Phase Ordering of Compiler Optimizations Using ML with Static and Dynamic Features

Ankit Goyal
University of Texas at Austin
ankit3goyal@gmail.com

Vivek Natarajan
University of Texas at Austin
natviv@gmail.com

Abstract

Modern day compilers implement a number of optimizations. However, it is important to choose the right set of compiler optimizations and apply them in the right order as this can have a significant impact on program performance. This is often referred to as the phase ordering problem of compiler optimizations in literature. However the optimizations interact in a complex manner with both the code and all other optimizations. Moreover the number of available optimization is also very high, hence the search space for the optimal optimization sequence is vast. Traditionally, compilers used pre-determined and generalized heuristics and apply the same set of optimizations in a fixed order regardless of the features of the program being optimized. In this project, we use an ensemble learning model based on a combination of performance counters, static code features and properties of the control flow graph learnt offline to predict a good set of compiler optimization sequences. We evaluate our learned model on a set of automatically generated programs using CSmith, a tool for generating random C programs that statically and dynamically conform to the C99 standard. Using our method, we are able to demonstrate that a combination of static code features and performance counters performs better than the use of only static or performance counters for training the model. We obtain an average speedup of 17.32% over the default order in the -O3 setting of the Clang compiler on our test set of automatically generated programs.

Keywords Machine Learning, Compilers, Hardware Counters, Optimizations

http://dx.doi.org/10.1145/

1. Introduction

The problem of determining the best set of optimizations for a particular program and their sequence, also known as the phase ordering, has been an open problem in compiler research. The problem is difficult because of two reasons. Firstly, modern day compiler implement a large number of optimization passes. For example, GCC has around 250 optimization passes and LLVM has at least 125 optimization passes implemented. However, because these optimizations transform the code in a non-linear manner, they could potentially alter the impact of other optimizations to be applied later. A classic example of this is the phase ordering problem between instruction scheduling and register allocation. Register allocation tries to make the optimum use of the available registers and prevent their spilling to the memory while instruction scheduling tries to reduce stalls by using more registers. This sort of phase ordering problem can arise between any pair of optimizations in a sequence. Hence it is imperative to select and schedule optimizations in a manner that their interactions are compatible and improve program performance. However, selecting the best sequence is highly non-trivial because the optimum solution may be located anywhere in this exponentially vast search space and the problem is thus NP-Hard.

The standard practice in compilers is to use a fixed set of optimization passes (with a fixed sequence) corresponding to different levels of optimizations based on heuristics tuned on a fixed set of benchmark programs. Most of the optimization passes in GCC or LLVM are typically turned off and it is left to programmer to decide when and which set of optimizations to apply. However, it is obvious that the best set and ordering of optimizations varies from program to program. Traditional methods typically involve iterative compilation [9] or the use of genetic algorithms [5] to solve the phase ordering problem and efficiently scan the vast search space. Iterative compilation typically involves iteratively running a set of optimizations on the program and using the performance to decide on a new optimization setting. Genetic algorithms were used by Cooper et al. [5] but involved the added constraint of optimizing for code size. Although they were able to reduce code size by as much

as 40%, the main bottleneck is they had to train the genetic algorithm for every program. These pure search techniques do use any apriori knowledge of the target machine, compiler or code under optimization instead iteratively evaluate and tune the heuristics to identify the best optimization setting. Cavazos et al. [4] suggest that this could take up to 600 evaluations of the program to zone in on the right optimization settings. In this project, we use a combination of performance counters, static code features and features of the control flow graph to learn a model offline and use it to predict the best optimization setting for the program under consideration. The performance counters are obtained from only three runs of a set of training programs. To the best of our knowledge, we are the first to use features of the control flow graph as static code features and also the first to use a combination of static code features and performance counters to tackle the phase ordering problem. Our results demonstrate that using the combination provides a significant improvement than using only one of static code features or performance counters as has been done in previous research.

In recent times, Machine learning has received a lot of attention in problems pertaining to the selection of optimizations [2] or even in cases like selection of the loop unroll factor [8]. Static code features were used by Agakov et al. [3] to identify a set of good optimizations to a program under test with the underlying assumption that optimizations which perform well on programs with similar static characteristics would perform well on the new program. However, static code features fail to model global characteristics of a program and fail when applied to large application programs especially those that are control-flow intensive. Cavazos et al. [4] came up with a solution to this issue by using performance counters which have been extensively used in performance analysis of programs. However, they do not address the issue of combining static and performance counters as features to train the model. In our opinion, the two provide orthogonal characterizations of the program and should complement each other well and our results substantiate our intuition. Moreover, our use of features which explicitly characterize the control flow graph of the program under consideration in combination with the static code features gives performance on par with the use of performance counters. This suggests that the features characterizing the control flow graph could be used as a useful substitute for the performance counters especially in the context of a static compiler.

2. Motivation

Previous work by Agakov et al. [1] have demonstrated the use of static program features to characterize a program. Moreover the authors used a set of 36 features and demonstrated by using a Principal Component Analysis technique that the 99% of the variance captured by this set could be captured by using a smaller subset of 5 features. Hence, for our experiments we chose a set of 9 program features which we felt best characterized our programs and could have a maximum impact on the optimization passes considered. The use of performance counters is motivated by Cavazos et al.[**?**] demonstrated that performance counters could prove effective as features. However, the problem they solve is not the phase ordering problem. They only attempt to predict what optimization passes should be enabled for any program based on a given subset. We also analyzed the control flow graph of the program to extract features corresponding to its shape for every function because a number of program optimizations indirectly depend on the shape of the Control flow graph like dominance relationships.

Table 1. Set of optimizations used for phase order optimizations with their description

LLVM Pass Name	Pass Description
-dse	Dead Store Elimination
-loop-unroll	Unroll Loops
-loop-unswitch	Unswitch loops
-licm	Loop Invariant Code Motion
-memcpyopt	MemCpy Optimization
-loop-rotate	Rotate Loops
-inline	Function Integration/Inlining
-reassociate	Reassociate Expressions
-sccp	Sparse Cond. Constant Propagation

3. Methodology

In this section, we describe our methodology for selecting the optimization passes considered for the phase ordering problem, the features used to characterize the program, construction of the training set and the learning model to predict the best optimization sequences for a new test program.

3.1 Selection of the Optimization passes considered for the phase ordering problem:

We considered the optimization passes listed in Table 1, (available in LLVM) for the phase ordering optimization. Most of the previous research do not really explain their reasons for selecting the optimization passes. In this project, we chose a set of passes which could potentially have a significant impact on the static code features and performance counters. Passes like function inline and reassociate have a strong correlation with the CFG and static code features. On the other hand optimizations like loop invariant code motion and loop-unroll have a direct impact on cache utilization and/or in the number of memory operations. It should be noted that majority of passes impact both static and dynamic features. As demonstrated previously, the ordering of the passes does have a significant impact on the performance of the program.

3.2 Feature selection

Feature selection has been an important topic of research in Machine learning and several heuristics have been proposed. The most important criteria is to select features that best characterize the variance in the data under consideration and are sufficiently discriminative to learn a good learning model. Previous work on this problem have either used static code features (Agakov et al. [1]) or performance counters (Cavazos et al. [4]) . In this project, we consider the combination of both static code features and performance counters for each program.

3.2.1 Static Features

Since a good number of our optimization passes deal with loops, we have selected on program features related to loops in the code. We have a taken a subset of features used by Stephenson et al [9] as shown in Table 2. As stated previously, a small set of static features is often sufficient to capture the variance in the set of programs under consideration and we decided to focus on the small set of features listed in Table 2. An LLVM pass to compute these features and we used the existing loop analysis pass provided by LLVM for some of the features. The features computed were averaged out if they were local in nature (like pertaining to individual loops) and normalize it by the number of instructions in the program for consistency among different programs.

3.2.2 Features obtained from the Control Flow Graph

We used the LLVM -dot-cfg pass to obtain the graphs representing the Control Flow Graph (CFG) corresponding to each program in the program under consideration. We used the NetworkX and PyGraphViz library implemented in python to analyze the CFG for every function in the program under consideration and obtained features corresponding to the number of nodes, number of nodes, the average degree, the diameter, the clustering co-efficient, the average path length and the graph density - features which we believe effectively characterize the shape of the CFG. We aggregate these values for the CFG corresponding to each function in the program and take the average value as features in addition to the static features mentioned previously.

3.2.3 Performance Counters

To measure performance counters, most processors today have a special set of registers that can do this without any side effects on the execution of the program. These counters typically describe dynamic program characteristics like cache hits, misses and branch prediction accuracies. Performance counters were obtained using PAPI Hardware Counters library API. These counters are hardware specific and support for all the counters is not necessarily available on all the hardwares. We used the Stampede machine on TACC (2/8 Xeon E5-2680 2.7GHz) for all our experiments. A total of 25 performance counters were available on stampede based on the hardware and counter types; 8-10 hardware counters can be calculated in a single run. So we had to run each program 3 times to get all 25 counters. All the performance counters obtained were normalized by the total number of instructions in the program in order to allow for generalization regardless of the length of the program. The goal is to use the statistics obtained from static program features and dynamic performance as features to learn a good model for automatically predicting a good sequence of optimizations.

Table 2. Static features used (subset of features used in [6])

Acronym	Description
LoopInstNum	Num. of instructions in a loop
LoopNum	Num. of loops
LoopBasicBlocksNum	Num. of basic blocks in a loop
LoopDepth	Average loop depth
LoopBranchInstNum	Num. of branching instructions
LoopLoadInst	Num. of Load instructions
LoopStoreInst	Num. of store instructions
LoopNumOperands	Num. of operands in loop
LoopNumMemInst	Num. of memory inst. in loop
LoopNumOperations	Num. of operations in loop

3.3 Generation of the set of best optimization sequences

In order to generate the training data, we considered all permutation of the optimization passes mentioned previously and obtained the execution times for our set of 27 different automatically generated programs. To do this efficiently, we divided the permutation into 4 and used OpenMP to parallelize the process of getting the execution times. All the programs were run on Stampede. We then sort the optimization sequences to obtain a set of 100 best optimization sequences and probabilistically compute a set of 5 best optimization sequences corresponding to each program.

3.4 Ensemble Classifier

We used the Exemplar SVM and Random Forest classifier as an ensemble classifier to predict a subset of optimization sequences for each program under test. Exemplar SVMs is a recently developed remodel of the conventional SVM classifier primarily tuned for meta-data transfer. It can be thought of as a combination of a discriminative and distance metric based Nearest Neighbor like classifier. The model is obtained by training a one vs all Linear SVM classifier for each positive instance in the training set consider vs a large set of negatives. In our case, our set of negatives is all the features corresponding to all the other programs bar the features corresponding to the program for which the model is being trained. Thus we train N different SVM classifiers and use the the predicted output of the classifier with the highest confidence. We also a train a separate random forest classifier and use it in an ensemble with the Exemplar SVM to obtain a prediction of the best optimization sequence

Performance Counter Name	Performance Counter Meaning
TOT_INS	Total Number of Instructions
TOT_CYC	Total Cycles
STL_ICY	Cycles with no instruction issue
FP_INS, FP_OPS	Floating point instructions, operations
BR_INS, BR_MSP, BR_TKN	Branch Instruction, Cond. Branches Mispredicted, Cond. Branches Taken
DCM, DCA, DCR, DCH, DCW	L1, L2 Data Cache: Accesses, Misses, Reads, Hits, Writes
LDM, STM	L1, L2 Cache: Load Misses, Store Misses
TCM, TCA	L1, L2 Total Cache Misses, Accesses
ICM, ICA	Instruction Cache: Misses, Accesses
TLB_DM	Data translation lookup buffer misses
TLB_IM	Instruction translation lookup buffer misses

Table 3. Performance counters used as features. The performance counter name is the acronym used for the counters by PAPI library

Table 4. Speedup achieved in different cases.

Method	Avg. Speedup
Static + CFG + Dynamic	1.173250279
Static + CFG	1.107801959
Dynamic	1.102995532

for the program under test. We do a leave one out cross validation on all programs in training set and report our results. The classifiers were implemented using the scikit learn package in Python.

4. Results

We trained the classifier using three different kinds of features: first - we considered both static and dynamic features. Second - we consider only static features. Third - we consider only dynamic features. As can be seen in table 4. that we received a significant speedup in all three cases however speedup achieved is much greater in static + dynamic case than either the only static or only dynamic case.

4.1 Dynamic Features

In this case only dynamic features are considered. Figure 1 shows average percentage improvement in execution time after applying the optimization passes in the predicted order over the default order of optimizations. It can be seen that in some cases the improvement is as high as 24%. Average improvement in execution time over all programs is about 9.07% which is very close to the 10% speedup reported by Cavazos et al. [4] albeit on different benchmark programs.

4.2 Static Features

In this case we only consider static features including CFG features. Figure 2. shows that in this case also we received a significant improvement in the execution time (as high as 27%) of predicted order of optimization over the default order which is consistent with the results from paper Agakov et al.[1] Average improvement in execution time received in this case is 9.24%

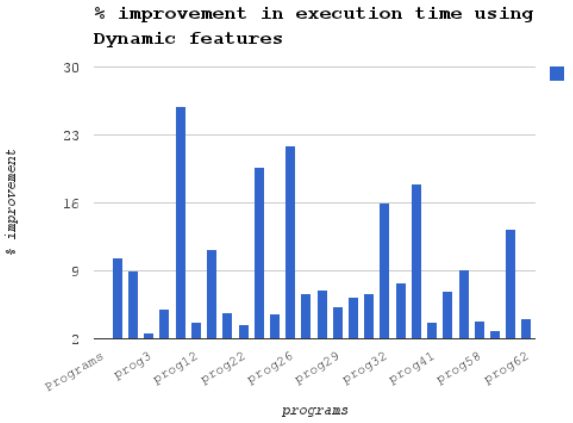

Figure 1. Improvement in execution time using dynamic features

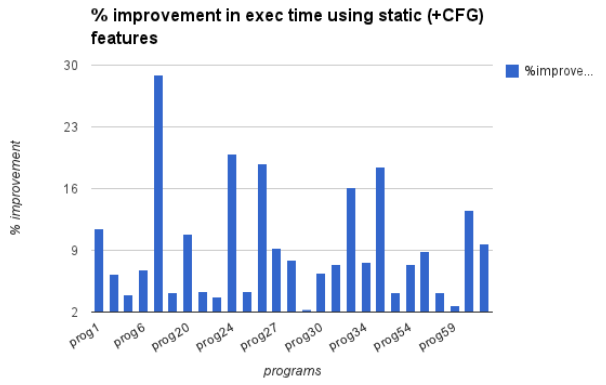

Figure 2. Improvement in execution time using static features

4.3 Dynamic and Static Features

According to the best of our knowledge our work is the first to consider both static program features and performance counters to characterize programs. As shown in Fig. 3 we were able to achieve much higher (upto 46%) in some cases or at least the same amount of speedup in almost all cases. This results shows that some combination of both static and dynamic features will result in close to optimal performance.

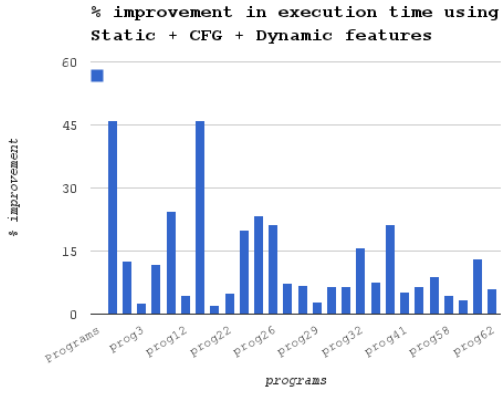

Figure 3. Improvement in execution time using static and dynamic features

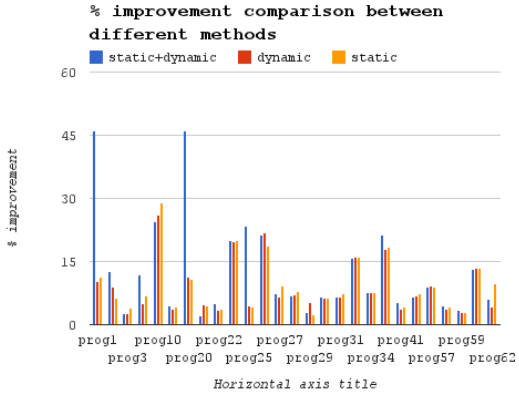

Figure 4. Comparison between improvement using different methods

4.4 Impact of static features on phase order prediction

The effect of same optimizations in different order can be seen in Figure 5 where predicted order has reduced the number of operations, memory instructions in the given program which directly impacts the execution time of the program.

We also observe variations in the CFG features obtained after applying the default order of optimizations and our predicted order. However, the variations are something which is consistent with our intuitions regarding compiler optimizations. However, their use is justified in the fact by the performance improvements obtained by the use of CFG features in addition to the traditional static features and performance counters.

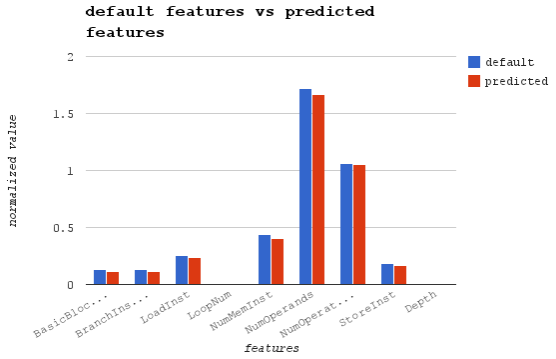

Figure 5. Default phase order static features vs predicted phase order static features

4.5 Impact of dynamic features on phase order prediction

Figure 6. shows that our predicted order was able to improve cache utilization over the default optimization order used in -O3 flag of Clang for a program (prog20) in which we saw improvement of 46.09%. Level 1 store misses reduced by 12.4% while Level 2 data misses reduced by 10.3%. It also reduced the TLB cache data misses by 21%. It did worse on some other performance counters related to the the cache like TLB instruction misses increased by 16%. However, we could generalize that it improved the cache performance over the default phase order.

It is evident from the results that both static features and dynamic features are crucial in predicting the performance. Since we need to train our model just once, use of dynamic features is practical. All that is required is to execute the training programs on a new target machine to learn the model and using the model we can predict suitable optimization order for other unseen test programs without any sort of iterative execution.

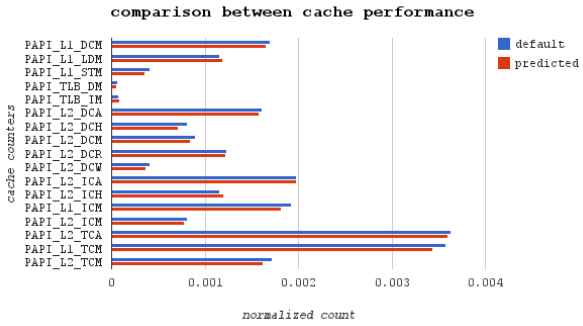

Figure 6. Cache performance of default order vs predicted order

5. Related Work

The phase ordering problem has been a open research topic in the field of Compilers for many years now. Traditional methods to tackle this problem have involved the use of iterative compilation [7] . Iterative compilation has been able to outperform the highest optimization settings in many compilers and also compares favourably when applied to many hand optimized vendor libraries. However, the key issue is it involves searching a combinatorially large space which could often take days or weeks to tune. Cooper et al. [7] introduced a system called ACME to speedup iterative compilation using a technique called Virtual Execution which introduced a pass to analyze the CFG and identify advantages and disadvantages of applying specific optimizations. The technique works well on simple models but fails to estimate performance of out of order issue processors and requires changes to be made to each optimization pass of a compiler.

Cooper et al. [6] have also used genetic algorithms to solve the phase ordering problem. However, they were primarily concerned with selecting a sequence optimization for code size. Moreover, they had to retrain the model every time to optimize for a new program.

The use of machine learning to construct heuristics to select near optimal phase orders has received considerable attention in recent years [[5],[8],[1], [3],[4]]. Cavazos et al. [7] used supervised learning techniques to decide whether instruction scheduling should be applied to basic blocks or not. Monsifrot et al. [8] used a decision tree based classier to determine which loops to unroll. Stephenson et al. [5] used genetic algorithms to tune heuristics for three compiler optimizations namely hyper block selection, data prefetching and register allocation. However, the key issue with all these works is they focus their attention on individual compiler optimizations, the heuristics perform poorly when considered in a sequence of compiler optimization.

The work most similar to ours is the use of performance counters for optimization pass selection by Cavazos et al. [4]. However, it is important to note that it is not the phase ordering problem that they consider. Moreover, they only consider using performance counters while we have considered a combination of both static code features and dynamic performance counters and demonstrated significant speedup on our test set of automatically generated programs.

6. Conclusion and Future Work

In this paper we address the phase ordering of compiler optimizations using machine learning techniques. To the best of our knowledge we are the first to consider both static code features and dynamic features in the performance counters. We saw significant improvement in the execution time (17.32% avg.) by using both static and dynamic features.

In the future we would like to test our method on several standard benchmarks and analyse if we get similar improvements in the execution times. Moreover we would also like to explore other optimization passes and their different combinations like allowing passes to occur multiple times or passes to not occur at all. It would also be interesting to include more complex features based on CFG. From our initial analysis static features including CFG features gave better results than the use of traditional static features alone. We would like to develop a more principled approach to characterize the CFG and learn the model for predicting phase orders.

We would also like to characterize the phase ordering problem as a sequence tagging problem like Markov Random Field or a Conditional Random Field often used in applications in NLP and Vision. We feel the phase ordering problem is amenable to such a formulation.

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 *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. . URL http://dx.doi.org/10.1109/CGO.2006.37.

[2] John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 183–194, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. . URL http://doi.acm.org/10.1145/996841.996864.

[3] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. *SIGPLAN Not.*, 41 (10):229–240, October 2006. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1167515.1167492.

[4] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. . URL http://dx.doi.org/10.1109/CGO.2007.32.

[5] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Not.*, 34(7):1–9, May 1999. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/315253.314414.

[6] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, August 2002. ISSN 0920-8542. . URL http://dx.doi.org/10.1023/A:1015729001611.

[7] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Acme: Adaptive compilation made efficient. *SIGPLAN Not.*, 40(7):69–77, June 2005. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1070891.1065921.

[8] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler

heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '02, pages 41–50, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44127-1. URL http://dl.acm.org/citation.cfm?id=646053.677574.

[9] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. . URL http://dx.doi.org/10.1109/CGO.2005.29.

Acknowledgments

We would like to take this opportunity to thank Prof. Calvin Lin for the compilers course and motivating us for the project. We would also like to thank Arthur Peters for his suggestions.