Clustering for optimizing compiler optimization parameters

Siddhartha Jain sj1@cs.cmu.edu Christian Kroer ckroer@cs.cmu.edu

May 1, 2014

Abstract

Modern compilers have a large number of optimization parameters. Machine learning has proven a robust and effective tool for selecting a good set of optimizations for a particular program. State of the art methods characterize a program using fixed-length feature vectors of either the source code or performance counters extracted when running the program. A relatively new approach based on graph kernels tries to characterize programs based on their control flow graphs circumventing the need to come up with an explicit set of features. All of the above however, require developing a performance model to predict the runtime performance of the program based on its characterization. This is problematic as peformance models are usually too simple and fail to take into account the complexity of today's processors (CPU, cache, etc.) and memory. In addition, the program runtime can be a highly complicated function of the program's characterization, which may not be adequately captured by the regression model being used. In this project, we investigate methods based on clustering similar functions, and generating optimized parameters for each cluster. New functions are assigned to appropriate clusters and the tuned set of parameters are applied to the instance. This approach has been shown to outperform prediction-based approaches in the combinatorial optimization community. We rely on a recent parameter optimization technique called ParamILS, originating in the SAT community, which has been shown to increase performance in a large range of application areas. Experiments on benchmark instances show this approach to be effective at finding a good set of instance-specific compiler optimization parameters.

Keywords compiler optimization, graph kernels, machine learning, support vector machine

1 Introduction

Modern compilers have many parameters, which leads to a large number of possible optimization configurations. The right optimization configuration can signficantly speedup the program [Pouchet et al., 2007, 2010, 2008]. While one

can try and handtune the parameters for a program, doing so would involve considerable analysis of the source code and performance characteristics of that program. It would be an impossible task to do it in a quick and efficient manner for a new program. What we need is an *automated* parameter tuner. There has been considerable research in machine learning algorithms to tune parameters for compilers automatically. Well engineered machine learning algorithms have had great success in optimizing the parameters and outperforming the most aggressive optimization options in commercial and open-source compilers [Agakov et al., 2006, Cavazos et al., 2007, Dubach et al., 2007, Scardovi and Sepulchre, 2006]. A machine learning algorithm however requires one to build a model to predict the runtime of the program given the optimization configuration. The running time of a program however can be a very complex nonlinear function of the program and the target hardware including the CPU, cache size, RAM speed, etc. Building such a model is extremely hard. To make matters worse, different optimization strategies may have different objectives. For instance, thread level parallelism SIMD-level parallelism in addition to possibly degrading data locality and increasing memory footprint [Park et al., 2013]. Thus an automated parameter tuner that does not rely on building a predictive model for runtime would be very practical.

There has been considerable work on automated parameter tuning for Boolean Satisfiability (SAT) solvers. Initially, the SAT community also relied on building a predictive model for the runtime of a SAT solver given a value for the optimization parameters (SATzilla [Nudelman et al., 2004]). Recently, the most competitive SAT solvers have relied on clustering. In this approach, the training data is split up into clusters, using some appropriate clustering technique. For each cluster, the best algorithm for the cluster is then selected. The algorithm can be selected from a set of different algorithms, one configurable algorithm where a tuned set of parameters is found, or a mixture thereof. Once new instances are encountered, the instance is assigned to the appropriate cluster, and the best algorithm (or algorithm configuration) is then applied. Thus, the algorithm applied to new instances has been tuned for a set of similar instances, and thereby achieves (somewhat) instance-specific optimized algorithm choices [Kadioglu et al., 2010, Leyton-Brown et al., 2014].

Briefly, this approach, as applied to tuning compilers, accepts a set of clusters of functions and tries to find the optimal set of parameters for each cluster. Then whenever we get a new program instance, we use the k-nearest-neighbors algorithm on each of its functions and assign them to their respective clusters. We then apply the optimization parameters – previously found for a cluster using ParamILS – to the respective function as we compile each method with a *different* set of optimization parameters. For clustering the programs, we use a graph kernel called the random walk kernel to come up with a kernel or distance matrix between each pair of functions. We then use hierarchical clustering to cluster the functions into different clusters. We call this approach *instance-specific compiler configuration (ISCC)*

In this project, we apply the framework to tune the optimization parameters of a compiler and show that we're able to achieve considerably better perfor-

mance compared to the baseline. The rest of the paper is organized as follows. We describe the related work – (1) a popular runtime prediction model for compiler optimizations (2) and go into detail on how ParamILS works. We then explain how ParamILS can be applied to tune compiler optimization parameters. We experimentally demonstrate the usefulness of our approach and then finally conclude.

2 Related work

In the area of automated compiler configuration, the most related work is based around predicting the performance improvement of various compiler optimization combinations [Park et al., 2013, 2012]. They build predictive models, that try to predict the speedup obtained from applying a given set of parameters to some file being compiled. This is done by using various program characterization approaches, such as static source code features or graph kernels computed on the control-flow graph. They then proceed to show that their graph kernel-based approach performs better than the static code features-based approach that they test against.

Machine learning for algorithm selection has received a lot of attention in the combinatorial optimization literature. In recent years, portfolio algorithms, where a set of algorithms are automatically chosen from at runtime, have dominated the SAT solving competition [Leyton-Brown et al., 2014]. An initially succesful algorithm, Satzilla, was a regression-based approach. In this approach, a portfolio of algorithms is selected, and a runtime prediction model is built for each algorithm, using a set of features from SAT instances. When a new instance arrives, the runtime of each solver is predicted, and the one with the lowest predicted runtime is used to solve the instance. In later work, clustering techniques were investigated. In the instance-specific algorithm configuration approach, a set of clusters are computed based on features of the instance, and a solver is then tuned for each cluster [Kadioglu et al., 2010]. When new instances arrives, it is assigned to the nearest cluster, and the algorithm tuned for the cluster is selected.

Approaches like the one introduced by Kadioglu et al. [2010] rely on an automated approach for configuring algorithms for each cluster. Several approaches have been introduced by this, based on genetic algorithms [Ansótegui et al., 2009] and local search [Hutter et al., 2009]. In this paper, we use paramILS, a local-search based approach.

3 ISCC

Our idea consists of the following parts :- (1) come up with the pairwise distances between every function in our training set (2) cluster the training functions using the distance matrix using agglomerative hierarchical clustering (3) use ParamILS to find the optimal set of parameters for each cluster (4) use k-

nearest-neighbors algorithm to classify each function of a new program into a cluster and apply the optimal optimization parameters we've found for that cluster onto that function. We give a more detailed description later.

Random walk kernel

A crucial component of our algorithm is step 1 where we have to generate pairwise distances between every function in our training set. One possible approach is to map the program into Euclidean space by characterizing it with a fixed length feature vector. The euclidean distances can then form the distance matrix. We do not take this approach however as it is hard to determine, without having run the program, a good set of features to characterize the program with. Instead we decide to go with a recent machine learning notion called a random walk kernel [Vishwanathan et al., 2010].

Roughly speaking, a random walk kernel computes the number of common walks between two graphs. The nodes of the graphs can have labels in which case a two walks are considered the same only if the labels of the nodes in the walk are teh same. A convenient way to compute the number of common walks is to compute the number of walks in the product graph of two graphs.

Let $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$, be two graphs. Then the product graph is $G_\times = (V_\times, E_\times)$ where

$$\begin{aligned} V_\times(G_1 \times G_2) &= \{(v_1, w_1) \in (V_1 \times V_2) \\ &\quad : label(v_1) = label(w_1)\} \\ E_\times(G_1 \times G_2) &= \{((v_1, w_1), (v_2, w_2)) \in V^2(G_1 \times G_2) \\ &\quad : (v_1, v_2) \in E_1 \wedge (w_1, w_2) \in E_2\} \end{aligned}$$

If A is the adjacency matrix, then $A^k(i, j)$ means that k walks exist between vertices i and j. Then the random walk kernel computes

$$K(G_1, G_2) = \sum_{i,j}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda^n A_\times^n]_{ij}$$

We use the random walk kernel as a measure of similarity, ρ between two graphs. A higher kernel value means that the two graphs have a lot of common walks. However, a larger graph would naturally also have more walks giving an artificially high value for the kernel. Thus we normalize the kernel of the two graphs as follows

$$\rho(G_1, G_2) = \frac{K(G_1, G_2)}{\sqrt{K(G_1, G_1), K(G_2, G_2)}}$$

When $G_1 = G_2$, $\rho = 1$ and otherwise it's < 1.

Hierarchical clustering

For step (2), we use agglomerative hierarchical clustering (AHC) to cluster functions together based on a distance matrix. AHC starts by putting every function into its own cluster. Then we compute a measure for each pair of clusters and merge the clusters with the lowest value for that measure.

The measure itself can be one of a variety of different functions. A popular measure is the complete-linkage measure which computes the distance between the two furthest apart nodes present in the two different clusters. That is the measure we use for our project.

paramILS

For step (3), we use the parameter tuning algorithm paramILS [Hutter et al., 2009]. ParamILS is given some configurable algorithm, in the form of an executable, along with a search space of parameters with various options available (e.g. whether to perform tiling, or choosing how much loop unrolling to perform). It also needs training data, which it will attempt to find the optimal set of parameters for. In our case, this is the set of functions belonging to some given cluster. It then proceeds to perform a local search over the space of parameters, evaluating each choice of parameters by running it on some or all of the training data. Various heuristics are applied to speed up the search, such as adaptively limiting the amount of time spent evaluating each set of parameters.

The algorithm

1. We extract the individual methods in each program and write it to a different file using one of our compiler passes.

2. Let the set of functions extracted from the files be $\mathcal{F}$. Then for every $f \in \mathcal{F}$, we compute the basic block control flow graph using llvm. We then use another one of our compiler passes to convert the basic block control flow graph into an instruction control flow graph.

3. We then compute the similarity ρ between two graphs as described above. We then take $\frac{1}{\rho}$ as the distance between two graphs and use agglomerative hierarchical clustering to cluster the program instances.

4. Then we individually feed each cluster into ParamILS which returns an optimal (possibly locally optimal) set of parameters for that cluster.

5. Once we've obtained an optimal set of compiler parameters for each cluster, when we receive a new program, we split it into its constitutent functions and assign each function to a cluster based on the similarity measure we used before. In particular, if the cluster $C = \{G_1, \ldots, G_n\}$ where G_i are control flow graphs belonging to the cluster and G' is the control flow

graph of the new function, then we compute the similarity as

$$\rho(G', C) = \sum_{i=1}^{n} \rho(G_i, G')$$

We assign G' to the cluster for which $\rho(G', C)$ is maximized and use the parameters found for that cluster to compile the function corresponding to G'.

4 Experimental Setup

We evaluated our optimization framework on a server with four quad-core Intel Xeon 3.4 GHz CPUs, with 16 MB cache. For our compiler optimization search space, we use the PoCC source to source compiler. It compiles from C code to C code. For compilation to machine language, we use GCC version 4.6.3, with the `-O3` flag.

To test our framework, we use the Polybench v3.2 benchmark suite, consisting of 30 C programs representative of various scientific computations.

To build our model, we use LLVM for generating the annotated control-flow graph. We use code from Vishwanathan et al. [2010] to generate graph kernels, and use Matlab's agglomerative clustering to generate our clusters. We use paramILS to optimize the parameters for each cluster.

We split the Polybench suite into training and test data sets. The files "3mm.c", "2mm.c", "dynprog.c", "syrk.c", and "gemver.c" are kept as test data, while the remaining 26 files are used as training data. Each file is split into a set of separate files, one for each method in the file.

For training, the specific function at hand is compiled using PoCC with the chosen set of optimization parameters, while all other functions belonging to the same file are not compiled by PoCC. All functions belonging to the file are then compiled with GCC, and the performance is evaluated as $\frac{P}{B}$, where P is the runtime of the program with PoCC applied to the function, and B is the baseline runtime, when all functions are compiled directly with GCC, without using PoCC. ParamILS is given a total runtime cap of 1200 seconds for tuning parameters for each cluster. The parameter space that we consider is shown in Figure 1.

We then test the performance of our approach on each test instance, by evaluating the improvement in performance over the baseline approach. For a given test file, we assign each function in the file to its appropriate cluster. Then, PoCC is applied to each function, with the parameters chosen according to the cluster that the function belongs to. Finally, the PoCC output for each function is compiled into an executable with GCC, and the runtime improvement of the executable, over baseline, is used as evaluation.

Loop fusion	(on, off)
Loop tiling	(normal, l2tile, off)
Loop unrolling factor	(0,2,4,8)
Prevectorization	(on, off)
Parallelization	(on, off)
Multipiping	(on, off)
Rar dependency analysis	(on, off)
Lastwriter dependency simplification	(on, off)
Scalar privatization	(on, off)

Figure 1: The parameter space that we search over.

5 Experimental Evaluation

In this section we describe the results that we achieved from the setup described above. The results are given in 2. As can be seen, using only a single cluster, i.e. tuning the parameters for all the instances at once, yields largely no performance gain over just applying GCC with `-O3` directly. We get small speed improvements on the instances "3mm.c" and "dynprog.c", but suffer comparable performance loss on "2mm.c".

When we use two clusters instead, as shown in 2, we see better performance. Two instances have negligible performance deterioration, one instance has negligible performance improvement, while the "3mm.c" and "dynprog.c" have considerable performance improvements of approximately 5 and 2 percent, respectively.

While we see some amount of performance increase using our approach, once we start clustering, it is still very little performance gain. In contrast with this, Park et al. [2012] show much more significant performance improvements using their approach. This is somewhat in constrast with the state-of-the-art in combinatorial optimization solving, where their approach is more akin to early approaches that did not perform as well as clustering-based schemes. We will now try to give some comparisons and weaknesses of our work, to shed some light on this difference. One large issue with our approach is that it requires significant quantities of training data. A rule of thumb given in the combinatorial optimizaiton community is that each cluster should have at least 30 instances. The Polybench suite contains 30 files, each with 4 functions, giving us 120 instances. Since we are looking to compute $2 - 4$ clusters, this would seemingly be sufficcient. However, we learned through our experiments that PoCC does not apply any optimizations to three out of four functions in each file, since they are auxiliary functions related to printing and benchmarking. Thus, we are effectively left with 30 instances that are sensible for tuning purposes. Further, $2 - 4$ clusters is quite low, and it could easily be the case that a higher number of clusters, with enough training data, would perform much better.

Parameter tuning is also a process with extremely high runtime. ParamILS performs a local search over the parameters space, and for each candidate set

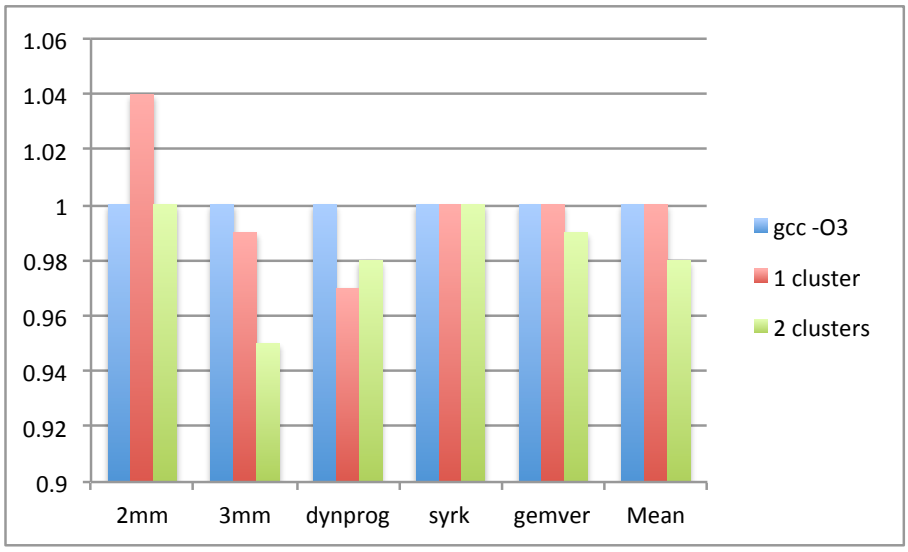

Figure 2: We give results for putting all functions in 1 cluster and 2 clusters, GCC shows baseline. We present results for the 5 test functions and the mean of the results. Results are normalized to improvement over baseline (GCC with `-O3`), so lower is better. As shown, we consistently get an improvement for 2 clusters over the baseline.

of parameters, it must perform several performance runs. For some files in Polybench, a single run of the program generated from the file takes about a minute, depending on the specific parameters given, a whole 5% of the runtime devoted to the whole parameter tuning process. We initially performed our experiments with a 600 second time cap. Doubling this time cap to the one presented in this paper gave close to a 10% improvement for several instances. Thus, simply increasing runtime of the algorithm could lead to better results. We were unable to do this due to time constraints for finishing the project. Based on our prior experience with clustering and parameter tuning, a whole week is ideally devoted to tuning parameters, to generate very high quality parameters.

Another issue is that both Park et al. [2012] and Park et al. [2013] give very vague and ambiguous descriptions of their experimental setup and machine learning model. Thus, it is hard to draw on their results, and know whether we are truly performing our clustering-based approach on the right parameter space, that will allow us to compare our approach with theirs on an even footing.

6 Conclusions and Future Work

We developed an algorithm, ISCC, for automatically tuning compiler optimization parameters in an instance-specific fashion, without relying on constructing

a predictive model for runtime performance of programs. Our approach was inspired by a similar framework which was developed by the SAT community called ISAC which clusters instances (in our cases specific functions of programs) into a set of clusters, finds the optimal parameters for each cluster, and then for a new instance, applies the parameters of the cluster "closest" to the new instance – where closest is defined by a distance matrix. ISAC itself uses a parameter tuner called ParamILS which uses local search to find an optimal set of parameters for a cluster of instances.

We evaluated our algorithm on the PolyBench test suite and showed significant improvements over the baselines of using just (1) gcc with the `-O3` option (2) putting everything into one cluster. One problem we ran into was that our test suite turned out to be too small to be able to do more effective training.

Future work would involve testing out ISCC on much larger test suites. In this project, our main focus was to show the viability of a non-runtime prediction model based approach to tune compiler parameters. However, another important direction to explore would be more sophisticated distance functions to cluster functions. In particular, it might be worth trying to characterize programs using features to map them to Euclidean space and then use a standard clustering algorithm like k-means clustering. This is the approach used by Kadioglu et al. [2010] to get state-of-the-art performance on SAT solving benchmarks. While Park et al. [2012] show that graph kernel based approaches are superior to static code features, it is not clear that their feature-based model is the correct model to use for this scenario, especially in light of the success of feature-based clustering schemes for combinatorial optimization.

7 Distribution of Credit

We both did equal amounts of work on the project. `sj1` setup the graph kernel algorithm for computing the distance matrix for the graphs. while `ckroer` helped with the clustering. Both `sj1` and `ckroer` wrote a compiler pass each for splitting the file into different methods and converting the basic block control flow graph into an instruction control flow graph. Both worked together on setting up the ParamILS framework for tuning parameters.

References

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society, 2006.

C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming-CP 2009*, pages 142–157. Springer, 2009.

J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 185–197. IEEE, 2007.

C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *Proceedings of the 4th international conference on Computing frontiers*, pages 131–142. ACM, 2007.

F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. Isac-instance-specific algorithm configuration. In *ECAI*, volume 215, pages 751–756, 2010.

K. Leyton-Brown, H. H. Hoos, F. Hutter, and L. Xu. Understanding the empirical hardness of np-complete problems. *Communications of the ACM*, 57 (5):98–107, 2014.

E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. Hoos. Satzilla: An algorithm portfolio for sat. *Solver description, SAT competition*, 2004, 2004.

E. Park, J. Cavazos, and M. A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 196–206. ACM, 2012.

E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 144–156. IEEE, 2007.

L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM, 2008.

L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

L. Scardovi and R. Sepulchre. Collective optimization over average quantities. In *Decision and Control, 2006 45th IEEE Conference on*, pages 3369–3374. IEEE, 2006.

S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.