INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

A LLVM Just-in-Time Compilation Cost Analysis

Rafael Auler *Edson Borin*

Technical Report - IC-13-13 - Relatório Técnico

May - 2013 - Maio

A LLVM Just-in-Time Compilation Cost Analysis

Rafael Auler* Edson Borin

Abstract

The compilation time and generated code quality are important factors of a Just-in-Time (JIT) compilation Dynamic Binary Translation (DBT) system. In order to decide which optimization level to apply, a good DBT must know in advance for how long it will halt the execution system until the optimization finishes. If this time is known, the system can make a better judgment on which subset of optimizations it should use. Nevertheless, the algorithms used for code optimization and generation execute a number of steps that is difficult to predict. The algorithms complexity in time is dependent on a transitory input that changes every time a pass from the compilation pipeline finishes. For worklist-based algorithms, the worst-case time and average-case time may differ greatly, affecting the usefulness of an analytical approach to time estimation. This technical report presents an analysis of the LLVM compilation cost and proposes a model to predict this compilation cost as a function of code properties. We use an empirical approach that provides an easy way to understand the behavior of code generation performance and provide a quantitative analysis of the compilation time of all C and C++ functions from the SPEC CPU2006 benchmarks. We also show that the error of our final model is under 21% for 90% of the tests.

1 Introduction

Nowadays the computing world has several machine languages and, thus, options for which a program may be compiled for. A program compiled to a specific machine language cannot run on another processor, forcing the user to either recompile the software or use virtual machine technology to emulate it, which involves the translation from one machine language to another.

Recompilation is not an option if the source code is not available, which is frequently the case for commercial software. Virtual machines, on the other hand, allow the same program to execute on many different computer platforms: the program may be compiled to run on a given processor, but an emulator is able to translate this program to run on another system. In this sense, the definition of the virtual machine concept is a computing platform that does not physically exist, but either mimics the behavior of an existing one to be able to run programs compiled for it or abides to a higher level machine concept that may never physically exist but is useful to represent programs in an easy way to translate them to any real machine.

*The author would like to acknowledge the support from FAPESP grant 09630/2011 and Microsoft Research.

Virtual machines typically translate code on demand. In this case, they either use an interpreter to simulate each program instruction or use a more time-expensive algorithm to translate groups of instructions into efficient groups of host-machine instructions that reproduce the same computation using the much faster physical processor. Furthermore, many workloads exhibit code repetition patterns, loops, that dominate dynamic execution times. In these cases, the expensive just-in-time compilation may yield better performance because, despite the elevated compilation times, the compiled code speed is much higher than that of interpretation, contributing to a lower overall emulation time as code that is already compiled is reused.

It is not simple to adjust a just-in-time compiler – or a dynamic binary translator [3, 8] when performing cross-ISA translations – to enable efficient virtual machines. There is an important relationship of time spent during code generation and the quality of the generated code. To increase performance of the virtual machine, the emulation manager – the component in charge of deciding when to translate a program fragment and which optimizations to apply – must correctly decide the most cost-effective optimizations to apply to a program fragment based on its predicted future frequency of execution.

Typically DBT systems use a simple threshold that determines the execution frequency a trace must reach to be compiled or to be further optimized. However, this is not as effective as the model-based predictor, which weights the benefits and costs of compiling using cost-estimation functions [3]. Therefore, a good DBT should know in advance the cost of compilation – for how long it will halt the execution system until the optimization finishes. If this time is known and if it can predict how many times the trace will repeat itself in the execution, it can precisely determine whether it is worth or not to apply a given set of optimizations.

Nevertheless, the algorithms used for code optimization and generation execute a number of steps that is not trivial. The algorithms complexity in time is dependent on a transitory input, which changes every time a pass from the compilation pipeline finishes. For worklist-based algorithms, the worst-case time and average-case time may differ greatly, making it difficult to build an analytical approach to time estimation that could actually provide useful information to a DBT system.

This technical report presents an analysis of the LLVM compilation cost and proposes a model to predict this compilation cost as a function of code properties. The proposed predictor relies on a linear model, based on Ordinary Least Squares (OLS), that correlates code properties (e.g. instructions counts) with compilation cost. This empirical approach provides an easy way to understand the behavior of code generation with respect to run time. We provide a quantitative analysis of the compilation time of all C and C++ functions from the SPEC CPU2006 [14] benchmarks, and we finish with the presentation of a linear model that is able to predict most inputs with error under 21%.

This technical report is organized as follows. Section 2 presents a brief motivation, Section 3 presents the LLVM compiler infrastructure, which was used in our experiments, Section 4 discusses the experimental framework, Section 5 presents the experimental results, Section 6 discusses related work and Section 7 presents the conclusions.

2 Motivating Example

Suppose a virtual machine is emulating via interpretation the pseudo code in Figure 1. In this example, we have a function call inside a loop in which we do not know how many times it will be executed until run time. When the virtual machine reaches the function call, it must decide whether to continue using the low startup cost but slow steady-state interpretation techniques or to activate the expensive startup cost but fast steady-state just-in-time compilation framework to translate the entire function to the host machine language.

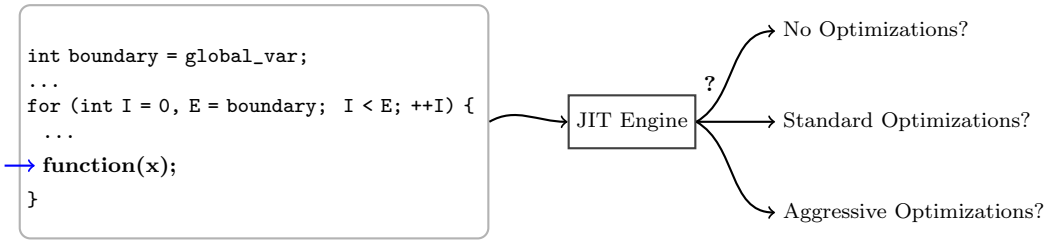


Figure 1: Example showing guest pseudo code and the virtual machine decision process for emulation.

We can instrument this code to discover at run time how many times the loop will be executed and thus extract a lower bound on how many times `function()` will be called. If this number is sufficiently high, we may employ an optimized translation, since the amount of time spent in just-in-time compilation will be paid off by the time gained when repeatedly executing the optimized code.

However, it is difficult to utilize many important optimizations because their cost are unknown and the system is not able to properly decide whether it is profitable to do so. In this technical report, we analyze many LLVM optimization passes to discover how we can better estimate for how long an optimization pass runs, which optimizations are more time consuming and how we can use this information to leverage JIT systems to accurately control code generation timings, providing a powerful technique to improve the JIT strategy.

3 The LLVM Framework

LLVA is a V-ISA (virtual ISA) [2], a class of ISAs that is not intended to be implemented in hardware. Instead, LLVA was conceived to serve as an intermediate language that is well suited for compiler optimizations and a framework for feedback directed optimizations that enabled programs to increase its performance even after deployment. Later, LLVM [2], the LLVA virtual machine, changed its focus and was leveraged to become a powerful open-source static compiler used chiefly by Apple, rather than being used to support a language in which programs are distributed.

Our experiments target the LLVM compiler infrastructure version 3.0. For a deeper discussion on the issues involved in selecting this version, please refer to the Appendix. LLVM is notable for its modularity and capability to be used in a virtual machine or DBT

framework. For instance, it is the first open-source compiler project to support writing the program in intermediate language in a self-contained file [4], which can be later processed by optimization and backend tools. This was essential to enable easy measurements of specific parts of the compiler framework in which we were interested in order to discover the behavior of just-in-time code generators.

Figure 2 depicts a diagram showing typical components of the LLVM framework and how they are utilized to build a just-in-time virtual machine platform. Even though LLVM can be used to build a static compiler, we focus on analyzing its JIT capabilities. Clang, the compiler frontend, translates programs written in the C language to the LLVM intermediate representation (IR). The LLVM IR is based on a three-address static single assignment (SSA) [7] representation suitable for compiler target-independent optimizations. The LLVM optimizer maps LLVM IR to LLVM IR, preserving program semantic but writing it in a more efficient way. Last, the LLVM JIT engine lowers the three-address LLVM IR notation to a DAG that is adequate to instruction selection. This translation process produces target machine code, for example x86 code, which can be used to run functions of the C program efficiently.

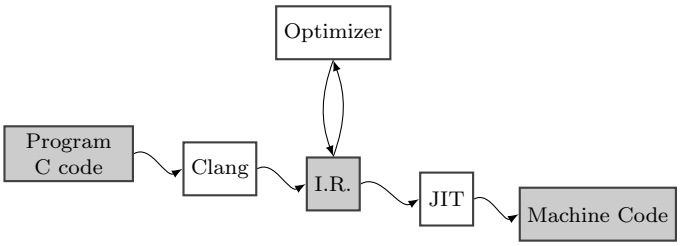


Figure 2: Diagram showing the usage of LLVM and its components as a just-in-time compiler.

Typically, LLVM performs method-based JIT compilation, translating each function (or method) on demand as soon as it is called in program code. However, it is outside the scope of this report to discuss method-based versus trace-based compilation, and we concentrate on analyzing the program execution time of each compiler pass necessary to the JIT compilation.

4 Methodology

4.1 Measuring the Just-in-Time Compilation Flow with Optimizations

Figure 3 shows a diagram explaining the experimental setup for our experiments. C/C++ SPEC CPU2006 benchmarks were compiled to LLVM bitcode, the intermediate language representation readable by the LLVM suite, without using any kind of early optimizations. The LLVM `extract` program, in charge of extracting a single function from the LLVM bitcode and outputting it to a new LLVM bitcode file, was used to organize our experimental inputs into separate functions. In this way, we could analyze JIT compilation time for each SPEC CPU2006 function separately. This step finishes the input preparation.

Later, we measured the execution time of two important LLVM programs. The first one is LLVM `opt`, responsible for reading an input LLVM bitcode, transforming the code using target-independent optimizations and outputting optimized bitcode. The latter is the LLVM `llc`, the LLVM compiler backend that converts LLVM bitcode to x86 assembly language. Together, these two components form an optimizing JIT engine that is the subject of analysis of our experiments.

These two programs are organized in passes, adhering to a classic compiler pipeline concept in which the current optimization pass works using as input the output of the preceding pass. After generating a LLVM bitcode file for each function of C/C++ SPEC CPU2006 benchmarks, we run all `opt` passes that are activated with the "-O2" flag followed by all `llc` passes that are activated by using the same flag in command line. The result is the wall time for each individual pass in each individual SPEC function.

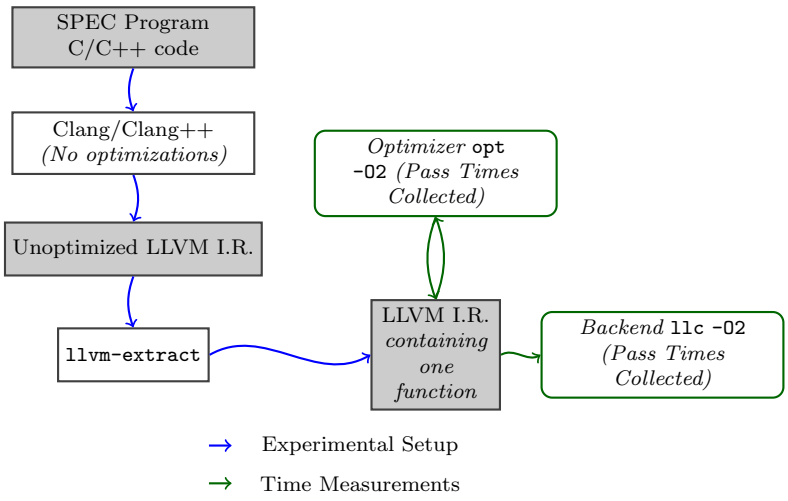


Figure 3: Experimental setup diagram for `opt -O2` and `llc -O2` passes measurements.

4.1.1 Measuring Small Time Deltas

In our experiments we are interested in measuring time as the number of host cycles spent in code translation. To do so, we adapted the LLVM framework to time each `opt` and `llc` pass. When the pass starts, an Intel Core2 hardware performance counter is read to determine the current cycle count since the processor started. After the pass ends, the HPC is read once again and the delta is used to estimate, in processor cycles, the time spent during a code optimization pass.

It may be complicated to measure and validate run times as small as the time needed to perform a single optimization pass on a single function. The time unit used subjects our data to high variability due to systematic errors introduced with operating system side activities. To ameliorate these problems, we repeat every measurement 10 times, discard two outliers and calculate the standard deviation.

We performed our experiments on an Intel Core2Quad Q6600 2.4GHz system with

4GB of RAM running Ubuntu 10.04. We booted Linux with the `bash` shell as the init process, which creates a Linux environment with only one user process (the shell), to run our experiments without unnecessary process switching activity that could disturb our time measurements.

5 Results

5.1 Predicting Optimization Passes Timings Using a Single Predictor Variable

We begin analyzing our data by assessing the efficiency of very simple models based on a single predictor variable and explore their limitations. Figure 4 presents a histogram of SPEC CPU2006 functions by its size, in number of instructions. Since our report focus on the LLVM infrastructure, we will henceforth refer to *instructions* as the number of LLVM intermediate representation instructions that compose the functions.

Roughly 90% of functions have less than 100 instructions of size. This is an important observation because the run time necessary to compile very large functions may be very different and difficult to incorporate into the model. In linear models, big functions have a high predictor value and frequently a high leverage, possibly hindering the quality of otherwise good linear fits for small-sized functions. For this reason we will use the strategy of building segregated linear models: one for the majority of the functions (less than 100 instructions in size) and another for the remaining functions (larger than or equal to 100 instructions).

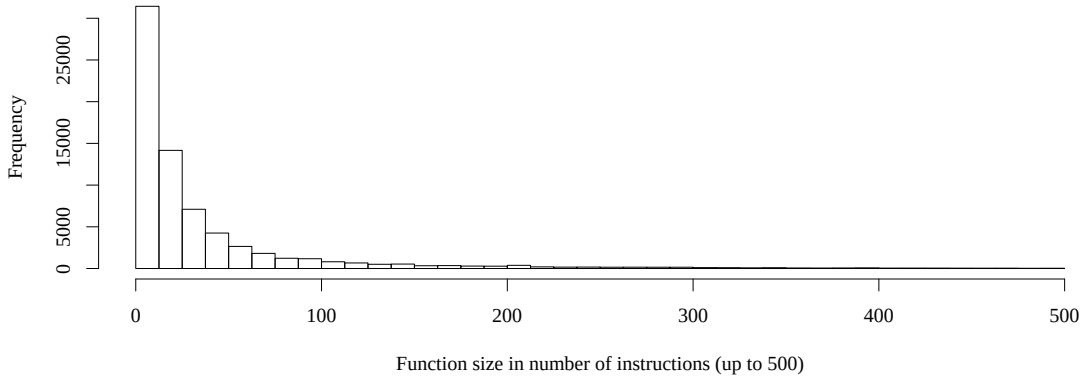


Figure 4: A histogram of SPEC CPU2006 functions sizes, in instructions.

Seeking to design a linear model that predicts the compilation time, we first try to fit optimization pass times (the model response) with the size of the functions being compiled (the predictor), and we assess the feasibility of building such a simple prediction model.

Our size criteria is the number of instructions of the function being compiled. In order to do this, we applied the Ordinary Least Squares (OLS) [20] linear regression technique to predict each execution pass times and measured the R^2 quantity for each pass, the coefficient of determination, which provides a ratio of the performance of the linear model for which the number of instructions of the function is known in advance over the performance of a crude model using the constant function of the average of the time spent in all cases as a predictor. The coefficient of determination ranges from 0, no fit, to 1, fits perfectly a linear combination of the predictor and a constant intercept value. Even though our data exhibit some level of heteroscedasticity [20], which violates the assumptions to correctly use statistical tests that depend on constant residual variance, this does not corrupt the R^2 values *per se* nor the calculated predictor coefficients [20]. Table 1 presents the top 8 best fits using the R^2 criterion.

Tool – Pass Name	R^2
`opt` – Aggressive Dead Code Elimination	0.902
`opt` – Early Common Subexpression Elimination	0.895
`opt` – Sparse Conditional Constant Propagation	0.882
`opt` – Lower expect Intrinsics	0.833
`llc` – Calculate Spill Weights	0.817
`opt` – Module Verifier	0.802
`opt` – Reassociate Expressions	0.787
`llc` – DAG to DAG Instruction Selection	0.739

Table 1: The top 8 optimization and compilation passes whose run times are most easily predictable using a linear model of the size of the function being compiled, in number of instructions, according to the R^2 criterion.

We measured 98 `opt` and `llc` passes, and 38 of them had R^2 lower than 0.5. These results show that estimating pass execution times using only the size of the input function (the function being compiled) may lead to poor models.

Figure 5 shows two graphs with the execution times versus input function size for the Aggressive Dead Code Elimination (ADCE) `opt` pass. In the first version of this graph, we show all 72,000 C/C++ SPEC CPU2006 functions and after a visual inspection, we may say that the linear model fits the data except for an outlier. However, since the majority of functions have less than 100 instructions, this first graph makes it difficult to visualize the most dense and important portion, and to address this problem the second graph focus on these functions with less than 100 instructions. The marks are colored with additional information to help us understand the effects of the number of loops in the time of this optimization. Blue marks are functions with no loops, yellow marks are functions with 1 loop etc. Notice that on the first graph, since a larger set of functions are considered, this range is much greater (from 1 to 60 loops).

Analyzing the second graph ($size < 100$), the functions seem to be more easily predictable if they have 1 loop. The graph also suggests an inflection point because larger functions are frequently underestimated by the model while shorter ones are not. This may

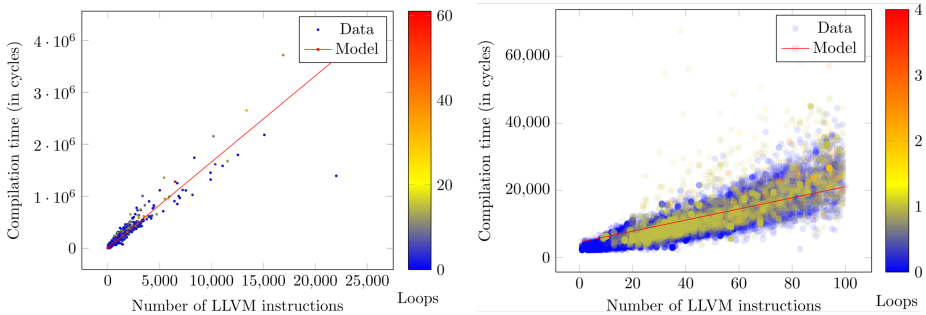
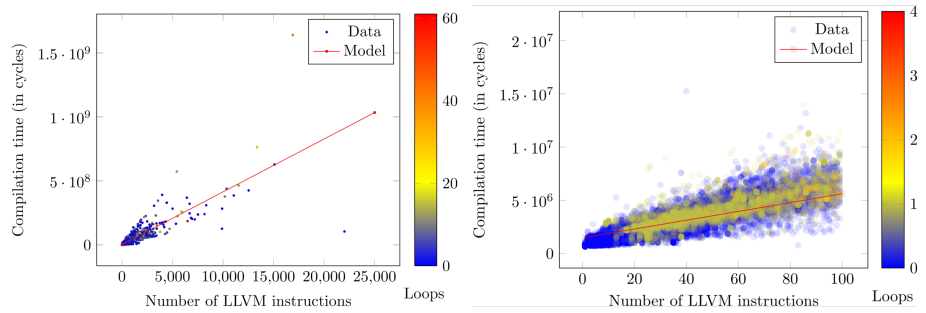


Figure 5: The optimization time, in cycles, required to perform aggressive dead code elimination versus the size of the function, in instructions. The first graph shows all 72.000 C/C++ SPEC CPU2006 functions and the second shows only the more dense region of 63.730 functions smaller than 100 instructions.

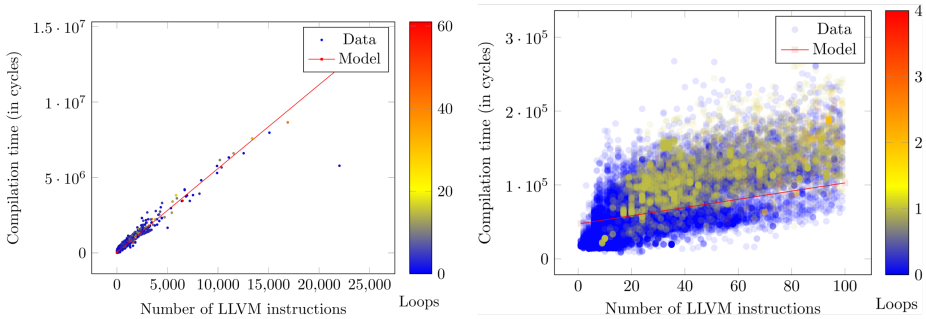
be addressed by adding a quadratic predictor to our model and, indeed, R^2 scores for ADCE improve by adding the size squared as a parameter.

In Figure 6 we present the graphs for other optimizations that have among the best correlation factors between their run time and the size of the input function. These graphs expose a big data variance, and although they suggest that it may not be difficult to make a rough prediction of their execution time using a single predictor variable, there is still room to improve the accuracy. For example it is possible to see in Figure 6 (b) that the model misses the trend for small functions due to high leverages that forces the curve to fit larger functions, as commented earlier. In Figures 6 (a), (c) and (d) we see an exacerbated heteroscedascity – to use statistical parameters in this kind of fit, it is important to rebuild the linear model using log versions of the values, which helps in reducing our kind of non-constant variance. In Figure 6 (d) we see a typical case in which a simple predictor fails to model accurately: two clusters that need additional predictor variables to be separated.

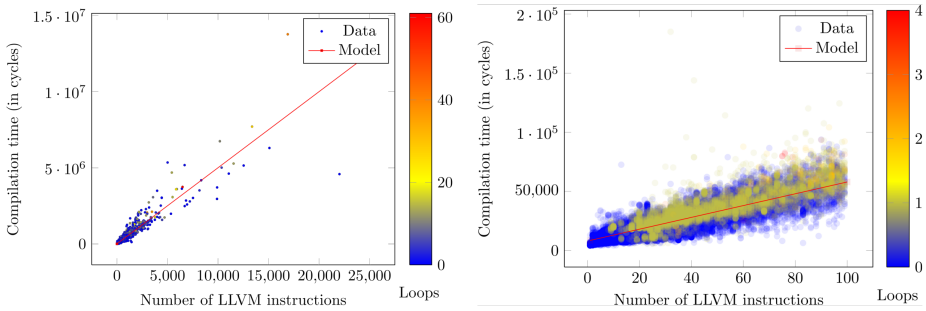
Passes that depend on control flow information, such as loop-based passes, tend to be poorly predicted by the simple model that is based solely on the instruction count. The graph in Figure 7 for the Loop Strength Reduction pass demonstrates this in practice. This bad fit has $R^2 \approx 0.10$.



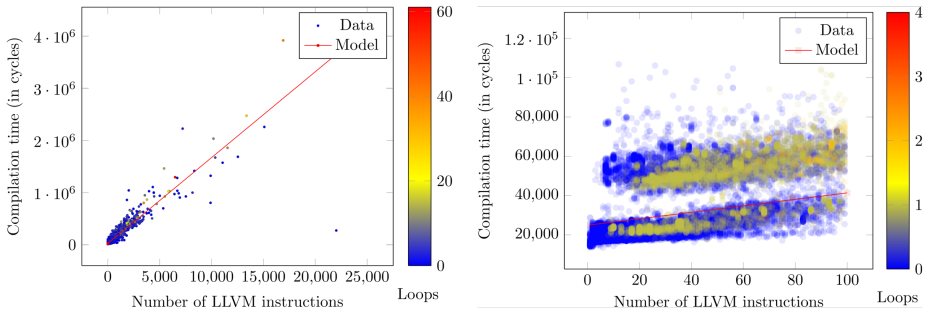
(a) Instruction Selection



(b) Early Common Subexpression Elimination



(c) Sparse Conditional Constant Propagation



(d) Reassociate Expressions

Figure 6: Graphs showing the relationship of the size of functions and the time required to run optimizations on them.

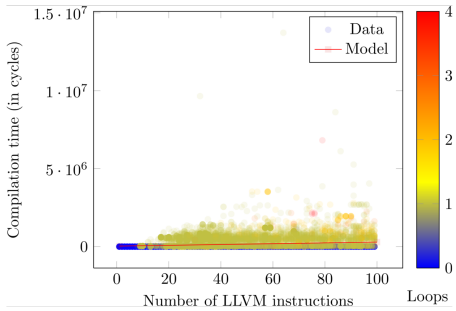


Figure 7: An example of poor correlation between optimization time and size of the input function. It is possible to argue that the Loop Strength Reduction has a stronger dependence on the number of loops rather than the number of instructions.

5.1.1 Sparse Conditional Propagation Case Analysis

Even for those optimizations with the best correlation factors between their run time and the size of their input, the observed variance is high. In the next paragraphs we focus on an analysis of the causes of high variance in the model of the Sparse Conditional Constant Propagation (SCCP) pass.

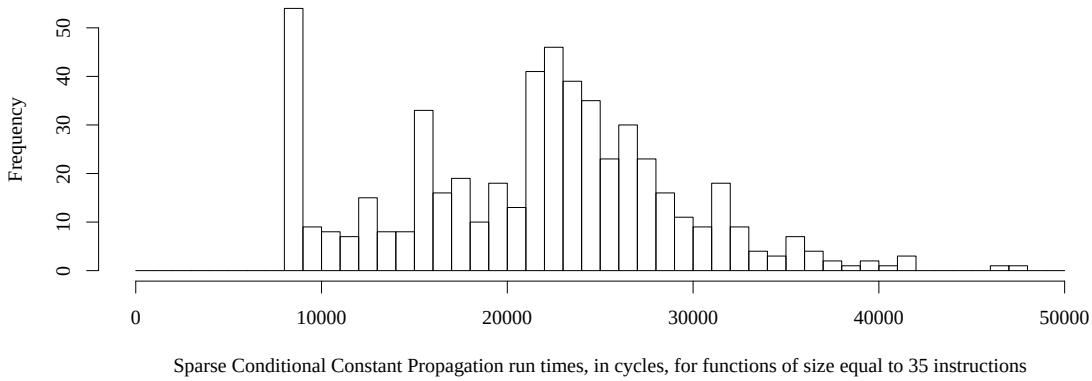


Figure 8: A histogram of SPEC CPU2006 functions with size equal to 35 instructions indicating that the sparse conditional propagation pass took different times to finish.

The histogram in Figure 8 shows SCCP run times when optimizing functions of 35 instructions of size. Ideally, all points should be concentrated around the 24,000 bin, since the linear regression model predicts a function of 35 instructions to take 24,185 cycles in the SCCP pass.

The lowest point, representing the largest negative residual or the fastest case of SCCP,

belongs to a libboost function used in the dealII benchmark, taking 8.709 cycles to optimize 35 instructions. This function is a wrapper of 11 arguments that uses 11 stack allocation instructions, 11 stores to put the arguments into these positions, 11 load instructions to load back these same values to a SSA register, a call instruction to the wrapped function and a return instruction. The fastest case of functions with 35 instructions, therefore, is a single basic block function without loops.

One case representing points from the upper quartile is a gcc function with 9 basic blocks and 1 loop, requiring 38.000 cycles to optimize it. Due to the unoptimized nature of the LLVM bitcode used, most of basic blocks in this function are unnecessary. The function has a single argument. The comparison of these two extremes show that it is hard to model time variations using a single predictor variable because of the different composition of the functions – a wrapper function is extremely simple and most optimizations would quickly dismiss it as a good candidate for optimization.

5.2 Correlation Study

Our experiment records the time of LLVM optimization passes, but it also stores additional information from the input functions, garnered with a simple tool that uses the LLVM library to read the functions and its attributes. They are all parameters known *a priori* that may help guess the run times of passes. We recorded, for each function, its number of basic blocks, the number of loops it has, the maximum nesting level of loops, the number of arguments and also the number of occurrences of each instruction type. For example, the `400.perlbench` function `Perl_allocmy`, which is in charge of helping allocation of memory space at Perl parse time, has 207 instructions, 40 basic blocks, only 1 loop and therefore its maximum nesting level is also 1 and receives 1 argument, the name of the object to be allocated. We also record the frequency of each instruction type for this instruction, finishing the set of information we know *before* compilation.

We also record the number of host machine cycles spent in each pass and these compose the set of information known *a posteriori*. We then performed hierarchical clustering of the pairwise correlation of parameters and found that although some optimizations run times correlates to *a priori* information, others are correlated with other optimizations run times, suggesting that it is possible to use optimization times as predictor variables to other optimization run times. For example, Figure 9 shows the graph of the number of cycles spent in Live Variable Analysis versus the number of cycles spent in the Live Interval Analysis pass, and we see that they share a strong dependence on each other.

This opens the possibility to a heuristic that is able to guess if the remaining compilation time will be big or small based on the tallied run time. We could use the execution time of early optimization passes to improve the predictor accuracy, enabling better compilation decisions on the fly. For example, the JIT system could decide to stop optimizing if it predicts the later optimization phases would take too long. Also, the system could add new optimization passes into the flow if it predicts they will run quickly. Such a possibility is illustrated in the diagram in Figure 10. Even though in our experiments the optimizations that can be predicted by using the run time of previous optimizations did not spend a fraction of the total compilation time large enough to yield substantial improvements in

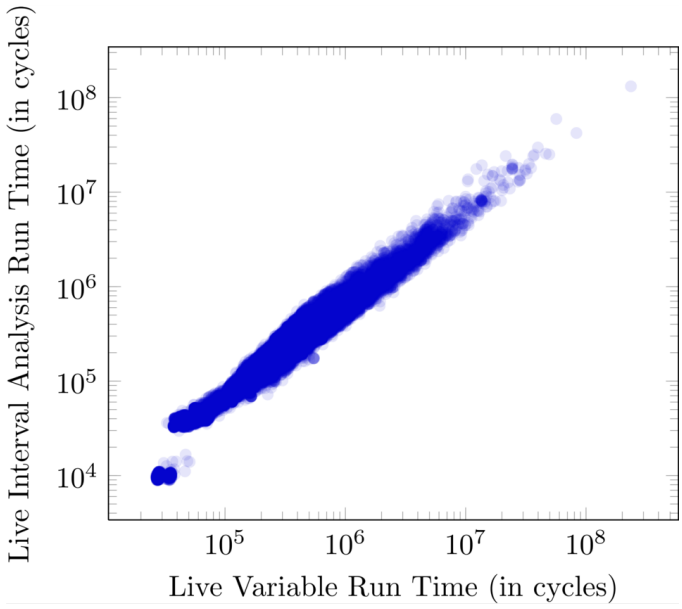


Figure 9: Live Variable Analysis versus Live Interval Analysis

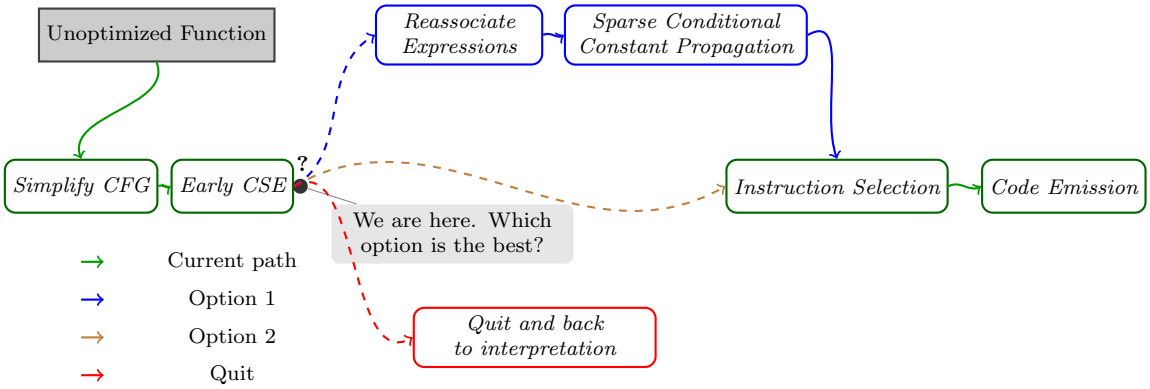


Figure 10: Hypothetical scenario of dynamic adaptive optimization scheduling.

our total compilation time model, this could be investigated in other scenarios and we leave this as future work.

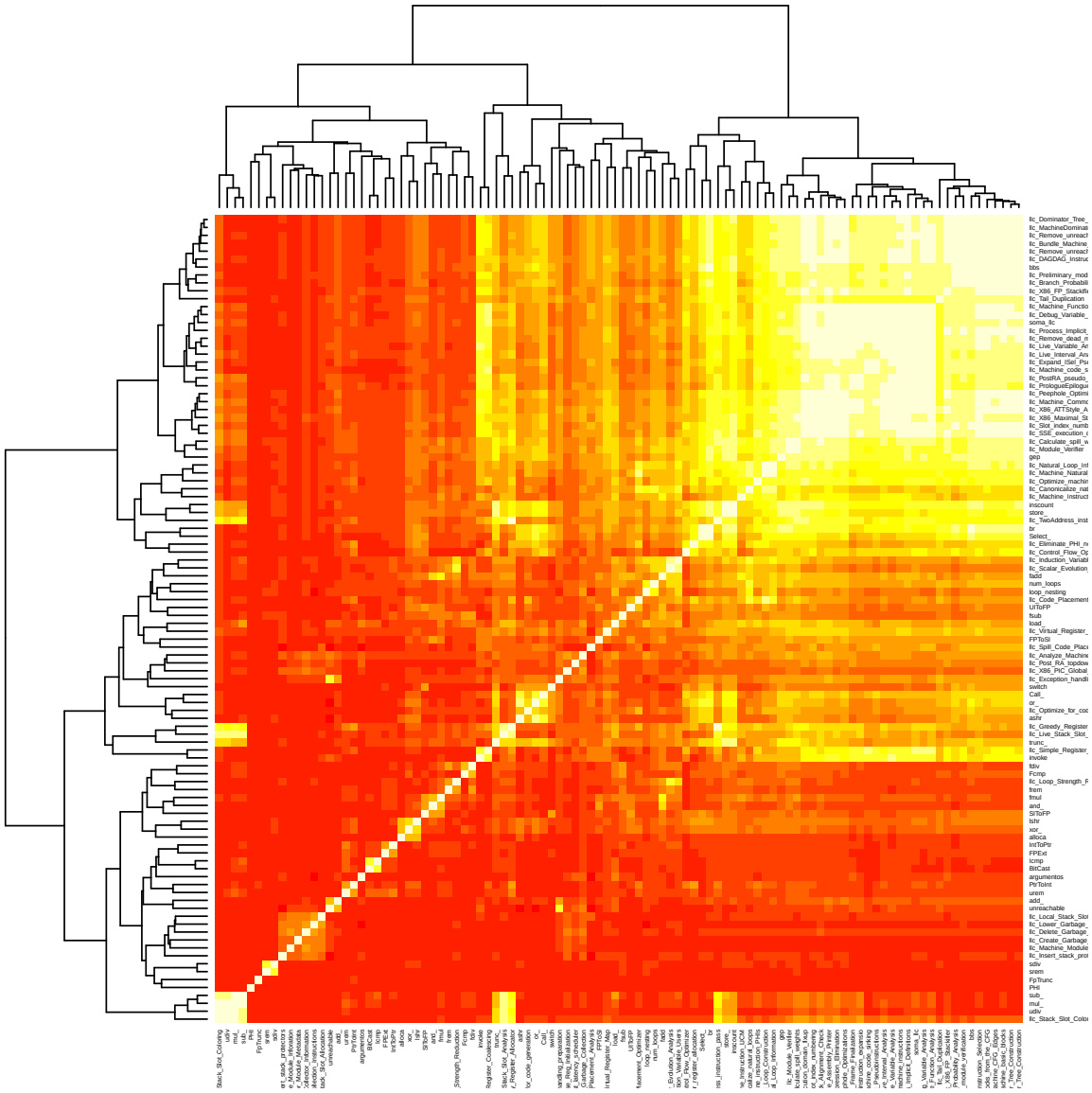


Figure 11: Heat map exposing the correlation factors between pairs of variables, ranging from dark red, low correlation, to bright yellow, high correlation. It also shows a dendrogram to identify clusters of correlated parameters. The bright yellow cluster in the upper right corner shows that many parameters are correlated with the *number of instructions*.

Naturally, not all passes are correlated with others and easily predictable, but our analysis reveals that more than half of all the *a priori* and *a posteriori* parameters we collected share some correlation factor in larger or smaller groups of correlated parameters. Figure 11

shows a matrix with the correlation coefficient between pairs of variables, ranging from dark red, the lowest correlation coefficients, to bright yellow, the highest ones. For example, the *number of instructions* parameter is correlated with several optimizations, as revealed by the bright yellow cluster in the upper right corner of the Figure 11. It is in accordance with the analysis of the previous subsection, where we showed that the number of instructions is a good predictor variable to many optimizations. However, there are examples of small clusters of parameters that do not share correlation or share with a small group: the nesting level parameter is only correlated with Canonicalize Natural Loops and the other *number of loops* parameter.

In another interesting example, the number of occurrences of a peculiar instruction, the LLVM `unreachable` instruction, which is used to inform the optimizer that a particular portion of the code is not reachable, can also be used as a good indicative of the time required to perform the Exception Handling Preparation pass. Therefore, the study of correlation coefficients also provided valuable information to determine *a priori* parameters.

5.3 Predicting Optimization Passes Times Using Many Predictor Variables

In this final section, we focus on building a more accurate and complete model to compilation passes run times, in contrast with the single predictor variable analyzed previously. To this end, we focus on a reduced set of compilation passes: the `llc` LLVM backend passes. We do not consider the passes of the `opt` LLVM target-independent optimizer because we suppose that program binaries would already be optimized and applying target-independent optimizations again would not improve code quality in our method-based JIT environment.

LLC Pass Name	Run Time Coverage and Std. Deviation	Maximum	Minimum
DAG to DAG Instruction Selection	49.52%±7.14%	81.44%	7.31%
Assembly Printer	8.93%±2.67%	15.80%	0.44%
Greedy Register Allocator	8.59%±2.78%	70.13%	0.38%
Live Variable Analysis	4.19%±1.58%	19.75%	0.32%
Live Interval Analysis	2.85%±1.26%	21.74%	0.20%
Prologue/Epilogue Insertion	1.90%±0.66%	3.97%	0.05%
Virtual Register Map	1.79%±0.83%	4.14%	0.005%
Simple Register Coalescing	1.64%±0.81%	57.65%	0.28%
Optimize for Code Generation	1.61%±0.69%	12.28%	0.02%
Module Verifier	1.22%±0.37%	6.02%	0.10%
Dominator Tree Construction	1.22%±0.45%	2.64%	0.004%
Machine Function Analysis	1.13%±0.43%	2.42%	0.05%
Machine CSE	1.08%±0.24%	4.14%	0.10%
Machine Dominator Tree Construction	1.06%±0.41%	2.20%	0.004%
Control Flow Optimizer	1.03%±0.36%	23.54%	0.002%
Calculate Spill Weights	0.96%±0.56%	2.34%	0.01%
Two-Address Instruction Pass	0.93%±0.26%	6.69%	0.13%
Machine Instruction LICM	0.85%±0.35%	2.32%	0.003%
Loop Strength Reduction	0.77%±2.93%	81.99%	0%
Remove Dead Machine Instructions	0.64%±0.16%	1.25%	0.03%

Table 2: The top 20 optimization and compilation passes whose run times takes the largest fraction of the total `llc` execution time, on average.

In Table 2 we show the top 20 `llc` passes that, on average, take the largest fractions of the total `llc` execution time. It is noteworthy that the most time-consuming pass, the instruction selection, has a strong dependence on the size of the input function according to Table 1. Therefore it is important to accurately predict the instruction selection run time because it takes, on average, almost half of the entire `llc` execution time.

The algorithm for the LLVM instruction selection pass iterates on the number of basic blocks and later applies pattern matching on the instructions inside a basic block. The time of the pattern matching depends on the type of the instructions. For instance, a `call` instruction activates a *lowering* routine that converts the high level language `call` to the calling conventions required for performing this operation on the host machine. On the other hand, `add` instructions require a simpler node translation and less changes to the Directed Acyclic Graph (DAG) representation.

We saw in our previous analysis that some parameters may have unexpected correlation with the time of an optimization pass, that the quadratic term of the number of instructions may improve the model quality and the importance of building separate models for each range of function sizes. For example, we know that SPEC functions may range from 0 to 25,000 instructions of size, and attempting to fit a curve using the entire range results in high R^2 linear models because they can converge to model larger functions. The problem lies in the fact that the improved fit of larger functions happens at the cost of a worse fit of smaller functions. This effect can be seen comparing the pair of graphs for the instruction selection and early CSE in Figure 6 – the fit seems good for all functions, but if we focus on the dense region of less than 100 instructions, we see that the curve clearly misses the trend. This is not a profitable trade off to our model because larger functions are rare and often cannot be predicted using a simple extrapolation of the linear fit for smaller functions. This happens due to the lack of a perfect model which would explain time variations for the compilation of any function size.

Considering these points, to increase the accuracy of our predictions of the total time needed to run all `llc` passes, we employ an OLS multivariate linear regression based on our data. We have different models for functions smaller than 100 instructions and for those greater than or equal to 100 instructions. The models use a large number of predictors: the number of instructions, the number of instructions squared, number of loops, nesting level of loops, number of basic blocks and the number of occurrences of each instruction type – a model with 43 predictors. We are not worried in building smaller models because all these parameters are easily computed in a simple pass on the input function. However, we eliminate some instruction type predictors due to its low significance without affecting the model quality, reaching 32 predictor variables and the intercept value. The selected predictor variables are:

> Basic Blocks, Instruction Count^2, Instruction Count, Loops, Loop Nesting, Invokes, Switches, Truncs, GEPs, No. of Function Formal Arguments, BRs, Unreachables, Fadds, Fsubs, Muls, FMuls, Udivs, Ashrs, Ands, Ors, Xors, Loads, Stores, FPToSIs, UIToFPs, SIToFPs, FpTruncs, PtrToInts, IntToPtrs, BitCasts, Fcmps and Selects

To assess its capacity of predicting the total `llc` (LLVM backend) run time using these

predictors, we divided our data into training set and test set. The first model and most important, which predicts the compilation time of functions smaller than 100 instructions, is trained with a set of 50,000 SPEC functions and tested with a set of 13,730 SPEC functions. The elements of these sets are randomly selected from a pool of 63,730 functions smaller than 100 instructions. After training and testing the model, we calculated the percentage of the error (error rate) in a single test using its expected value and the predicted value. Figure 12 presents the density function of the error rates found when testing with 13,730 functions. We can see that the number of errors above 50% are negligible.

In Table 3, we present the percentiles of our error population. For example, the second line shows that 10% of the errors are less than or equal to 1.69%. According to Table 3, the 90th percentile is 20.74%, meaning that in 90% of the cases we can expect the model to be accurate within ±20.74% of the predicted value. The coefficient of determination R^2 for this model is 0.9163. In contrast, the simple linear model using a single predictor and whose domain is all function sizes has a 90th percentile of 49.37% of error and a coefficient of determination R^2 of 0.5178. In our experience, any predictor that is related to the function size roughly approximate the compilation time, but some extra parcels are important to improve accuracy. For example, the number of instructions squared parcel improves the fit considerably. Other parcels give minor improvements, but when combined together lead to a substantial accuracy improvement.

Percentile	Error Ratio
0%	0.002%
10%	1.69%
20%	3.47%
30%	5.30%
40%	7.05%
50%	9.14%
60%	11.59%
70%	13.92%
80%	16.79%
90%	20.74%
100%	155.39%

Table 3: Percentiles values for the error rates found when training the model that predicts total compilation time for functions containing up to 100 instructions.

Trying to predict total compilation time for functions larger than 100 instructions is much harder, since there are fewer data to train the linear model. The 90th percentile for the model predicting such big functions is 58%, even though its coefficient of determination R^2 is 0.928.

By measuring errors as a percentage of the predicted values, we abide by the fact that the data exhibits heteroscedacity: the variance is larger when the predicted compilation time is larger.

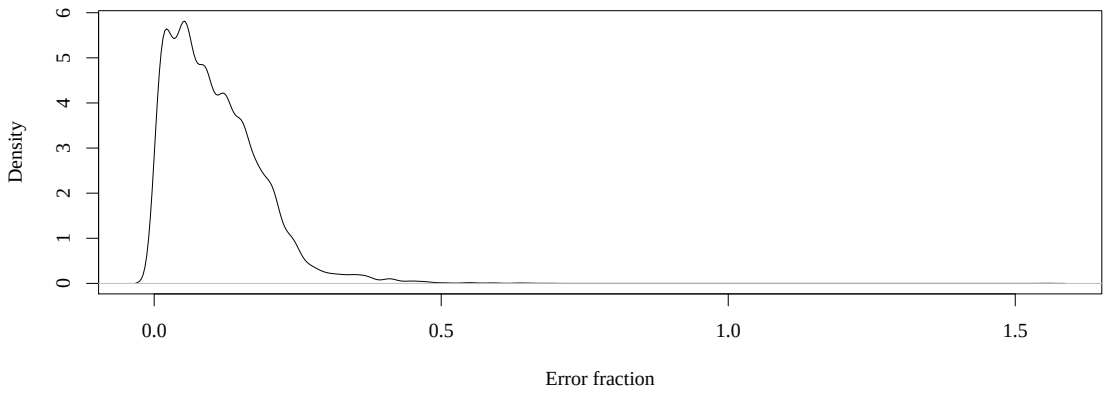


Figure 12: Density function of the error rates for the model predicting total compilation time for functions containing up to 100 instructions.

6 Related Work

Shaw Jr. et al. [19] proposed, in 1989, a linear fit model to predict ADA compilers run times using software science parameters. These parameters and the basic equation was originally created by Halstead [12] and aim at capturing software complexity to estimate the time required by a programmer to complete its implementation. Shaw Jr. et al. extended this work to estimate the time required by a compiler to generate machine code. Since it is a generic model, it could be applied to any compiler and be used to compare compilers and computers. We focus on separate functions smaller than 100 instructions, while Shaw Jr. focus on entire compilation units. However, only the coefficient of determination R^2 and the F test are used to measure the model quality. We trained our model and tested on separate data to evaluate its real predicting capacity on new data. Nevertheless, using the coefficient of determination as a quality factor, our model fits with $R^2 = 0.9163$ while the software science best model had $R^2 = 0.8381$ in one of the systems tested. We believe this difference exists because our model focus on smaller compilation units (functions) and predictors tied closely to LLVM. Similar to our analysis, Shaw Jr. et al. also observed heteroscedacity in their data.

The Jikes RVM (Research Virtual Machine) [1], formerly an IBM internal project called Jalapeño JVM, is a large open-source project and a Java [16] virtual machine implementation frequently used for research. Arnold et al. [3] describe an analytical model used by Jikes' Controller to determine when it is profitable to compile a given code region and in which optimization level, similar to a multi-staged DBT which is able to apply different optimization levels. Arnold et al. measure the speed of compilation as the "compilation rate" in bytecodes per millisecond calculated using the average over seven benchmarks. Thus, they rely on a simple model using only the bytecode size as the linear model predictor.

They do not comment on the accuracy of their model.

An important related field of study is in determining the best set of optimizations to apply on a given compilation unit [5, 6, 10, 13, 15, 17, 18]. Since this problem involves the phase ordering problem, it is not trivial and typical solutions apply a fixed set of optimizations to all programs. Cavazos et al. [5] were able to reduce the Jikes virtual machine execution time on the SPECjvm98 benchmark by 29% on average. They employ machine learning techniques to train the Jikes system to recognize methods and decide which subset of optimizations to apply and its order. Pan et al. [17], Haneda et al. [13] and Pekhimenko et al. [18] investigate methods to automatically find a good subset of optimizations to apply to a given program.

Perkhimenko et al. apply a similar technique of Cavazos to a commercial static compiler, reporting a compilation run time speed up by a factor of at least 2. To do this, a feature vector – characteristics that describe a method – is computed from a program at compile time (statically) for the commercial compiler Toronto Portable Optimizer (TPO), similar to our technique of computing the predictor variables for the open-source LLVM compiler. They also extract instruction types and loop-based parameters to describe methods, using a simpler set of instruction types but a richer set of loop-based parameters. They do not explore using other optimization times as features, and their model differs from ours because it is built to predict a set of 24 values of optimization parameters that preserve code quality while reducing compilation time. Our model focus on the similar problem of predicting compilation time to help scheduling the right subset of optimizations in the JIT compilation.

The GCC MILEPOST project [10] is an adaptive compiler framework that was created for research purposes. MILEPOST relies on machine learning techniques to train the traditional open-source compiler GCC [9] in how to best optimize programs for configurable heterogeneous embedded processors, controlling the internal optimization decisions of GCC via the Interactive Compilation Interface (ICI). The feature vector used by MILEPOST has 55 elements, while we use 32 predictor variables. The majority of the features used by MILEPOST express CFG properties, as the number of edges in the CFG or the number of basic blocks with a single successor, but they do not experimented with using other optimization times as features. Using MILEPOST, Fursin et al. [10] were able to learn a model that improved the performance of the MiBench [11] benchmark by 11%.

7 Conclusion

We present the results of an experiment aimed at assessing the LLVM compilation cost and propose a model to predict the compilation cost. We measured the compilation time spent in code generation and optimization of each SPEC CPU2006 function for C/C++ programs when using the LLVM compiler infrastructure. In order to do this, we measured the time required for each optimization pass separately. We analyzed if it is reasonable to model the time spent in each optimization using a simple linear model based on function size in number of instructions and found that, although the time spent in some optimizations exhibit strong correlation with the number of instructions, this is not true for most passes

because they may depend on other parameters.

We gathered information about the number of basic blocks, number of instructions, number of loops, nesting level of loops, number of formal arguments and an instruction type histogram for each function and used a subset of these parameters as predictor variables of a linear model to predict the total compilation time spent in the LLVM backend. We also separated the model into two different curves, one for the majority of SPEC functions which are smaller than 100 instructions and other for the rest. We found this model to be more accurate than the simple linear model based solely on function size in number of instructions, fitting it with R^2 above 0.9. We tested the model prediction on functions that were not used for training and found that in 90% of the tests the error was less than 20.74% for small functions.

We show evidence that it is possible to predict the run time of some optimizations by using the time of other optimizations as predictor variables, but in our experiments these optimizations did not spend a fraction of the total compilation time large enough to yield substantial improvements in our total compilation time model.

References

[1] Jikes RVM. `http://jikesrvm.org`. Accessed February 2013.

[2] V Adve, C Lattner, M Brukman, A Shukla, and B Gaeke. LLVA: a low-level virtual instruction set architecture, 2003.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F Sweeney. Adaptive optimization in the Jalapeno JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.

[4] Amy Brown and Greg Wilson. *The Architecture of Open Source Applications*. lulu.com, 2012.

[5] John Cavazos and Michael FP O'boyle. Method-specific dynamic compilation using logistic regression. In *ACM SIGPLAN Notices*, volume 41, pages 229–240. ACM, 2006.

[6] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *ACM SIGPLAN Notices*, volume 40, pages 69–77. ACM, 2005.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[8] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization:*

feedback-directed and runtime optimization, pages 15–24. IEEE Computer Society, 2003.

[9] Free Software Foundation, Inc. *Using the GNU compiler collection*, Mar 2013. For GCC version 4.9.0.

[10] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, 2008.

[11] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[12] Maurice Howard Halstead. *Elements of software science*, volume 19. Elsevier New York, 1977.

[13] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 123–132. IEEE, 2005.

[14] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[15] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 62–72. ACM, 2010.

[16] Tim Lindholm and Frank Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 12–pp. IEEE, 2006.

[18] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, page 335, 2010.

[19] Wade H Shaw Jr, James W Howatt, Robert S Maness, and Dennis M Miller. A software science model of compile time. *Software Engineering, IEEE Transactions on*, 15(5):543–549, 1989.

[20] Jeffrey Wooldridge. *Introductory Econometrics: A Modern Approach*. South-Western College Pub, 2 edition, July 2002.

A Appendix: List and Order of LLVM Optimizations

Tables 4 and 5 list the correct order in which LLVM optimizations are applied using the `-O2` switch. This list shows both `opt` and `llc` passes and also shows the relative cost of each optimization, which was calculated using the sum of the number of cycles spent in each pass for all analyzed SPEC functions and then divided by the total number of cycles spent in all passes.

We excluded the Xalan benchmark from this analysis and considered all other C/C++-based benchmarks because if it is included, we observe a distortion in the `opt` pass times for Basic Call Graph Construction, Target Library Information, Dead Argument Elimination, Remove Unused Exception Handling Info and Function Integration/Inlining. The reason is that we use the `llvm-extract` tool as part of our experiment framework (see Section 4) to isolate a single function from a LLVM bitcode, and, in version 3.0, `llvm-extract` bundles into the output too many unnecessary function declarations for the Xalan functions. Although unnecessary function declarations seem harmless to the compilation time, some `opt` passes operates over Call Graph Strongly Connected Components (SCC) and the LLVM pass manager interprets each function declaration as a new call graph component. Even though the passes do not perform useful work using these components as inputs, they do have the time-consuming task of iterating over each component, which can reach over 1200 independent components for Xalan functions and cause SCC-dependent passes to significantly raise their time in an unrealistic way.

We also tested versions 3.1 and 3.2, but although `llvm-extract` was fixed to avoid including unnecessary function declarations, these versions were non-functional for many functions from the SPEC benchmark. Since this effect is harmless for `llc` passes and all other `opt` passes and Xalan contributes to a significant number of functions to our analysis (roughly 50%), we chose to do not exclude Xalan for all other analysis in this technical report.

`opt` Pass	Relat. Cost	Acc.
Dominator Tree Construction†	0.43%	0.43%
Simplify CFG†	1.21%	1.63%
Early CSE	0.73%	2.37%
Target Library Information	0.29%	2.66%
Global Variable Optimizer	0.26%	2.92%
Interprocedural Sparse Constant Propagation	0.72%	3.64%
Dead Argument Elimination	0.49%	4.14%
Combine Redundant Instructions†	3.52%	7.65%
Simplify CFG	–	7.65%
Basic Call Graph Construction	0.81%	8.47%
Remove Unused Exception Info	0.89%	9.36%
Function Integration/Inlining	1.29%	10.65%
Deduce Function Attributes	1.22%	11.87%
Early CSE	–	11.87%
Simplify Well-Known Libcalls	0.36%	12.23%
Combine Redundant Instructions	–	12.23%
Tail Call Elimination	0.09%	12.32%
Simplify CFG	–	12.32%
Reassociate Expressions	0.29%	12.60%
Dominator Tree Construction	–	12.60%
Natural Loop Information	0.18%	12.79%
Canonicalize Natural Loops†	0.15%	12.94%
Loop-Closed SSA Form Pass†	0.13%	13.06%
Rotate Loops	0.30%	13.36%
Loop Invariant Code Motion	0.41%	13.77%
Loop-Closed SSA Form Pass	–	13.77%
Unswitch Loops	0.04%	13.82%
Combine Redundant Instructions	–	13.82%
Scalar Evolution Analysis	0.07%	13.89%
Canonicalize Natural Loops	–	13.89%
Loop-Closed SSA Form Pass	–	13.89%
Induction Variable Simplification	1.45%	15.34%
Recognize Loop Idioms	0.04%	15.37%
Delete Dead Loops	0.04%	15.41%
Unroll Loops	0.14%	15.55%
Memory Dependence Analysis†	0.09%	15.64%
Global Value Numbering	3.18%	18.82%
Memory Dependence Analysis	–	18.82%
MemCpy Optimization	0.15%	18.97%
Sparse Conditional Constant Propagation	0.35%	19.32%
Combine Redundant Instructions	–	19.32%
Lazy Value Information Analysis	0.08%	19.40%
Jump Threading	0.78%	20.17%
Value Propagation	0.71%	20.89%
Dominator Tree Construction	–	20.89%
Memory Dependence Analysis	–	20.89%
Dead Store Elimination	0.25%	21.14%
Aggressive Dead Code Elimination	0.13%	21.27%
Simplify CFG	–	21.27%
Combine Redundant Instructions	–	21.27%
Strip Unused Function Prototypes	0.02%	21.29%
Bitcode Writer	12.53%	33.82%

†Pass ran multiple times – total time on its first appearance.

`llc` Pass	Relat. Cost	Acc.
Preliminary Module Verification†	0.01%	33.84%
Dominator Tree Construction†	0.43%	34.27%
Module Verifier†	0.51%	34.77%
Preliminary Module Verification	–	34.77%
Natural Loop Information†	0.20%	34.97%
Canonicalize Natural Loops	0.15%	35.12%
Scalar Evolution Analysis	0.08%	35.20%
Induction Variable Users	0.86%	36.06%
Loop Strength Reduction	2.14%	38.20%
Lower Garbage Collection Instructions	0.01%	38.21%
Remove Unreachable Blocks from CFG	0.06%	38.27%
Exception Handling Preparation	0.17%	38.44%
Optimize for Code Generation	0.77%	39.21%
Insert Stack Protectors	0.01%	39.22%
Preliminary Module Verification	–	39.22%
Module Verifier	–	39.22%
Machine Function Analysis†	0.40%	39.62%
Natural Loop Information	–	39.62%
Branch Probability Analysis†	0.13%	39.75%
DAG to DAG Instruction Selection (Includes legalization and scheduling)	35.14%	74.89%
Natural Loop Information	–	74.89%
Dominator Tree Construction	–	74.89%
Branch Probability Analysis	–	74.89%
X86 PIC Global Base Reg Initialization	0.01%	74.90%
Expand ISel Pseudo Instructions	0.02%	74.93%
Tail Duplication	0.09%	75.02%
Optimize Machine Instruction PHIs	0.03%	75.13%
Local Stack Slot Alloc.	0.01%	75.14%
Remove Dead Machine Instructions	0.29%	75.43%
Machine Dominator Tree Construction†	0.38%	75.81%
Machine Natural Loop Construction†	0.16%	75.97%
Machine Instruction LICM	0.41%	76.38%
Machine CSE	0.60%	76.97%
Machine Code Sinking	0.26%	77.23%
Peephole Optimizations	0.13%	77.37%
X86 Maximal Align Check	0.01%	77.38%
Remove Unreachable Machine Blocks	0.07%	77.45%
Live Variable Analysis	3.04%	80.49%
Eliminate PHI nodes for RA	0.44%	80.93%
Two Address Ins. Pass	0.55%	81.49%
Process Implicit Definitions	0.19%	81.68%
Slot Index Numbering	0.32%	82.00%
Live Interval Analysis	2.03%	84.03%
Debug Variable Analysis	0.14%	84.17%
Simple Register Coalescing	2.97%	87.14%
Calculate Spill Weights	0.48%	87.62%
Live Stack Slot Analysis	0.02%	87.64%
Virtual Register Map	0.48%	88.12%
Bundle Machine CFG Edges	0.03%	88.15%
Spill Code Placement Analysis	0.08%	88.23%
Greedy Register Allocator	6.27%	94.50%
Stack Slot Coloring	0.16%	94.66%

Continued on the next page on Table 5.

Table 4: The order of optimizations that the `opt` and `llc` tools apply when using the `-O2` switch and their cost in time. The time required for optimizing and compiling all C/C++ SPEC CPU2006 (Xalan excluded) functions was tallied and we show the share that each optimization pass has on this total time. Since some optimizations are applied multiple times and we do not discriminate the time used for each different invocation of the same pass, we report their total share in their first appearances in the table.

`llc` **Pass (continuing from Table 4)**	**Relat. Cost**	**Acc.**
Machine Instruction LICM	–	94.66%
Bundle Machine CFG Edges	–	94.66%
X86 FP Stackifier	0.21%	94.86%
Post-RA Pseudo Instruction Expansion	0.06%	94.92%
Prologue/Epilogue Insertion & Frame Finalization	0.64%	95.56%
Post-RA Top-Down List Latency Scheduler	0.08%	95.56%
Machine Natural Loop Constructor	–	95.56%
Machine Dominator Tree Constructor	–	95.56%

†Pass ran multiple times – total time on its first appearance.

`llc` **Pass (continuing)**	**Relat. Cost**	**Acc.**
Control Flow Optimizer	0.94%	96.50%
Tail Duplication	–	96.50%
Analyze Machine Code for GC	0.01%	96.51%
Machine Dominator Tree Constructor	–	96.51%
Machine Natural Loop Constructor	–	96.51%
Code Placement Optimizer	0.05%	96.57%
SSE Execution Domain Fixup	0.12%	96.69%
Assembly Printer	3.32%	100.00%

Table 5: Continuing the list of passes, in order, that the `llc` tool applies when using the `-O2` switch and their relative time cost based on the sum of the time for optimizing all C/C++ SPEC CPU2006 functions (Xalan excluded).