1. Background

Compiler writers have a difficult task when making their compilers produce efficient code. Due to pipe-lining, store forwarding, multi-level caching, prefetching, hyperthreading, multiple functional units, and the like, modern processor architectures are hard to model accurately yet simply enough to make it possible to solve optimization problems analytically. Given a machine code snippet, it is not always easy to estimate its efficiency in the context it will run. Moreover, a compiler makes many optimizing transformations. Many of these transformations (such as loop fission, loop fusion, loop tiling, loop unrolling) are only beneficial under certain circumstances and it is more or less impossible to predict analytically when and where to apply them. In addition, different transformations affect each other in complex ways, making it very hard to find the best order in which to apply them. Finally, in the cases we do know how to solve a problem analytically, the problem is likely to be NP-complete, implying that we do not have the time to solve it at compile-time. For all these reasons, modern compilers rely heavily on heuristics. Currently, these heuristics are found by tedious manual trial-and-error. What if we instead could use machine learning to find good heuristics automatically? The authors of the paper did some experiments to find out if that approach is feasible.

2. Introducing genetic programming

The machine-learning strategy selected for the experiments was genetic programming. Genetic programming is a search strategy inspired by Darwin’s ideas of biological evolution. The goal is to find a better solution to some problem. The method is to regard candidate solutions as individuals of a population subject to evolution from generation to generation. More precisely, the generic template of GP search looks like this:

1. Create an initial generation by randomly generating a population, but include individuals that for some reason are already known to be good.

2. Create the next generation by modifying the population as follows:
   a. Partition the population into tournaments of equal size. For each tournament, select the fittest individual as winner.
   b. Let the elite be the N fittest individuals for some natural number N.
   c. Kill some randomly selected individuals not belonging to the elite.
   d. Replace the dead ones by interbreeding the tournament winners.
   e. Let a few randomly selected individuals mutate in some random way.

3. Repeat the previous step a fixed number of times, then return the fittest individual in the last generation.

There are several important advantages of using GP:

1. GP can be used to solve problems without actually understanding them (that is, it works even when the relationship between the properties of a solution and its quality are poorly understood).
2. GP can search among complex solutions.
3. GP search is highly parallelizable.
4. The solutions found by GP are human readable, making it possible to find out at least something about why they are good.

The disadvantages are that GP seldom find optimal solutions and that search times easily can grow towards infinity if proper care is not taken. But these problems are not fatal in our case, because we can not find optimal solutions anyway and we have plenty of resources to do the search since it is done at compiler construction time rather than at compile-time (which is not to say that we do not need to care about the size of the search space).

3. Using GP to find heuristics

In order to apply GP to our particular problem, the problem of finding good heuristics for optimizations performed by a compiler, we need to instantiate the generic template from the previous section.

3.1 Search space definition

The individuals inhabiting our particular search space are heuristics for some particular optimization algorithm we want to improve. We could have chosen to regard complete optimization algorithms. But this would have caused the search space to become too large and polluted with incorrect algorithms. A heuristics can make an optimization algorithm bad, but never incorrect (in the sense of violating the semantics of the optimized code). Thus we do not need to care about correctness during fitness evaluation, and that makes things a lot easier.

3.2 Fitness evaluation

The fitness of a heuristics is estimated like this:

1. Use the heuristics to compile a some applications from a benchmark.
2. Run each resulting executable on several fixed input scenarios.
3. Measure the average time needed to produce the expected output.
4. Let the fitness be in inverse proportion to this average.

3.3 GP parameter settings

In the experiments described in the paper, the following configuration was used:
5. Population size: 400. 399 of these were randomly generated, one was the state-of-the-art heuristics subject to improvement (from now on called the “baseline”).
6. Number of generations: 50. (This resulted in search times less than one day on the computer cluster used for the experiments.)
8. Elite size: 1. (That is, the best individual was guaranteed to survive to the next generation.)
9. Mortality: 22%. (That is, kill 22% of the population was killed when creating the next generation.)
10. Mutation rate: 5%. (That is, 5% of the population was mutated when creating the next generation.)

The choice of these values was based solely on intuition. Future work includes some fine-tuning, manually or by using GP.

3.4 Gene definition

The concept of “heuristics” needs to be made more concrete in order to enable definitions of how off-springs and mutations should be generated. In other words, we need to identify the genes of the heuristics inhabiting our search space. A heuristics should be completely defined by its genes. It turns out that this can be done in a very straightforward way by encoding heuristics as priority functions defined by expressions. We use the ASTs of these expressions as our genes.

3.4.1 Priority functions

A priority function maps things to real numbers. An algorithm A can use a priority function P as a heuristics if P assigns a real number to each alternative of each choice that A wishes to delegate to the heuristics. A does this simply by for each such choice choosing the alternative to which P assigns the highest number among the alternatives of that choice.

As an example, a graph-coloring register allocator can use a priority function P as a heuristics for choosing what variable to spill when the allocation graph is not colorable, if P maps all variables occurring in the basic blocks of the control flow graph to real numbers.

3.4.2 Expression trees

The ASTs of the expressions defining priority functions have operators as internal nodes and environment parameters (also called “features”) and real number constants as leaves. The operator set should be rich enough to enable a large set of priority functions, but small enough keep the size of the search space reasonable. In the experiments, the operators used were roughly the standard C operators.

The values of environment parameters need to be supplied by other parts of the compiler or by a profiler. What parameters to use are highly dependent on the optimization problem. Currently, identifying the relevant parameters has to be done manually. Future work includes inventing algorithms for doing this automatically.

Continuing our register allocation example, the priority function may be defined by an expression as follows:

\[ P(x) = \sum_{b \in \text{blocks}} w_b \cdot \left( \text{LDsave} \cdot \text{uses}_{x,b} + \text{STsave} \cdot \text{defs}_{x,b} \right) / |x| \]

where
1. x is a variable,
2. |x| is the live range for x (that is, the set of basic blocks through which the value of x must be preserved),
3. b is a basic block,
4. \( w_b \) is the estimated execution frequency for block b,
5. \( \text{LDsave} \) and \( \text{STsave} \) are estimated of the execution time saved by keeping a variable in a register for references and definitions, respectively,
6. \( \text{uses}_{x,b} \) is the number of uses of variable x in block b, and
7. \( \text{defs}_{x,b} \) is the number of definitions (assignments) to variable x in block b.

This seems like a reasonable expression, but nevertheless this expression was replaced during GP search, probably suggesting that the at least one of the features involved (\( w_b \), \( \text{LDsave} \), \( \text{STsave} \), \ldots) cannot be estimated accurately enough.

Anyhow, we are now able to define mutation and crossover.

3.4.3 Mutation

An expression tree is mutated simply by randomly selecting a node in the tree and then replace the subtree rooted in that node with a randomly generated tree. The new subtree may be bigger or smaller than the old one. Hence there is no upper limit on how big expression trees may become. To avoid large expressions favor smaller expressions are favored by fitness evaluation when all other things are equal.

3.4.4 Crossover

Crossover is equally easy to define: To generate off-springs from two expression trees, we randomly select a node in each tree and swap the subtrees rooted in those nodes.

4. Experiments

The approach outlined above was tested on three back-end optimization algorithms for, respectively, hyperblock formation (HF), register allocation (RA) and data prefetching (DP; the problem of finding out where to insert prefetch instructions). For fitness evaluation, a subset of the application and input scenarios from several benchmarks (MediaBench, Spec95 and Spec2000) was used. However,
only about half of these was used for fitness evaluation during training (that is, during GP search). The heuristics found by was compared with the base-line heuristics by evaluating their fitness on the same applications and scenarios used during training. The average improvement factor is listed for each optimization problem in the row "training input" in the table below. The same comparison was then done by evaluating fitness using the input scenarios not used during training. The average speedups are listed in the row labeled "novel input". Finally, the comparison was made by evaluating fitness using the applications not used during training. The results are listed in the row labeled "cross-validation".

<table>
<thead>
<tr>
<th></th>
<th>HF</th>
<th>RA</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training input</td>
<td>1.44</td>
<td>1.03</td>
<td>1.31</td>
</tr>
<tr>
<td>Novel input</td>
<td>1.25</td>
<td>1.03</td>
<td>1.36</td>
</tr>
<tr>
<td>Cross-validation</td>
<td>1.09</td>
<td>1.02</td>
<td>1.01</td>
</tr>
</tbody>
</table>

As you can see, the improvements are significant. In fact, for HF and DP the base-line heuristics died very early, suggesting that good heuristics can be found from scratch by using GP. The result for RA may not look impressive. But considering that RA is a well-studied problem, it is impressive that GP search did manage to come up with an improvement. The somewhat disappointing cross-validation results suggests that the heuristics found were over-fitted to the particular applications used during training. Future work will have to address that problem.

5. Conclusion

The results from the experiments are encouraging enough to conclude that there is no reason why compiler writers should spend their time finding good heuristics manually.

Though the experiments was done with low-level back-end optimizations, nothing prevents us from using GP to find or improve heuristics for high-level optimization problems such as query optimization. Doing so would probably yield very impressive speedup factors.