A Neural Network Based GCC Cost Model for Faster Compiler Tuning

Hafsah Shahzad*, Ahmed Sanaullah+, Sanjay Arora+, Ulrich Drepper+, and Martin Herbordt*
*CAAD Lab, ECE Department, Boston University
+Red Hat Inc.

***Abstract*—Machine learning models have been found to be effective in predicting compiler heuristics, but are limited by their very long training times. This is because computing the impact of transformations on a code, e.g., through the performance values, involves invoking the downstream compiler. One way to circumvent the cost-computation bottleneck is to devise accurate cost models that can be trained to predict the target metric. In this paper, we develop a neural net based cost function that can accurately predict binary code size for GCC-based compilation. The input to the model is a comprehensive list of features that have been extracted offline from GCC's intermediate tree representation and the compiler flags that need to be evaluated by a compiler tuning workload. To extract the code features, we have built a *GIMPLE* analysis framework that can generate feature sets from intermediate representations at different stages of the compilation process. Our results show that the cost model has a mean absolute percentage error of just 8%, and a Spearman correlation of 0.98 between predicted and measured binary size of the test applications. We also demonstrate that compiler pass selection for feature extraction has a significant benefit on the accuracy of the model. Finally, we show that the cost model can reduce metric evaluation time by multiple orders of magnitude.**

I. Introduction

Even after decades of efforts on improving the code optimization process, there are continually new improvements [1]–[10]. For example, in the last few years, machine learning (ML) has been employed to build models used within the compiler to help make optimization decisions for any given program. Building an efficient ML model requires two things. First, efficient feature engineering is required to derive quantifiable properties to characterise a given code and then iteratively refine them to improve the accuracy of the model. Second, a training data set must be built and fed into a learning algorithm. Here, a training data set comprises a tuple of code features, performance values, and transformations that allow the ML algorithm to build its correlations and enable it to make predictions for an unseen code.

The problem addressed here is that computing the impact of transformations on a code, i.e., the performance values, is an expensive process. It involves invoking the downstream compiler and may take a significant amount of time. And often performance characteristics of the target hardware is altogether unavailable. Due to this cost bottleneck, models that can predict optimization decisions across a wide range of codes require days to train [11]–[13].

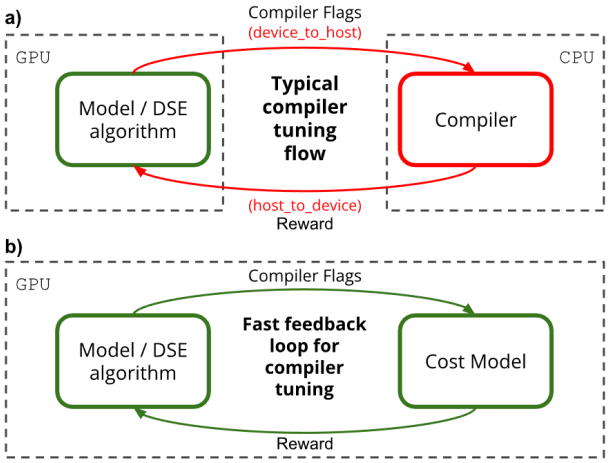


Fig. 1: Circumventing the training time conundrum of compiler tuning models and Design Space Exploration (DSE) algorithms by replacing downstream compiler with neural net based cost model

In particular, while cost functions can be designed to quickly predict performance values, thereby substantially reducing the training times, simultaneously achieving accuracy from them is challenging. In the past several groups have tackled this limitation of compiler optimization by designing manual cost functions [14], [15]. However, this requires expert knowledge about the specific target architecture and can take months to tune [16]. With the proliferation in deep learning, some groups have proposed neural network based cost functions. The cost model implementations in Halide [17] predict runtimes for image processing and deep learning programs, but is limited in scope to Halide programs. The cost model implementation for the Tiramisu compiler [18] predicts speedup in runtime, but focuses on loop transformations.

In this work, we propose a neural network based cost function that addresses the limitations of previous work. We demonstrate accuracy in terms of mean absolute percentage error that is roughly $2\times$ better than previous efforts. Moreover, to the best of our knowledge, our work is also the first employing a neural network based cost model for a production compiler such as GCC. While LLVM is a popular choice in academia, GCC is default choice for a large majority of significant software projects. We are also one of the first efforts in predicting code size as a performance metric, rather than standard schedules and runtime. Code size has been used as the optimization criteria in a number of recent works for compiler optimization [8], [11], [19]–[24]. This is partly because code size reduction, firstly, leads to cost reductions in terms of storage, bandwidth and memory, all critical for today's resource constrained applications [25]; and secondly is platform-independent, deterministic, and relatively noise-free, and hence an ideal target for compiler optimizations [13], [26], [27].

A major challenge has been creating the training set for a cost function since this requires code features for labeling each transformation set and its code size reduction. Feature extraction for GCC is not straightforward: the last major work to extract static features from GCC codes was Milepost [7] and their framework does not work beyond GCC 6+ [28]. We tested other alternatives such as a Python plugin by GCC developers [29]. However the limitation of most prior works is that they are not updated to recent GCC versions and hence are not upgraded for vectorization, inter-procedural optimizations, or new architectures. For LLVM, there are many efforts that allow feature extraction [12], [26], including a custom pass within LLVM [30]. There is a need to develop similar feature extractors at different abstraction levels for GCC's latest versions (11-13).

We have therefore developed a *GIMPLE* (Intermediate Representation for GCC) parser that allows users to extract more than a 100 function-level static features. In addition, users can choose among types of feature extractions: (i) After which GCC pass the features should be extracted? and (ii) Which subset of features are important? We also compare the subset of features extracted with state-of-art work (Autophase [12]) and demonstrate that our framework allows feature selection according to (i) and (ii) that train neural net with better accuracy in predicting code size. To the best of our knowledge, this is the first work to explore static feature extraction in such depth and with such success in predicting accuracy.

The specific contributions of this paper are:

- An automated framework for extracting over 100 function-level static features from GCC's intermediate representation, *GIMPLE*, that allows users to configure their optimization level and optimization flags, select the subset of features and the level of feature extraction (i.e., to analyze tree dumps after a selected compiler pass during the optimization pipeline).
- A neural network based cost-model that takes these code features and GCC flag transformations, and outputs the binary size prediction for GCC.
- An implementation and evaluation of the proposed cost model and mean average error comparison with the state of art.
- A demonstration that models trained using features extracted after individual passes in the optimization pipeline have different accuracy. Also, that by using the comprehensive list of features from our framework, better prediction accuracy is achieved than with features used in prior efforts.

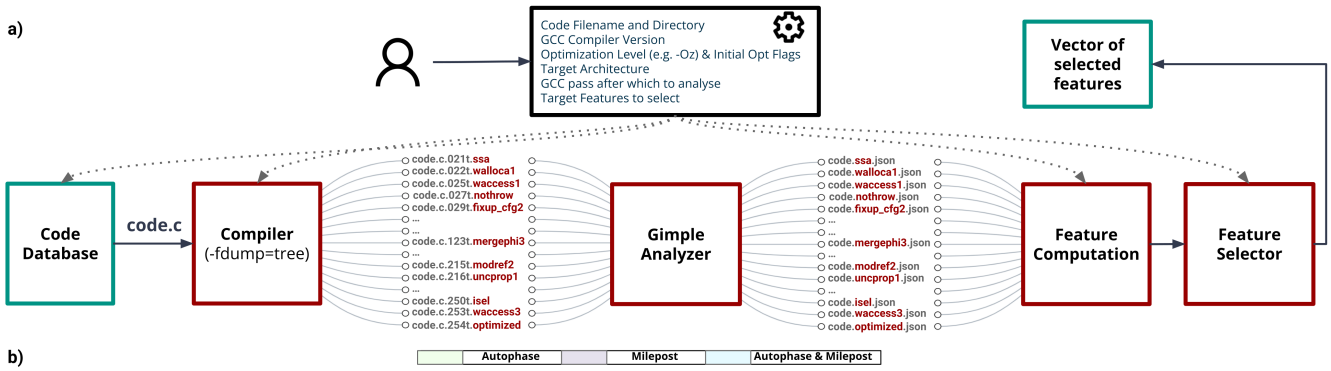
II. Related Work

One of the earliest works in ML-based compiler cost models for GCC was MILEPOST [7]. It trained a k-nearest neighbor model using engineered static features and predicted the best combination of GCC flags for unseen applications. Our work trains a deep neural net to predict code size given a much larger pool of code features and set of GCC flags compatible with latest versions of GCC. Ithemal [31], Granite [32], Facile [33] and DiffTune [34] are basic block throughput estimators. Their limitation is that they do not support loop level transformations and memory accesses.

Static code analyzers like LLVM-MCA [35], OSAKA [36], and IACA [37] are not always accurate and sometimes give large errors [38]–[40]. Similarly early efforts by [41] developed a multilayer perceptron trained on SMG2000. Our approach allows various algorithms.

An artificial neural net to predict tile size performance was developed by [42]. Our work encompasses broader performance characteristics and not a specific trait of the program. Similarly, the static features-based performance model such as [43], and performance counter based feature vectors by Park et al. [44], [45] predict runtime speedup over a limited set of applications. Our work predicts code size with reasonable accuracy. To the best of our knowledge, only [46] have looked at neural nets for code size. However, the scope of their work encompasses only the code duplication and only for GraalVM compiler. We look at a much broader action space of GCC flags.

Static code features for compiler optimization were developed mainly by Milepost [7] and Autophase [12]. Most other groups define their subset depending upon



b)

Autophase | Milepost | Autophase & Milepost

Number of assignment instructions	Number of BitCast Instr	Number of unary operations	Number of volatile in function arguments	Number of EqualToExpr
Number of SwitchInstr	Number of Add insts	Number of BB with instructions less than 15	Number of constant in function arguments	Number of useVars
Number of BB with instructions greater than 500	Number of Multiply insts	Number of BB with instructions between [15,500]	Number of pointers in local declarations	Number of NotEqualToExpr
Number of BB's with >2 successors	Number of Subtract insts	Number of occurrences of 32-bit integer in local declarations	Number of booleans in local declarations	Number of FloatExpr
Average number of instructions in basic blocks	Number of Trunc insts	Number of occurrences of 64-bit integer in local declarations	Number of read-only variables	Number of LabelInstr
Number of local declarations	Number of LShr insts	Number of BB's with 1 successor	Number of Pointer Add insts	Number of defVars
Number of BB's with 2 predecessor and 2 successors	Total arguments to PHI nodes	Number of BB's with 2 predecessors	Number of Pointer Subtract insts	Number of operands
Number of BB's with more than (2 predecessors and 2 successors)	Number of calls that return an int	Number of BB's with 2 predecessors and 1 successor	Number of NOP operations	Number of forward edges
Number of BB's with 2 predecessors and successors	Number of PHI nodes	Number of BB's with 2 successors	Number of for-loops	Number of backward edges
Number of Zero Extension Instr	Number of Return Statements	Number of BB's with >2 predecessors	Maximum loop depth	Number of SSA_NAME insts
Number of unconditional branches	Number of GotoInstr / Number of Br	Number of Binary operations with a constant operand	Count of loop nesting levels=1	Number of GreaterThanExpr
Number of Conditional Statements	Number of ShiftInstr	Number of insts that do pointer arithmetic in the method	Count of loop nesting levels=2	Number of LessThanEqualToExpr
Number of MEMREFS	Number of Sign Extension instr	Number of BB where total args for phi nodes is [1,5]	Count of loop nesting levels>=3	Number of unique operands
Number of Alloca insts	Number of occurrences of constant 0	Number of BB's with Phi node # in range (0,3]	Number of total operators	Halstead_ProgramLength
Number of Call Statements	Number of occurrences of constant 1	Number of BB's with more than 3 Phi nodes	Number of total unique operators	Halstead_ProgramVocabulary
Number of loads	Number of total instructions	Number of BB's with 1 predecessor	Number of Bit insts	Halstead_Volume
Number of stores	Number of conditional branches	Number of BB's with 1 predecessor and 1 successor	Number of DivExpr	Halstead_Difficulty
Number of ICmp insts	Number of basic blocks	Number of BB's with 1 predecessor and 2 successors	Number of CST inst	Halstead_Effort
Number of AND instr	Total number of edges	Number of BB where total args for phi nodes >5	Number of NegateExpr	Number of GreaterThanEqualToExpr
Number of OR instr	Number of critical edges	Number of function arguments	Number of LessThanExpr	Number of array accesses
Number of XOR instr	Number of BB with no phi nodes	Number of pointers in function arguments		

Fig. 2: (a) Feature Engineering Workflow (b) Function level features that can be extracted. The color codes show features that have also been used in two prior state-of-the-art works [7], [12]

the use case, e.g., loop tiling [18] or inlining [13]. We have curated a large list of static features that uniquely characterise different aspects of a code and can also be extracted at intermediate compilation steps of the compiler's optimization pipeline.

III. Feature Engineering

Figure 2(a) shows the framework for static feature extraction. The output is a vector of those feature values that the user specified for the target code. Figure 2(b) lists all of the function-level features that can be extracted from the framework. Features extracted by prior state of art work are shown in color. Green color code specifies features that have been considered by Autophase [12] and later used in many recent compiler optimization works [11], [26]. Purple shows features that Milepost [7] used and Cyan color code shows features that both Milepost and Autophase included in their subset to describe the code. Note some features have been omitted either because they are not relevant for GCC, or for latest versions, or are not appropriate for describing function-level attributes. Below we describe the important blocks.

A. User Configurations

Four sets of parameters are specified by the user: (i) code name, code directory, and build directory; (ii) details about the compiler such as the GCC version (tested for versions 11-13) such as, optimization level (e.g., -O3, -Oz), other flags or command options as specified in [47], the name of the GCC pass applied in the middle-end optimization pipeline for GCC, and the pass after which to analyse the tree dump for features; (iii) target architecture as specified in [48]; and (iv) target features, which should be a subset of features of 2(b).

B. GIMPLE Analyzer

In the first step, the GCC tree representation of functions is extracted. The advantage of using *GIMPLE* is that it is language-independent and its context is simple for optimization passes to operate on [49]. It is also relatively stable with the changes across compiler versions being minimal. GCC can dump out *GIMPLE* representation after every optimization pass it applies. Some of the passes vary with the compiler flags and command options. However, for a fixed optimization level -O(0,g,1,s,2,3,z,fast), the changes are minimal and the framework emits all options available to the user for feature analysis. *GIMPLE* tree dumps for the relevant compile options of the specific C code (i.e., optimization level and GCC flags) are dumped into the build directory. These dump files are named according to the pass after which these dumps are generated. For example, code.c.130t.dom2 is the tree dump after applying the

dominator tree optimizations to the code. 130 is the static pass number for the dom2 pass and can vary across compiler versions. code.c.115t.vrp1 is after value range propagation is applied to the code. All tree dumps are then parsed by the *GIMPLE* Analyzer. The *GIMPLE* Analyzer stores all extracted information from the tree dump into a dictionary. We have tested it across a large range of codes (including elfutils and linux kernels) to ensure the *GIMPLE* Analyzer has the ability to handle corner cases. This information contains details about the functions within the C code, their arguments and types, local declarations with the functions and their values and types, basic blocks in the control flow and all the instructions, phi nodes, and edge source and destination within the basic block. It also decodes all the operands, expressions and variables used within every instruction in a basic block. This information is decoded for tree dumps after every pass and stored in a code.*pass_name*.json file.

C. Feature Computation

The Feature Computation block contains one function for every feature from 2(b). These use the information from the output of the *GIMPLE* Analyzer and compute the relevant feature. For example, to compute the number of back-edges in the control flow graph after a vectorization pass is applied, the file code.c.veclower21.json is read and, from the information on edge source and destination for each basic block, an adjacency matrix is computed. This is then used to calculate the back-edges.

D. Feature Selector

The Feature Selector uses the target features selected by the user in the configuration file and outputs a vector of those feature values from a list of all available features. The Feature Selector has three parts. First, it allows the user to select features using Principal Component Analysis (PCA). PCA is a linear dimensionality reduction technique that transforms the p features into a smaller k (k«p) by taking advantage of the correlations between the input variables in the dataset. Second, it uses features selected in prior works [7], [12]. And third, it shortlists important features using lasso regression analysis. Lasso or L1 is a powerful regularization technique in statistics that shrinks the coefficients of less important features to 0 thus reducing variance in ML models and enabling better learning. A detailed analysis of results from these techniques is presented in Section VI.

IV. Dataset Generation

Since deep neural nets(DNNs) require a large amount of training data, we create a large dataset that is then used to train the DNN. A similar dataset is created for the test set. As a first step, selected features are extracted for the target configurations across 81 C functions. These functions are from a curated list of applications including linear algebra, image and signal processing, and sorting. They also include applications from the Polybench benchmarks suite [50], which has been widely used in prior compiler optimization work [18], [51]. The dataset used in this paper will be open sourced for the community. We split the functions into a randomly selected training set (74 functions) and test set (7 functions).

-fassociative-math -fno-trapping-math -fno-signed-zeros, -fconserve-stack, -ffinite-loops, -fgcse-after-reload, -fgcse-las, -fgcse-sm, -fgraphite, -fgraphite-identity, -flpa-cp-clone, -flpa-pta, -fira-loop-pressure, -fisolate-erroneous-paths-attribute, -fkeep-gc-roots-live, -flimit-function-alignment, -flive-range-shrinkage, -floop-interchange, -floop-nest-optimize, -floop-parallelize-all, -floop-unroll-and-jam, -fmodulo-sched, -fmodulo-sched-allow-regmoves, -fnon-call-exceptions, -foptimize-strlen, -fpeel-loops, -fpredictive-commoning, -frename-registers, -freschedule-modulo-scheduled-loops, -fsched-pressure, -fsched-spec-load, -fsched-spec-load-dangerous, -fsched-stalled-insns, -fsched2-use-superblocks, -fschedule-insns, -fno-section-anchors, -fsel-sched-pipelining, -fsel-sched-pipelining-outer-loops, -fsel-sched-reschedule-pipelined, -fselective-scheduling, -fselective-scheduling2, -fsignaling-nans, -fsplit-loops, -fsplit-paths, -fsplit-wide-types-early, -ftracer, -ftree-cselim, -ftree-loop-distribution, -ftree-loop-vectorize, -ftree-lrs, -ftree-partial-pre, -ftree-slp-vectorize, -ftree-vectorize, -funconstrained-commons, -funroll-all-loops, -funroll-loops, -funswitch-loops, -fvariable-expansion-in-unroller, -fversion-loops-for-strides, -fweb, -fno-aggressive-loop-optimizations, -fno-allocation-dce, -fno-asynchronous-unwind-tables, -fno-auto-inc-dec, -fno-bit-tests, -fno-branch-count-reg, -fno-caller-saves, -fno-code-hoisting, -fno-combine-stack-adjustments, -fno-compare-elim, -fno-cprop-registers, -fno-crossjumping, -fno-cse-follow-jumps, -fno-dce, -fno-defer-pop, -fno-devirtualize, -fno-devirtualize-speculatively, -fno-dse, -fno-early-inlining, -fno-expensive-optimizations, -fno-forward-propagate, -fno-fp-int-builtin-inexact, -fno-function-cse, -fno-gcse, -fno-gcse-lm, -fno-guess-branch-probability, -fno-hoist-adjacent-loads, -fno-if-conversion, -fno-if-conversion2, -fno-indirect-inlining, -fno-inline, -fno-inline-atomics, -fno-inline-functions, -fno-inline-functions-called-once, -fno-inline-small-functions, -fno-ipa-bit-cp, -fno-ipa-cp, -fno-ipa-icf, -fno-ipa-icf-functions, -fno-ipa-icf-variables, -fno-ipa-modref, -fno-ipa-profile, -fno-ipa-pure-const, -fno-ipa-ra, -fno-ipa-reference, -fno-ipa-reference-addressable, -fno-ipa-sra, -fno-ipa-stack-alignment, -fno-ipa-strict-aliasing, -fno-ipa-vrp, -fno-ira-hoist-pressure, -fno-ira-share-save-slots, -fno-ira-share-spill-slots, -fno-isolate-erroneous-paths-dereference, -fno-ivopts, -fno-jump-tables, -fno-lra-remat, -fno-math-errno, -fno-move-loop-invariants, -fno-move-loop-stores, -fno-omit-frame-pointer, -fno-optimize-sibling-calls, -fno-partial-inlining, -fno-peephole, -fno-peephole2, -fno-plt, -fno-printf-return-value, -fno-ree, -fno-reg-struct-return, -fno-reorder-blocks, -fno-reorder-blocks-and-partition, -fno-reorder-functions, -fno-rerun-cse-after-loop, -fno-sched-critical-path-heuristic, -fno-sched-dep-count-heuristic, -fno-sched-group-heuristic, -fno-sched-interblock, -fno-sched-last-insn-heuristic, -fno-sched-rank-heuristic, -fno-sched-spec, -fno-sched-spec-insn-heuristic, -fno-sched-stalled-insns-dep, -fno-schedule-fusion, -fno-schedule-insns2, -fno-semantic-interposition, -fno-short-enums, -fno-shrink-wrap, -fno-shrink-wrap-separate, -fno-signed-zeros, -fno-split-ivs-in-unroller, -fno-split-wide-types, -fno-ssa-backprop, -fno-ssa-phiopt, -fno-stdarg-opt, -fno-store-merging, -fno-strict-aliasing, -fno-thread-jumps, -fno-toplevel-reorder, -fno-tree-bit-ccp, -fno-tree-builtin-call-dce, -fno-tree-ccp, -fno-tree-ch, -fno-tree-coalesce-vars, -fno-tree-copy-prop, -fno-tree-dce, -fno-tree-dominator-opts, -fno-tree-dse, -fno-tree-forwprop, -fno-tree-fre, -fno-tree-loop-distribute-patterns, -fno-tree-loop-im, -fno-tree-loop-ivcanon, -fno-tree-loop-optimize, -fno-tree-phiprop, -fno-tree-pre, -fno-tree-pta, -fno-tree-reassoc, -fno-tree-scev-cprop, -fno-tree-sink, -fno-tree-slsr, -fno-tree-sra, -fno-tree-switch-conversion, -fno-tree-tail-merge, -fno-tree-ter, -fno-tree-vrp, -fno-unwind-tables, -fno-exceptions, -fno-vect-cost-model

Fig. 3: Flags used in this work for binary size cost model

For each C function, we create 10,000 different random sequences of code transformations through GCC flags given in Fig3. Each random sequence of code transformation corresponds to on or off (0 or 1) for the corresponding flag. Then each of the C functions is compiled using each GCC flag sequence and the resulting binary size is measured from the compiled object file. Each vector in the dataset is then a tuple of the form [code features, sequence of code transformations/flags, binary size] with features and flags being model inputs and binary size being the expected output. Overall, we generated 740,000 training and 70,000 test vectors.

V. Model Training

We model the binary size estimation as a regression problem implemented using a neural net in PyTorch. The network architecture is fairly simple, with two dense hidden layers of size 512 each. The model uses the rectified linear unit (ReLu) activation function. We used the Adam optimizer with a learning rate of 10e-3 and Mean Absolute Percentage Error (MAPE) as the loss function. This is suitable for binary code size prediction since the target value is positive by design [18]. The other optimizer parameters are set with default values.

On average across multiple tests, we find that the best accuracy is achieved at around 800 epochs of training. The input features are normalized to the instruction count

of the function. We tried min-max scaling and also standardizing features by removing the mean and scaling to unit variance. We noticed, however, that the learning of the model was better when normalized with respect to the instruction count. In addition to normalizing the input training data to the network, we also normalize the inputs to the layers within the network using the mean and variance of the values in the current batch, a technique called batch normalization [52]. This can improve the efficiency of the neural net and also improve the training speed. In our model batch normalization is applied to the output of both hidden layers. We calculate the MAPE on both the test and training sets and validate that the values are close. This ensures that the model is not over-fit or under-fit and can generalize well on unseen data.

VI. Evaluation

A. Methods

The model evaluation and data collection are performed on a single multi-core CPU + GPU node. The CPU is a 16-core AMD Ryzen Threadripper PRO 5955WX, with 196GB of RAM. The GPU is a NVIDIA GeForce RTX 4070, with 5,888 CUDA cores and 12GB RAM. GCC compiler version 13.2.1 is used to generate the training and test sets. For every compilation run, the -Oz flag is always applied; only the sequences of flags given in Fig. 3 are varied. The test applications used for our experiments were: *2mm* (polybench), *doitgen*(polybench), *heat3d* (polybench), *shellSort*, *boxBlur*, *dotProduct* and *jacobiEigenvals*.

Evaluating Model Accuracy: Model accuracy was evaluated using Mean Average Percentage Error (MAPE). This is given by:

$$\frac{1}{n}\sum_{i=1}^{n} \left|\frac{(actual\_byte\_size - predicted\_byte\_size)}{actual\_byte\_size}\right| \times 100\%$$

We have also used the Spearman rank-order correlation coefficient to measure the monotonicity between the measured and predicted test datasets. A high value shows that the two datasets are highly correlated.

Levels of Feature Extraction: A major advantage of our framework is that we can compute a large list of function-level features and also at intermediate points during the optimization pipeline of the compiler. As discussed in Section III-B, this means that we compile once using the -Oz flag and omit all tree dumps that GCC generates as it optimizes the code for -Oz. These tree dumps are labeled with the optimizing passes that the compiler applies during that compile. For each tree dump of the optimizing pass, we then compute the feature vectors as discussed in Section III. As a next step, for features corresponding to every pass, we generate the training and test dataset as discussed in Section IV. The training dataset at pass *xyz* is used to train the cost model, and its prediction accuracy is recorded on the test dataset. In this way, we shortlist which features extracted after which pass within the optimization pipeline would result in the best cost function.

Selecting a Subset of Features: Our framework has the capability to extract over 100 features (describing the function) as given in Figure 2(b). To demonstrate the value of having such a comprehensive set of code features, referred to as *All Features*, we evaluate prediction accuracy with two methods of feature space pruning. The first method is to statically prune the feature space. This is done by using the feature set given in [12], which is the state-of-the-art work in compiler optimization. We refer to this feature space as *Autophase*. The second method is to prune the feature space at runtime. This is done using L1 or Lasso Regularization. Lasso regularization is applied to the weights of both the layers in the neural net and added to the overall loss computed for the training loop. We refer to this feature space as *Lasso*. We also tried other feature selection techniques such as PCA and Random Forests. However, these last techiques did not give any significant improvement in prediction accuracy of the cost model and hence have been omitted.

B. Finding the Best Level of Feature Extraction

Figure 4 shows the results. A lower MAPE means higher accuracy. The results are shown for different types of feature selections used to train individual cost functions for different passes: All Features from Figure 2(b), prior state-of-art, Autophase [12], and using Lasso Regression. We see that the best accuracy for All Features and Autophase Features occurs at the ccp2 and ccp3 passes, respectively. This is the conditional constant propagation optimization that aims to simplify instructions and local scalar variables [53]. The numbers “2" and “3" mean that the respective pass is being applied for the 2nd or 3rd time during the optimization pipeline. For Lasso, features extracted for tree dumps after the waccess3 pass and then used to train the cost model give the best cost model accuracy. We find that waccess3 is one of the last *GIMPLE* passes to be applied in the optimization pipeline and that, for majority applications, it indeed gives improved feature values (such as lower Halstead difficulty, fewer local variables, and a smaller average number of instructions per basic block).

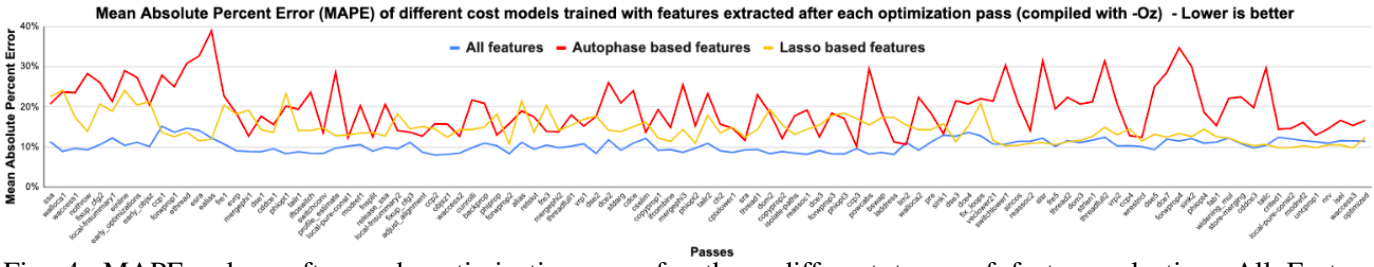


Fig. 4: MAPE values after each optimization pass for three different types of feature selection: All Feature (Comprehensive), Autophase (Prior State-of-art Subset) and Lasso (Runtime Statistical Pruning in Neural Net).

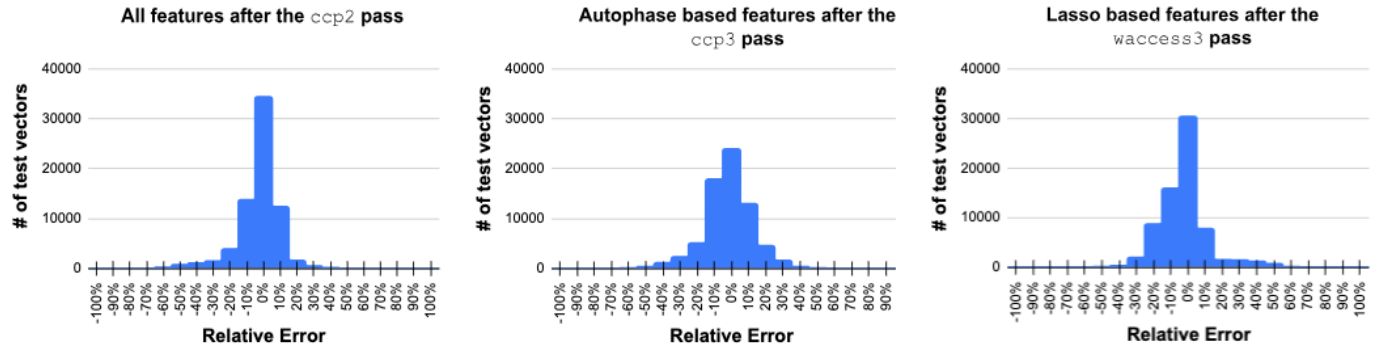


Fig. 5: Percent error distribution for passes that give the best cost model accuracy.

TABLE I : Summary of results

	All Features	Autophase	Lasso
Best test MAPE	8.00%	10.14%	9.79%
Compiler pass	ccp2	ccp3	waccess3
Accuracy	92.0%	89.9%	90.2%
Spearman coefficient	0.98	0.97	0.97

C. Computing the Best Subset of Features

We ask: Which subset of the selected features' set is better at training the binary size cost model for GCC? From training individual cost models using features extracted after passes, we find that the least MAPE error occurs using the proposed comprehensive list of All Features. The MAPE of this cost model is 8%. If we train the cost model using features proposed by Autophase then the MAPE is 10%, and for features selected by Lasso it is 9.79%. These results are summarized in Table I. The Spearman rank coefficient of the cost model is 0.98 when using All Features. This shows that the predicted and measured binary sizes on the test set are highly correlated. Figure 5 shows the accuracy of the cost model visualized as bar plot when used to evaluate the test dataset. The x-axis gives the relative error between predicted and measured sizes, aggregated into buckets of size 10%. The y-axis lists the number of predictions made on the test set that lie in that range of relative error. We note that for "All Features/ccp2" the majority of the test cases have error less than 5%. In contrast, the relative error of "Autophase/ccp3" has higher variance.

D. Accuracy of individual test applications

Table II shows the prediction accuracy for individual applications in the test set, evaluated using the All Features/ccp2 model. The majority achieved >95% accuracy, while the lowest accuracy was 80%.

E. Model Accuracy Comparison with Prior Work

We compare the performance of other cost models trained for compilers such as Tiramisu and GraalVM. In [18], the authors have 16% MAPE in predicting speedups on full programs for the Tiramisu compiler. We achieve MAPE of 8%. In [46] the authors designed a cost model for the GraalVM compiler and quote accuracy in terms of 70% applications lying within 10% of the correct target value. In our case over 86% applications lie within 10% of the corrected predicted binary size.

F. Performance Improvements With Our Cost Model

We ask: How much speedup is possible by substituting GCC's binary size computation with the proposed neural net cost model? How much advantage can be extracted by using a GPU for the inference from the trained cost model? And, How many test cases (batch size) would be needed for the GPU inference to outperform the CPU?

To answer these questions, we measured the time to evaluate the binary size for a given tuple of C function

TABLE II : Average accuracy for test applications

shellSort	boxBlur	doitgen	heat3d	dotproduct	jacobi	2mm
80%	95%	95%	88%	94%	96%	96%

and target compiler flags. Measurements were done for the following combinations: running the actual GCC compiler (CPU + GCC), running inference for the cost model on the CPU (CPU + Cost Model), and running inference on the GPU (GPU + Cost Model). For GPU inference, we varied the batch size from 2 to 1024. From Table III, see that speedup of multiple orders of magnitude is achieved by using the cost model. Even if a compiler tuning workload is fully deployed on the CPU, it is still over 700× faster to use the cost model than the actual compiler. Moreover, in scenarios where multiple test vectors need to be evaluated in a batch, the GPU based inference will significantly outperform CPU.

TABLE III : Performance improvement from cost model

Method	Speedup over CPU + GCC
CPU + GCC	1
CPU + Cost Model	705
GPU + Cost Model (Batch size=2)	134
GPU + Cost Model (Batch size=16)	1675
GPU + Cost Model (Batch size=1K)	134000

VII. Conclusion and Future Directions

In this paper, we present a GCC feature extraction framework that is able to generate a comprehensive set of code features. The intermediate representation used for it can be selected after multiple points in the compilation process. We also present cost model architecture for faster compiler tuning that achieves a high accuracy of 92%, and can deliver orders of magnitude speedup versus running the compiler.

Though we are able to achieve good numbers on MAPE and Spearman's correlation, we still believe we can improve our cost model by incorporating further representation learning to better generate embeddings from the feature vectors. We can also improve the structure of the neural net to enble better learning. Moreover, we can increase the number of applications both for train and test. We can also create a better pool of applications, with varied binary sizes and codes. We can identify which type of applications the model is not able to predict so well and then employ advanced techniques such as code similarity analysis or fine tuning over specific application sets to increase their prediction accuracies.

Acknowledgements

This work was supported, in part, by grant 2024-01-RH01 from Red Hat. The authors also want to thank Professor Manuel Egele of Boston University for his valuable guidance and mentorship.

References

[1] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2012, pp. 147–162.

[2] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[3] R. Mammadli, A. Jannesari, and F. Wolf, "Static Neural Compiler Optimization via Deep Reinforcement Learning," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 1–11.

[4] Z. Gong, Z. Chen, J. Szaday, D. Wong, Z. Sura, N. Watkinson, S. Maleki, D. Padua, A. Veidenbaum, A. Nicolau *et al.*, "An empirical study of the effect of source-level loop transformations on compiler stability," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[5] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 697–709.

[6] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert, "Towards compile-time-reducing compiler optimization selection via machine learning," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–6.

[7] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[8] A. F. d. Silva, B. N. De Lima, and F. M. Q. Pereira, "Exploring the space of optimization sequences for code-size reduction: Insights and tools," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 47–58.

[9] H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "Autoannotate: Reinforcement learning based code annotation for high level synthesis," in *2024 25th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2024, pp. 1–9.

[10] H. Shahzad, A. Sanaullah, S. Arora, R. Munafo, X. Yao, U. Drepper, and M. Herbordt, "Reinforcement learning strategies for compiler optimization in high level synthesis," in *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2022, pp. 13–22.

[11] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. J. Leather *et al.*, "Learning compiler pass orders using coreset and normalized value prediction," in *International Conference on Machine Learning*. PMLR, 2023, pp. 20 746–20 762.

[12] A. Haj-Ali, Q. J. Huang, J. Xiang, W. Moses, K. Asanovic, J. Wawrzynek, and I. Stoica, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *Proceedings of Machine Learning and Systems*, vol. 2, pp. 70–81, 2020.

[13] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," *arXiv preprint arXiv:2101.04808*, 2021.

[14] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization,"

in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994, pp. 85–96.

[15] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–12.

[16] Z. Wang and M. O'Boyle, "Machine Learning in Compiler Optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[17] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[18] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba *et al.*, "A deep learning based cost model for automatic code optimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 181–193, 2021.

[19] M. Poorhosseini, W. Nebel, and K. Grüttner, "A compiler comparison in the risc-v ecosystem," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2020, pp. 1–6.

[20] V. Seeker, C. Cummins, M. Cole, B. Franke, K. Hazelwood, and H. Leather, "Revealing compiler heuristics through automated discovery and optimization," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 55–66.

[21] A. F. Da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimaraes, and F. M. Q. Pereira, "Anghabench: A suite with one million compilable c benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2021, pp. 378–390.

[22] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "Hyfm: Function merging for free," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2021, pp. 110–121.

[23] H. Massalin, "Superoptimizer: a look at the smallest program," *ACM SIGARCH Computer Architecture News*, vol. 15, no. 5, pp. 122–126, 1987.

[24] H. Wang, Z. Tang, C. Zhang, J. Zhao, C. Cummins, H. Leather, and Z. Wang, "Automating Reinforcement Learning Architecture Design for Code Optimization," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, 2022, pp. 129–143.

[25] R. C. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective function merging in the ssa form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 854–868.

[26] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner *et al.*, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 92–105.

[27] T. Theodoridis and Z. Su, "Refined input, degraded output: The counterintuitive world of compiler behavior," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 671–691, 2024.

[28] CTuning, "Reproducing MILEPOST Project using CK Framework (Machine Learning based Self-tuning Compiler)," https://github.com/ctuning/reproduce-milepost-project?tab=readme-ov-file [Last accessed: July 1, 2024].

[29] DavidMalcolm, "GCC Plugin that Embeds CPython inside the Compiler," https://github.com/davidmalcolm/gcc-python-plugin [Last accessed: July 4, 2024].

[30] LLVM, "LLVM opt -stats," https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/InstCount.cpp [Last accessed: July 9, 2024].

[31] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks," in *International Conference on machine learning*. PMLR, 2019, pp. 4505–4515.

[32] O. Sỳkora, P. M. Phothilimthana, C. Mendis, and A. Yazdanbakhsh, "Granite: A graph neural network model for basic block throughput estimation," in *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2022, pp. 14–26.

[33] A. Abel, S. Sharma, and J. Reineke, "Facile: Fast, accurate, and interpretable basic-block throughput prediction," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2023, pp. 87–99.

[34] A. Renda, Y. Chen, C. Mendis, and M. Carbin, "Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 442–455.

[35] LLVM, "llvm-mca - LLVM Machine Code Analyzer," https://llvm.org/docs/CommandGuide/llvm-mca.html [Last accessed: July 2, 2024].

[36] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein, "Automated instruction stream throughput prediction for intel and amd microarchitectures," in *2018 IEEE/ACM performance modeling, benchmarking and simulation of high performance computer systems (PMBS)*. IEEE, 2018, pp. 121–131.

[37] Intel, "Intel® Architecture Code Analyzer," https://www.intel.com/content/www/us/en/developer/articles/tool/architecture-code-analyzer.html [Last accessed: July 2, 2024].

[38] Z. Lin, *Performance Modeling and Optimization for Machine Learning Workloads*. University of California, Davis, 2023.

[39] A. Abel and J. Reineke, "uica: Accurate throughput prediction of basic blocks on recent intel microarchitectures," in *Proceedings of the 36th ACM International Conference on Supercomputing*, 2022, pp. 1–14.

[40] Y. Chen, A. Brahmakshatriya, C. Mendis, A. Renda, E. Atkinson, O. Sỳkora, S. Amarasinghe, and M. Carbin, "Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 167–177.

[41] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30-September 2, 2005. Proceedings 11*. Springer, 2005, pp. 196–205.

[42] M. Rahman, L.-N. Pouchet, and P. Sadayappan, "Neural network assisted tile size selection," in *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*, 2010.

[43] 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*, 2007, pp. 131–142.

[44] E. J. Park, *Automatic Selection of Compiler Optimizations using Program Characterization and Machine Learning*. University of Delaware, 2015.

[45] 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*, vol. 41, no. 5, pp. 704–750, 2013.

[46] R. Mosaner, D. Leopoldseder, L. Stadler, and H. Mössenböck, "Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler," in *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2021, pp. 127–135.

[47] GNU, "GCC Command Options," https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html [Last accessed: July 7, 2024].

[48] GCC, "x86 Options," https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html [Last accessed: July 7, 2024].

[49] J. Merrill, "Generic and gimple: A new tree representation for entire functions," in *Proceedings of the 2003 GCC Summit*, 2003, pp. 171–180.

[50] L.-N. Pouchet *et al.*, "Polybench: The Polyhedral Benchmark Suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, vol. 437, pp. 1–1, 2012.

[51] N. B. Agostini, S. Curzel, V. Amatya, C. Tan, M. Minutoli, V. G. Castellana, J. Manzano, D. Kaeli, and A. Tumeo, "An MLIR-based compiler flow for system-level design and hardware acceleration," in *International Conference on Computer-Aided Design*, 2022, pp. 1–9.

[52] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*. PMLR, 2015, pp. 448–456.

[53] GCC, "Sparse Conditional Constat Propagation," http://gnu.ist.utl.pt/software/gcc/news/ssa-ccp.html [Last accessed: July 13, 2024].