize that induction variable elimination would provide an additional marginal speedup for Bubble, IntMM, and Puzzle (in each case probably less than 5%). It would also make finite differencing an improvement for Perm, but not for Queens. The situation for Sieve and Towers is not certain.

4 Future Research and Conclusions

The major open question raised by our study of global optimizations and the CCG is how to decide when to apply certain transformations.

Some transformations, such as the simplification transformation discussed above, present no difficulty. The program retains sufficient information that the effect of the transformation, if not the transformation itself, can be undone by the CCG. Other transformations, such as finite differencing, do not have this property. Finite differencing generally makes a program more complex by introducing additional variables and statements, and the inverse transformation is at least as difficult as the original.

One could accommodate such transformations using the conventional approach of providing compiler switches. However, because CCG relies on profile information and an explicit cost model, one would hope that the decision of whether to apply a transformation could be made automatically. Furthermore, given the interprocedural nature of CCG, it can be advantageous to apply a transformation in certain areas, but not others.

One potential solution is to project the target-specific cost model into earlier phases. The engineering cost would be additional compiler complexity, but a more fundamental problem is that costs cannot be completely estimated before registers and instructions are selected.

Another potential solution is to annotate CCG's input with alternative transformations and let CCG make the decision. This involves designing annotations to indicate potentially useful global transformations. An interesting issue is the effectiveness of the strategy of local optimality in optimizing over the space of the various combinations of candidate transformations. Otherwise, the question arises of how CCG could organize the search for a program with good global transformations. To the extent we can answer these questions the benefits of CCG's approach can be extended to a larger solution space. In the meantime the situation is no worse than for traditional optimizing compilers, e.g. gcc offers four different flags that affected the timings of the benchmarks; GCC burdens the programmer with only a single flag to control finite differencing.

Besides this general architectural issue the authors identified two additional transformations that would be beneficial to CCG.

1. Some recursive procedure calls involve the application of a cheap, invertible function to one of the parameters. An example would be the classical recursive definition of the factorial function, where `fact(x)` is defined in terms of `fact(x - 1)`. For CCG it may well be advantageous to set `x` to `(x - 1)` prior to the recursive call and then to `(x + 1)` after the recursive call, rather than to save a copy of `x` in memory. The reason is that two memory accesses are considerably more expensive than a single addition. (Note that `x` is live across the recursive call in the classical definition of `fact`.)

This transformation can be performed easily in base E-L, but here again the decision of when to perform the transformation is best made by CCG.

2. The authors have also identified a new, profile-based optimization that subsumes finite differencing and induction variable elimination. This transformation is completely interprocedural, and can have the effect of transforming a program written in terms of array indexes into a program predominantly using pointers into arrays, including procedure parameters that were formerly indexes. This transformation should be beneficial for more of the benchmarks than finite differencing, including Perm. The challenge remains of applying the transformation only when advantageous. A full report on this transformation will appear in [KRc].

In conclusion, the implementation of global optimizations for the CCG let us compare it with a highly-regarded optimizing compiler, `gcc`, and shows that CCG generates extremely efficient code. Using programmer-controlled switches CCG's timings were 30% faster than `gcc`'s on the ten benchmarks studied. Work is underway to include additional global optimizations and to use a profile- and cost-based approach for controlling global transformations. These changes should further improve CCG's advantage.

References

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.
- [E-L] *E-L System*. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138. Internal implementation documents.
- [E-L86] E-L definition. Technical Report SOI-03-86, Software Options, Inc., April 1986.
- [FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, March 1986.
- [Gab85] R. P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, Cambridge MA, 1985.
- [Kar84a] Michael Karr. *Code Generation by Coagulation*. PhD thesis, Harvard University, May 1984.
- [Kar84b] Michael Karr. Code generation by coagulation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 1–12, June 1984. (SIGPLAN Notices (19,6)).
- [KRa] Michael Karr and Steve Rozen. *Finite Differencing in Base E-L*. E-L implementation document in [E-L].
- [KRb] Michael Karr and Steve Rozen. *Flow-Graph Model: Implementation*. E-L implementation document in [E-L].
- [KRc] Michael Karr and Steve Rozen. Program optimization by group operations. In preparation.
- [Mor91] Walter G. Morris. CCG: A prototype coagulating code generator. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.
- [Mot85] Motorola. *MC68020 32-Bit Microprocessor User's Manual*. PRENTICE-HALL, Inc., Englewood Cliffs, NJ 07632, second edition, 1985.
- [MR91] Walter G. Morris and Steve Rozen. Interim report on E-L/CCG benchmarks, version 2. Technical Report SOI-01-90, Software Options, Inc., June 1991.
- [Pai84] Robert Paige. *Applications of Finite Differencing to Database Integrity Control and Query/Transaction Optimizations*, volume 2 of *Advances in Database Theory*, pages 171–209. Plenum Press, 1984.

- [PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM TOPLAS*, 4(3):402–454, 1982.
- [Sta] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 675 Mass. Ave., Cambridge MA 02139.
- [Tow] Judy G. Townley. E-L User's Manual. E-L implementation document in [E-L].