15-745 Class Project Proposal

Brendan Meeder, Jamie Morgenstern, Richard Peng

April 13, 2011

1 Group Info

Brendan Meeder (bmeeder@cs.cmu.edu)
Jamie Morgenstern (jamiemmt@cs.cmu.edu)
Richard Peng (richard.peng@gmail.com)

2 Project Webpage

`http://www.cs.cmu.edu/~bmeeder/15745/proposal.html`

3 Project Description

We propose implementing and benchmarking a classifier for ordering and selection of dependent LLVM code passes. Many optimizations we would like to make at compile-time are dependent upon the other optimizations we make during compilation. For example, if one performs dead code elimination before common subexpression elimination, one would expect the results to be quite different from performing the optimizations in the reverse order. We would like to analyze these dependencies: given the set of optimizations implemented within LLVM, which subsets are heavily dependent upon one another? Does it always make sense to perform these optimizations in a particular order, or does the ordering within these dependent bundles depend upon features of the code we're compiling? Does the ordering of the highly dependent bundles matter?

There are several components to such a project: dependent subsets of the optimizations must be chosen for analysis, determining and extracting relevant features of code for compilation, training a classifier to choose an ordering of passes to run on code given the features, and testing the classifier on new code.

First, we will need to choose which subsets of transformations in LLVM we will bundle together. A first cut will be to examine those sets that "look dependent" (e.g., one can imagine that `mem2reg` and `memcpyopt` will be dependent). A more ambitious goal would be to formally measure some notion of dependency between passes by benchmarking code comiled with different collections of optimization.

Next, our group will decide which features of code seem most relevant in determining which ordering of passes to run on the code. We expect iterature to be helpful in this search: papers which first implemented these passes often describe informally what properties of code will reap the most benefit from the transformation. Once we've decided which features will be useful in our classification, we will need to build a feature extractor to profile code and obtain these features (time spent in loops, average length of execution path, cache misses, etc).

Next, building and training a classifier (either based on logistic regression or clustering) or finding an off-the-shelf classifier to use with our feature extractor. We plan to extract features and benchmark based on either the SPEC benchmarks or some more realistic code.

Finally, we plan to test our classifier in one of two ways. Depending upon the necessary training data, we can either use cross validation (training on some 75% of the benchmarks and testing on the remaining 25%) or train on some other benchmarking code.

A 75% goal is to build a feature extractor, and generate a classifier which suggests orderings of optimizations which performs not too much worse than the "-O1" optimizations in general, and in certain instances outperforms them.

A 100% goal would be to successfully build a feature extractor and generate a classifier which beats the speedups that the "-O1" or "-O2" flags provide.

A 125% goal would be to look at JIT compilation, and do some analysis of which optimizations actually save time *when compilation time is a part of runtime.* We expect this to be difficult; and this is left as an additional component to the project if we finish early.

4 Logistics

4.1 Plan of Attack and Schedule

In the next week, we will work out which subsets of optimizations we will consider for the purposes of this project, and decide which features will be relevant for profiling code with respect to these passes. By 3/23, we will have a plan of attack for how we will collect the features from code via profiling.

For the next week (the last week in March), we will write the feature extractor and begin extracting features from the code we plan to use for training and testing our classifier.

The first week in April will have us running the feature extracting, making modifications as necessary to the way we are collecting the features if the first plan was too costly with respect to compile-time.

The second week in April, we will write the classifier and train it on the collected feature data. This is where we hope to be for the midway milestone.

The remaining time will be for catching up (if any of the above takes longer than expected), testing the classifier, writing the paper, and optionally considering JIT tradeoffs as an extension.

4.2 Milestone

By April 17th, we would like to have our features extracted from all the training and testing data (the SPEC2000 benchmarks). This will require the following to have been completed:

1. Select dependent passes
2. Select relevent features regarding these passes
3. Design profiling passes to collect these features
4. Profile or training and test code

If this is successful, we will be training our classifier either just before or just after the deadline for the milestone report.

4.3 Literature Search

[1] addressed a similar problem of finding the set of passes that optimizes code performance. The related problem of reducing compilation time by selecting a useful set of passes was the main focus of [3]. The list of features listed on page 22 of [3] will also act as a starting point for our initial passes for identifying features of the program. However, techniques such as the ones used in [2] to automatically finding features might also be worth exploring.

4.4 Resources Needed

1. Access to Matlab for running machine learning algorithms.
2. Compiler benchmark suites. SPEC is the default one we will consider. We are also planning on finding additional benchmarks or libraries (BLAS and LAPACK, for example) that we can write a wrapper around and make performance measurements.
3. Additional machines. Brendan has access to some of the cluster machines in GHC, so we can run non-hardware based measurements (cache hit-rate in a simulator, for example) on all of these machines. Additionally, we can pick a set of machines that have the same specifications and run experiments on them when other users are not using the machine.

4.5 Getting Started

We have looked into the two papers mentioned in the literature search. Additionally, we have all taken the machine learning course and feel as though this gives us adaquate preparation for setting up a good ML experiment, understanding the ML tools available to us, and interpreting the results.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *In Proceedings of the International Symposium on Code Generation and Optimization (CGO*, pages 295–305, 2006.

[2] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 81–91, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Gennady Pekhimenko. Machine learning algorithms for choosing compiler heuristics, 2008.