MACHINE LEARNING FOR COMPILER OPTIMIZATION
Self-tuning or adaptive methods can be used to optimize compiler performance. In some cases performance improvement can be made over maximum optimization level i.e. –O3 or –fast. Objective: Generate a combination of compiler flags that will minimize runtime.
Example (faster than -O3)

NP-Complete

- Many program optimization problems are NP-Complete

- Implications
  - Assumed intractable
  - Exponential algorithms required to provide global optimization
  - Heuristics or approximation algorithms must be used
cTuning Framework

- cTuning is a large set of tools enabling adaptive tuning for GCC
- Open source
- Interesting project with many features
- Complex
- Stability issues
Iterative Compilation

- Over $2^{80}$ possible flag combinations for GCC
- In cTuning a random search algorithm is used to find good compiler optimization flag combinations
  - Each GCC flag included in combo with probability 0.5
  - Run 1000 iterations per program
  - Flag combination with best performance kept
- Iterative compilation provides speedup over GCC `-O3` but is slow
Machine Learning

- Learning from experience how to improve performance at accomplishing a set of tasks [MIT97]
- Goal of using ML is to predict good optimization flags for a new (unseen) program while reducing time required by iterative compilation [FKMC11]
- Classification of an unseen program is performed by a learning algorithm that compares the features of the new program to others in the training database and selecting the flags from the best match
Collective Tuning

- Supervised Learning
  - A training set is provided to the learning algorithm that includes inputs and known outputs or targets
  - Algorithm generalizes a function from inputs and targets in training set
  - Algorithm uses the function to determine output of previously unseen inputs
Collective Tuning

- Program features and transformations (flags) collected from iterative compilation of many programs form the training set
- Example features
  - ft20 = Conditional branch count
  - ft21 = Assignment instruction count
  - ft24 = Instruction count
- 56 features collected in total
- Stored in a collective database (training set)
Training

Classification

Adapted from: http://www.ctuning.org/wiki/index.php/CTools:CTuningCC
KNN

- K-Nearest Neighbor or KNN algorithm
  - Classification is based on majority rule of the k nearest neighbors
  - Algorithm
    - Calculate Euclidean distance between previously unseen feature vector and features of every training example and find k nearest neighbors
    - Assign output of the majority class among the k nearest neighbors
  - cTuning optimizations classified based on the nearest neighbor $k = 1$

[SEG07] [MAR09]
2D KNN Classification Example

- Assume square and triangle classes can represent two other programs
- $k = 1$ green query point assigned to red triangle class
- $k = 3$ triangles win so query point assigned to red triangle class
- $k = 5$ squares win so query point assigned to blue square class

Image Credit: Antti Ajanki
http://en.wikipedia.org/wiki/File:KnnClassification.svg
Creative Commons Creative Commons License Deed Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)
2D Nearest Neighbor Decision Surface

- $k=1$ or Nearest Neighbor
- Voronoi Diagram
- All query points falling in a polygon are closer to the training example point shown in the cell than any other
- Query point will be classified to the label of the training point for the cell

[MIT97]

Image Credit: Mysid (SVG), Cyp(original)
Creative Commons Creative Commons License Deed Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0)
Extension of cTuning

- New key, value based database format was developed using Python due to stability issues with cTuning database
- Two training databases created using new data gathered
  - Fastest runs of 10 cBench benchmark programs with the GCC compiler
  - Fastest runs of the same 10 cBench benchmarks along with 10 SPEC and PARSEC benchmark programs with the PGI compiler
More Extension of cTuning

- knn algorithm with feature normalization implemented in Python to work with databases
- Scripts to parse, create database, and perform leave one out cross validation
- Modifications to ctuning-cc (GCC) compiler to use Python knn algorithm
Research Conclusions

- Out of 10 GCC training programs iterative compilation reduced runtime for all over –O3
- Out of 20 PGI training programs iterative compilation reduced runtime for all but 1 over –fast
- Leave one out cross validation used
  - Leave out 1 program for testing and use remaining N-1 programs for training
  - Speedup results from using flags predicted by KNN on the following 3 slides
GCC Speedup Over -O3 with KNN
GCC Speedup Over –O3 with KNN and Normalized Features 1-36
PGI Speedup Over –fast with Normalized KNN
Future Research Opportunities

- Application of iterative compilation to targeted programs instead of using machine learning
- Exploration of feature space in an attempt to develop better methods/heuristics
- Identification of optimal scaling factors for training data for knn algorithm
- Plug-in new learning algorithms
References


