On the Determination of *Inlining Vectors* for Program Optimization

Rosario Cammarota¹, Alexandru Nicolau¹, Alexander V. Veidenbaum¹,
Arun Kejariwal², Debora Donato², and Mukund Madhugiri²

¹ University of California Irvine

{rosario.c,nicolau,alexv}@ics.uci.edu

² Yahoo! Inc.

{akej,donato,mukundm}@yahoo-inc.com

Abstract. In this paper we propose a new technique and a framework to select inlining heuristic constraints - referred to as an *inlining vector*, for program optimization. The proposed technique uses machine learning to model the correspondence between inlining vectors and performance (*completion time*). The automatic selection of a machine learning algorithm to build such a model is part of our technique and we present a rigorous selection procedure. Subject to a given architecture, such a model evaluates the benefit of inlining combined with other global optimizations and selects an inlining vector that, in the limits of the model, minimizes the completion time of a program.

We conducted our experiments using the GNU GCC compiler and optimized 22 combinations (*program, input*) from SPEC CINT2006 on the state-of-the-art Intel Xeon Westmere architecture. Compared with optimization level, i.e., -O3, our technique yields performance improvements ranging from 2% to 9%.

1 Introduction

The widespread use of object-oriented programming models and software engineering methodologies often leads to complex program structures that are composed of a multitude of functions and source files. The presence of these files and functions unfortunately limits the scope of global optimizations and their forced separate compilation reduces the performance in complex and unpredictable ways. As a result, when relying on current compiler technologies and rigid compiler heuristics, programs achieve in practice only a portion of the performance that they could in principle achieve on a given architecture.

Function inlining [1] provides a simple - in principle - way of overcoming these barriers to program optimization - it removes the boundaries of function calls by expanding call sites with the body of the callee - but it too is not without pitfalls. Prior studies acknowledge the potential that function inlining offers at compile time to other optimizations [2–4], parallelization [5] and vectorization [6]. The importance of function inlining is also evidenced by the fact it is used in all optimizing compilers - e.g., the GNU GCC, Intel ICC, IBM XLC - as a first

pass of their aggressive optimization levels such as -O3. Unfortunately, inlining in an effort to enable maximal optimization to the code has been shown to be NP-Complete [7, 8] and while heuristics are implemented by all good optimizing compilers, the number of constraints and their combinations to the process are extremely large - making the achievement of best, or even good inlining for a given application, practically unachievable. The above represents a limitation to the direct application of specialized search techniques [9, 10] to the case of static compilation of production software written in C or C++. For a given program, searching the space of inlining vectors would involve a large number of expensive recompilations and runs. For example, the number of inlining constraints vary from compiler to compiler - GCC v3.4 has 4 inlining constraints, whereas GCC v4.5 has 7 inlining constraints - and while in practice the range of values of each constraint is somewhat limited (because of compiler malfunctioning), the number of combinations is still enormous. In this work - refer to Section 4.2, we estimate the number inlining vectors to be $> 10^{15}$.

Despite the importance of function inlining, the influence of inlining constraints on performance of arbitrarily complex programs and on high-performance architectures had received little attention in the past. Cavazos and O'Boyle in [9] developed a technique for dynamically tuning the JikesRVM [11] inlining heuristics and selectively inline frequently used functions. However, in the case of static compilation, inlining decisions incrementally influence any inlineable functions within each module [7, 10], making the problem of selecting inlining vectors more complex than that of inlining frequently used functions. Cooper et. al [10] proposed a new heuristic for function inlining for C programs and a technique for auto-tuning its parameters using genetic algorithms. While such an heuristic performs better inlining decisions compared with the inlining heuristics implemented in production compilers, e.g., GCC v3.3, the technique proposed in [10] does not account for separate compilation, where additional barriers to global optimization are the boundaries of source files.

In this paper we develop a new technique for function inlining, given a compiler and its parameterized heuristic. The proposed technique attempts to achieve the best settings of the inlining constraints - referred to as an *inlining vector*, that will result in the most efficient (*fastest*) execution of the code produced by this compiler. Our technique is suitable to be used in industry settings as a support to production compilers as it does not modify the compiler. On the contrary, the technique proposed in this work suggests inlining vectors for program optimization and provides an estimation of the performance that can be achieved after recompiling a program using the predicted inlining vectors.

The proposed technique is implemented in as a performance framework whose design is shown in Figure 1 and is composed of two main passes. In the first pass, it uses machine learning to model the relation between inlining vectors - *ivs*, and performance (completion time) - *p*. The fact that the completion time of a program is a continuous quantity leads us to the adoption of supervised learning algorithms to build regression models. As part of our technique we provide a quantitative procedure to evaluate and select a regression algorithm, amongst

Fig. 1. Performance framework design

many regression algorithms, that best models the relation $\hat{h} : \mathbf{iv} \rightarrow p$ subject to a number of training samples - referred to as *training set*. Such a procedure is based on 10-fold cross validation [12] that, in our technique, is used to evaluate and compare different models in terms of mean absolute error - i.e., the average accuracy of performance prediction, and mean coefficient of correlation - i.e., the average correlation/alignment between a batch of performance predictions and the actual batch of performance - refer to Appendix A. Once assigned a threshold on the maximum admissible mean absolute error, the model corresponding to the algorithm with maximum mean coefficient of correlation is selected amongst the models whose mean absolute error is within the assigned threshold.

In the second pass of our technique, the selected model is leveraged to search for an inlining vector - $\mathbf{iv}^*$, which is able to maximize predicted and ultimately achieved performance of a program subject to a given workload (input). At the cost of training a model using a relatively limited number of compilations and

runs, such a model supplies for the intractability of searching the space of inlining vectors without performing additional recompilations and runs.

In our experiments, we used GNU GCC version 4.5 - referred to as GCC v4.5 - to optimize 22 combinations (*program, input*) on the state-of-the-art Intel Xeon Westmere architecture. These combinations belong to four programs selected from SPEC CINT2006 (refer to Section 4.1 and [13]) because these programs have a larger number of input files, compared with other applications in the same benchmark suite, and because their behavior is highly influenced by the selection of a particular input file [14]. While the optimization of a program subject to a workload may seem a limiting factor of the proposed technique, in industry settings programs are purposely manually optimized for certain a certain class of workloads [14]. We train our technique using a limited number of instances of the 22 combinations (*program, input*), for a limited number of training inlining vectors - refer to Section 3.2. Our technique identifies random forest - M5P [15] - as a suitable algorithm to model the relation between inlining vectors and completion time. Next, the performance model is used to predict inlining vectors to optimize the performance of each combination (*program, input*). Lastly, we recompile our programs using the inlining vectors predicted above. Experimental results show: (1) Performance improvements ranging from 2% to 9% on the state-of-the-art Intel Xeon Westmere architecture and without manually modifying neither the structure of the program source code nor that of the compiler; (2) Performance results outperform both the baseline performance and the best performance recorded during the training phase.

To the best of our knowledge, this is the first paper investigating the influence of inlining vectors on program performance and in the presence of separate compilation. This is also the first paper proposing a rigorous procedure to build and enforce machine learning based performance models. The contributions of this paper can be summarized as follows:

- I - We propose a new machine-learning based technique to select inlining vectors for program optimization and implement the proposed technique in the performance framework illustrated in Figure 1. Our technique relies on the selection of a performance model to quantify the influence of inlining vectors on program performance. The presence of such a model represents a practical mean to perform a fast exploration of the a large set of inlining vectors, whereas the model is built using a limited number of training examples - refer to Section 3.3.
- II - We propose a rigorous procedure to select a machine learning algorithm to build the model above. Using 10-fold cross-validation, such a procedure focuses on algorithms providing models whose performance prediction accuracy is below an admissible threshold - which is a parameter of our framework. Amongst these model, our procedure selects the model with highest mean coefficient of correlation.

The rest of the paper is organized as follows: In Section 2 we describe the opportunity of improving performance by determining inlining vectors and motivate the need of using machine learning to address the problem of selecting

Table 1. 401.bzip2, hot functions

Source file:Function name	Size [kB]	# of calls	Coverage
compress.c:BZ2_compressBlock	15.76	143	28.75
blocksort.c:fallbackSort	2.48	34	27.72
decompress.c:BZ2_decompress	10.92	17681	16.00
blocksort.c:mainSort	4.45	143	9.71
blocksort.c:mainGtU	0.55	1515464	7.96

Table 2. 401.bzip2, hot paths on the precise dynamic call graph

Source file:Caller name	Source file:Callee name	# of edge traversals	Coverage [%]
bzip2.c:main	bzip2.c:compressStream	3	79.14
bzlib.c:BZ2_bzCompress	bzlib.c:handle_compress	36345	78.95
bzlib.c:compressStream	bzlib.c:BZ2_bzWrite	18876	77.99
bzlib.c:BZ2_bzWrite	bzlib.c:BZ2_bzCompress	35986	77.93
bzlib.c:handle_compress	compress.c:BZ2_compressBlock	143	74.61

inlining vectors. In Section 3 we present our technique, including (a) a quantitative procedure to build and evaluate a suitable performance prediction model to select inlining vectors; (b) the detailed discussion on how to build the training set. In Section 4, we describe the experimental setup, the experiments and discuss experimental results. We highlight both the significance of our performance improvements and trends that are common to the predicted inlining vectors, subject to the architecture and the applications in use. Related work is discussed in Section 5. Key highlights and conclusion are remarked in Section 6.

2 Motivation

In this Section we use the program 401.bzip2 from the industry standard benchmark suite SPEC INT2006 [13] as an example program to show opportunities to improve program performance by properly selecting inlining vectors. We compile the program using the compiler optimization level `-O3` and refer the corresponding binary file to as 401.bip2_{O3}. We use Trin-Trin [16] to collect the dynamic call graph of the program subject to one of the reference inputs. Trin-Trin allows the identification of hot functions - refer to Table 1, and hot paths - refer to Table 2. Figure 2 shows a snapshot of the call graph concentrated in the proximity of the hot functions and including the hot paths. 401.bzip2 has a peaked profile [17] as it spends most of its execution time in the functions `blocksort.c:fallbackSort` and `compress.c:BZ2_compressBlock`. The dynamic call graph of the binary 401.bip2_{O3} is composed of 59 nodes - where a node represent a function, 69 edges, 2,454,278 directs call and 36 indirect calls. Given the limited number of hot functions and hot paths, we manually searched for opportunities for performance improvement coming from the reduction of the number of function calls. In Table 1, Table 2 and Figure 2 we list the name of the source file containing the implementation of a given function, along with the name of the function. We can distinguish three groups of

functions belonging to the source files `bzip2.c`, `bzlib.c` and `compress.c`. The function `BZ2_compressBlock` calls only the function `BZ_blockSort` with a single loop. The inlining of the latter function in the former would save 143 function calls, with an increase of the size of the caller of ≈ 16 kB. However, because these two functions appear in different source files (and modules), the decision to inlining `BZ_blockSort` in its caller is never evaluated by the inliner.¹

Fig. 2. 401.bzip2 call graph snapshot

Nevertheless, within the same source file (or module) there can be inlining decisions that the inliner misses because of inappropriate settings of its inlining parameters. For example, the function `blocksort.c:mainGtu` is a leaf function in the call graph that is called 1,515,464 times. The large call count stems from three different call sites embodied in three different loops. In this case there is a trade-off between increasing the size of the caller by roughly 1.65 kB and reducing the overhead of 1,515,464 function calls. This is a missed inlining opportunity for improving performance due to the default inlining settings.

The analysis that we carried out for 401.bzip2 would not have been possible for other programs from SPEC CINT2006, because of the large number of functions and files composing their source code. For example, the dynamic call graph of 403.gcc accounts 2,072 nodes, 7,868 edges, 216,947,768 function calls and a rather flat profile [17]. Such a complexity requires to approach our problem from different angles and with automatic methodologies that can learn from examples and are capable of predicting optimal inlining settings per program.

Lastly, we show that an obvious solution to tune inlining heuristics is not suitable to approach program optimization. For this, we study the variation in size and performance of 401.bzip2 with the increase of one inlining parameter ruling indiscriminate inlining, `max_inlining_insns_auto`. All the functions whose estimated cost - i.e., pseudo-instructions count [19] - is less than the value assigned to `max_inlining_insns_auto` are inlined. Therefore, the larger is the value assigned to `max_inlining_insns_auto`, the more functions are inlined, independently of the values of other inlining parameters.² One could potentially argue that the more functions calls are eliminated, the more performance of a program varies and improves. To show that this is not the case, we assigned

¹ In commercial and open source compilers the inliner works on the intermediate representation and so the inliner acts separately on single files or on a single module. The case of link-time optimization - e.g., GCC LTO [18] - is left to future work as we believe that our technique would speedup performance of GCC LTO.

² The list of inlining parameters of GCCv4.5 is at

<http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/Optimize-Options.html>

values from 10 to 190, with step 10, to the parameter `max_inlining_insns_auto` and compile 19 binary versions of `401.bzip2`. These 19 versions falls in four bins of different sizes 75 kB, 79 kB, 83 kB and 87 kB and not all the version of the binary files are unique. After 110 there is no variation in size, whereas after 190 the compiler exhibits malfunctioning. The performance of the binaries generated with different values of `max_inlining_insns_auto` as discussed above is slower ≈ 3 to 6% than performance achieved by the baseline `-O3`. Only in one case and for the first input there is a speedup of $\approx 2\%$.

3 Technique

In this section we first establish the background to build our technique, including the notations and the terminology used through the paper, and then we describe our technique to select inlining vectors for program optimization.

3.1 Notations

In this paper we denote an inlining vector as $\mathbf{iv}$, whereas we denote an inlining vector selected for optimizing a certain program as $\mathbf{iv}^*$. The set of all possible inlining vectors is denoted as F . While F is in principle an unbounded set - as each component of the inlining vector can assume any positive integer value, in practice the search for inlining vectors is confined into a subset of F that is practically bounded due to compiler malfunctioning. Any limited the subset of F that our technique explores to determine inlining vectors for program optimization is denoted as $\hat{F}$.

A machine learning algorithm is denoted as $\mathcal{A}$, whereas the model built with the algorithm $\mathcal{A}$ is denoted as $\hat{h}_{\alpha, \rho}$ - and is referred to as an hypothesis. An hypothesis is characterized by a number of parameters indicating its quality. As a measure of the quality of an hypothesis, in this work we use the mean absolute error - denoted as α , and the mean coefficient of correlation - denoted as ρ . The training set - that is the set of sample runs used to train our model - is denoted as T and is composed of pairs $(program_{\mathbf{iv}}, p)$, where several instances of programs subject to different inputs are compiled with a limited number of inlining vectors and executed. p is the completion time corresponding to a particular execution. During the execution of an instance of $program_{\mathbf{iv}}$, a vector of hardware performance counters is collected. Such a vector is denoted as $\mathbf{cnt}_{\mathbf{iv}}$ and features the run-time behavior of the program subject to an inlining vector. The set of $\mathbf{ivs}$ used to build the training set is within a limited subset of $\hat{F}$ and is denoted as $\mathcal{M}$.

3.2 Collection of the Training Set

The training set for building our model is populated as follows. A set of programs is compiled using $\mathcal{M}$ inlining vectors to generate an equal number binary files per program. A subset of these binary files are executed. A vector of hardware

performance counters and the completion time of each run is recorded. Each component of these inlining vectors assumes values in a sequence of integer values extracted from a limited range of integer values. These ranges are upper-bounded in practice by the maximum integer value that produces a valid binary file for a given program. Within each range we select a number of consecutive values uniformly spaced and the combinations of these values, from all the ranges, compose the set of $\mathbf{iv} \in \mathcal{M}$. However, not all these combinations of inlining parameters produce distinct binary files and amongst distinct binary files not all of these have distinct file sizes. In this work, distinct binary files with distinct sizes are referred to as *unique* files; distinct binary files with same sizes are referred to as *alias* files; other binary files that are identical byte-by-byte are referred to as *clone* files. To build the training set our technique first produces $\mathcal{M}$ binary files and second it dissects these files in *unique*, *alias* and *clone* files. While the process still involves a large number of compilations, such a dissection reduces the number of runs to the execution of only the *unique* files to populate the training set - *alias* files achieve nearly equal performance subject to the same workload. Therefore we decided not to include aliases in the training set.

The procedure to dissect the binary files in *unique*, *alias* and *clone* files is implemented as follows in our framework. The available binary files are scanned and a table indexed using the binary size is maintained. Each entry of the table is a tuple **<key, list of values>**, where the key is binary size and the list of values contains the names of alias and clone files. When a new binary is examined, its size is measured and used to access the table. If there is not an entry in the table for this key, then the binary is marked as *unique* and a new entry **<key, list of values>** is added to the table. If such an entry exists, the binary is appended to the list of values and marked as an *alias*. To test if the new binary is *unique*, it is compared byte-to-byte with the binary file corresponding to the first entry of the list.³ If the comparison is positive, the new binary is re-marked as a clone. The list of values is kept to provide statistics about the clone, alias and unique files, whereas the first element of each list is a unique binary that will be used to populate the training set. The training set is populated as follows: for each element in the first column of the table, execute the corresponding binary file and record a vector of hardware performance counters - a parameter of our framework - and the corresponding performance.

3.3 Hypothesis Selection

Once the training set is available, our technique builds hypotheses from different types of regression algorithms. Let us denote these algorithms as $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_S$. For example, $\mathcal{A}_1$ could represent a regression tree such as C4.5 [20], $\mathcal{A}_2$ could be support vector regression [21], etc. Each algorithm is trained using the training set to build one (or more) hypotheses. Each hypothesis is subsequently validated

³ Byte-to-byte comparison is a simple, yet convenient way to classify binary files as *unique*, *alias* and *clones*. Alternatively, a digest such as SHA-2 (Secure Hash Algorithm version 2) can be used for such a classification.

using 10-fold cross-validation, as discussed in the next Section and Appendix A, and is characterized with a coefficient of correlation and a mean absolute error. We denote the hypothesis associated to each algorithm as $\hat{h}_{\alpha_i, \rho_i}$, with $i = 1, 2, \dots, S$.

Given a threshold z - corresponding to the maximum acceptable mean absolute error - each hypothesis whose mean absolute error is greater than the z and whose coefficient of correlation is negative is rejected. In our technique, we assign a value to z according to the range of the outcomes of our experiments at the baseline - the baseline is a parameter of our framework. For example, let us assume -03 to be the baseline. If our $(program_{03}, input)$ pair runs for 500 seconds, a threshold tolerating an error of 1% on performance prediction would be set as $z = 5$ seconds. Therefore, all the hypotheses exhibiting an average mean absolute error lower than 5 seconds and exhibiting a positive coefficient of correlation are good candidates to model our experiments.

Amongst the remaining hypotheses, the one with maximum mean coefficient of correlation - and not necessarily with minimum mean absolute error - is selected.⁴

3.4 Model Validation

The evaluation of one or more regression algorithms is usually performed via a standard statistical technique called k -fold cross-validation [12]. In k -fold cross-validation, the training set is first partitioned into k nearly equally sized segments or folds. Subsequently k iterations of training and validation are performed such that within each iteration a different fold of the data is taken out for validation while the remaining $k - 1$ folds are used to train a regression model. At each iteration, one or more regression algorithms are trained using $k - 1$ folds of data to build one or more hypotheses. These hypotheses are subsequently used to make predictions using the features in the validation fold - refer to Appendix A for an example.

The performance of each learning algorithm on each fold can be assessed using performance metrics, such as the mean correlation coefficient and mean absolute error. At the end of the cross-validation process, k samples of each performance metric are available for each regression model. Different methodologies, such as the average of the k samples, can be used to obtain an aggregate measure from these samples, or these samples can be used in a statistical hypothesis test to show that one regression model is better to another.

In this paper we consider mean absolute error and mean coefficient of correlation as metrics to evaluate and to select a suitable regression model for our analysis. The former represents the average accuracy of performance prediction, whereas the latter represents the average correlation/alignment between a batch of performance predictions and the actual batch of performance - refer to Appendix A.

⁴ In the case of hypotheses with the same mean absolute error and/or the coefficient of correlation, a test of significance can be executed to select an hypothesis with the most significant mean [22].

3.5 Selection of an Inlining Vector

Without additional recompilations and runs, the hypothesis enables a rapid search of the space of inlining vectors and the determination of the best inlining vector to optimize performance of a given program subject to a given workload. Let us denote the selected hypothesis $\hat{h}^*$. The hypothesis predicts performance for unseen inlining vectors in $\hat{F}$. Therefore the hypothesis is used to search for the inlining vector that maximizes predicted performance and outperforms an assigned baseline. In other words our technique leverages the hypothesis to solve the following problem: select one inlining vector to minimize the completion time of a given combination (*program, input*). Eventually, our framework verifies the efficacy of the predicted inlining vector by recompiling the program using the inlining vector $\mathbf{iv}^*$ and measuring its performance on an average of several runs compared with the performance of the baseline.

4 Experiments

We implemented the technique presented in Section 3 as a performance framework written in Perl [23] and R [24–26]. The experiments we carried out to evaluate the efficacy of the proposed technique are presented in this section.

4.1 Experimental Setup

We used the state-of-the-art Intel Xeon Westmere architecture for our experiments - refer to Table 3. We evaluated the proposed technique optimizing 22 combinations (*program, input*) from SPEC CINT2006 - refer to Table 4. The combinations above belong to four programs selected from SPEC CINT2006 [13], because these programs have a larger number of input files, compared with other applications in the same benchmark suite, and because their behavior is highly influenced by the selection of a particular input file [14]. The components of the inlining vector for GNU GCC v4.5 and the ranges estimated by our technique are shown in Table 5.

To represent the run-time behavior of a binary compiled with a certain inlining vector, we select a vector of hardware counters - **cnt** - composed of the following components:

Table 3. System configuration

Model	Intel X5680 (Westmere) @ 3.33GHz
L1 I/D cache [kB]	32
L2 cache [kB]	256
L3 cache (shared) [MB]	12
Main Memory [GB]	24
Compiler	GNU GCC 4.5
Baseline optimization level	-O3
Operating system	Linux Red Hat AS 4 update 7

Table 4. Combinations (*program, input*)

Program name	Application domain	Input
401.bzip2	Compression	I1:chicken.jpg I2:control I3:input.source I4:liberty.jpg I5:text.html
403.gcc	C Language Optimizing Compiler	I1:166.i I2:200.i I3:c-typeck.i I4:cpdecl.i I5:expr.i I6:expr2.i I7:g23.i I8:s04.i I9:scilab.i
445.gobmk	Artificial intelligence	I1:13x13.tst I2:nngn.tst I3:score2.tst I4:trevorc.tst I5:trevord.tst
464.h264ref	Video compression	I1:foreman_ref_encoder_baseline.cfg I2:foreman_ref_encoder_main.cfg I3:sss_encoder_main.cfg

Table 5. Practical search space and $\hat{F}$ for our programs

iv component	Range
inline_call_cost	{10}
max_inline_insns_auto	[10-190]
large_function_insns	[1100-3100]
large_function_growth	[20-100]
large_unit_insns	[6,000-16,000]
inline_unit_growth	[30-300]
inline_recursive_depth	[4-8]

- **CPI** : Cycles per instruction.
- **Br rate** [%] : Number of branch retired as a percentage of the total instructions retired.
- **L2 miss** [%] : The count of L2 cache misses per thousand instructions.
- **L3 miss** [%] : The count of L3 cache misses per thousand instructions.

While the second component of the vector of counters, i.e., **Br rate**, includes the variation in the number of function calls due to a different amount of inlining, the other components indicates indirectly the influence of global optimizations to local performance, i.e., **CPI**, and the memory hierarchy behavior, i.e., **L2 and L3 miss**. Admittedly, the vector of hardware counters selected in this study may be complemented with other counters aimed to capture more specific run-time features. However, for the architecture considered in this paper, the counters above can be collected in a single run. This alleviates the problem(s) which may arise due to various sources of inaccuracies in the process of collection of the hardware performance counters [27]. Furthermore, the counters are normalized to instructions retired with different scale factors - % in the case of branch retired and % in the case of the cache misses - because for ordinary programs,

Fig. 3. Binary files dissection in *unique*, *alias* and *clones*

the cache miss count is usually one order of magnitude lower than the count of other hardware events [28]. We use Intel VTune [29] to collect **cnt**.

4.2 Inlining Parameter Ranges and the Set $\hat{F}$

We compile $\approx 30,000$ binary file per program - the size of $\mathcal{M} \approx 30,000$ - and dissect the number of unique, clone and alias files as it is shown in Figure 3.

In our technique, the set of inlining vectors used to build the training set is limited to the unique and not alias binary files for each program to optimize - $\approx 26,000$ unique binary files in total, corresponding to $\approx 20,000$ inlining vectors. However, inlining vectors for program optimization are determined in a larger set $\hat{F}$ composed of $> 10^{15}$ inlining vectors - refer to Table 5.

4.3 Baseline

In our experiments we set `-O3` as the baseline performance for our framework. Our framework compiles the programs using `-O3` and runs them to assess the baseline performance for each reference input. **cnt** is collected for the baseline. In the program selected for our experiments, each reference input exercises a different and/or the same path of the call graph of a program, but in different ways that are captured by the values of the components of **cnt**. For example, $(403.gcc, I6) = \langle 1.24, 25.11, 1.98, 0.12 \rangle$ and $(403.gcc, I7) = \langle 1.42, 26.32, 2.89, 0.22 \rangle$. Furthermore, our framework provides the percentage of total cycles spent in program routines versus external library/system routines - referred as cycles breakdown. The cycles breakdown is an indication of opportunities for performance optimization exploitable at compile-time. Arguably, the more the program runs into external/system routines the less opportunities exist for performance optimization exploitable at compile-time. Our framework shows that the average number of cycles - average on the reference input - spent

within program routines ranges from 45% to 98%. It follows the presence of opportunities for program optimization for our set of programs.

4.4 Collection of the Training Set

Our framework compiles each program times the size of $\mathcal{M}$ - where size of $\mathcal{M} \approx 30,000$. According to the analysis explained in Section 3 and for each program, our framework separates unique binary files from the binary files compiled. Next, our framework populate the training set. For each combination ($program_{iv}, input$), the framework runs the unique binary files and collects (a) the completion time expressed in second, and (b) the vector **cnt**.

4.5 Model Selection

Once the training set is available, the framework selects the regression algorithm that best models our experimental setup.⁵ In this section we report the comparison between three regression algorithms - Least Median Square - **LeastMedSq** [30], Radial Basis Function network - **RBFnetwork** [31], and Random Forest for regression - M5P model tree [15]. These algorithms belong to three different types of regression algorithms. **LeastMedSq** builds a linear function as an hypothesis, whereas **RBFnetwork** builds a form of artificial neural network for regression, whereas M5P builds a form of random forests for regression. The framework compares **LeastMedSq**, **RBFnetwork** and M5P in terms of their mean absolute errors and coefficients of correlation - refer to Table 6. The model selection is favorable to the adoption of M5P, which is the algorithm exhibiting the lower prediction error and the maximum coefficient of correlation.

Table 6. Hypotheses comparison and selection

Learning algorithm	(α, ρ)
LeastMedSq	(0.27, 93.17)
RBFnetwork	(0.77, 38.67)
M5P	(0.99, 2.15)

4.6 iv^* Determination

After the model is selected, the framework leverages the model to find inlining vectors that minimizes the predicted completion time for each combination ($program_{03}, input$). The predicted inlining vectors are listed in Table 7. The predicted inlining vectors never correspond to that of the baseline or to any others present in the training set.

On the current architecture the components of the predicted inlining vectors exhibit the following properties (*trends*):

⁵ Our framework explores all the regression algorithms available in the default package Rweka [24] for the language R.

Table 7. iv^* determined per combination ($program, input$); $Speedup = \frac{p_{03}}{p_{iv^*}}$; $iv_{03} = < 10, 120, 2000, 300, 10000, 150, 8 >$

$(program, input)$	iv^*	p_{03}	s	p_{iv^*}	s	Speedup
(401.bzip2,I1)	<10, 125, 2043, 197, 10000, 103, 8>	40.92		39.63		1.03
(401.bzip2,I2)	<10, 142, 2042, 299, 10000, 115, 8>	363.00		354.24		1.02
(401.bzip2,I3)	<10, 134, 2021, 185, 10000, 73, 8>	116.60		116.60		1.00
(401.bzip2,I4)	<10, 132, 2032, 147, 10000, 64, 8>	64.98		63.47		1.02
(401.bzip2,I5)	<10, 144, 2079, 159, 10000, 128, 8>	141.75		140.27		1.01
(403.gcc,I1)	<10, 126, 2415, 297, 6000, 135, 8>	36.15		35.34		1.02
(403.gcc,I2)	<10, 165, 2405, 296, 6000, 66, 8>	48.21		47.43		1.02
(403.gcc,I3)	<10, 127, 2524, 46, 6000, 103, 8>	69.07		66.70		1.04
(403.gcc,I4)	<10, 167, 2697, 207, 6000, 105, 8>	43.52		39.94		1.09
(403.gcc,I5)	<10, 167, 2693, 33, 6000, 95, 8>	48.56		47.61		1.02
(403.gcc,I6)	<10, 168, 2684, 183, 6000, 145, 8>	67.98		65.50		1.04
(403.gcc,I7)	<10, 152, 2373, 298, 6000, 106, 8>	85.07		83.18		1.02
(403.gcc,I8)	<10, 150, 2637, 127, 6000, 149, 8>	86.73		84.13		1.03
(403.gcc,I9)	<10, 138, 2691, 299, 6000, 44, 8>	16.25		16.08		1.01
(445.gobmk,I1)	<10, 129, 2004, 179, 6000, 40, 8>	72.54		70.78		1.02
(445.gobmk,I2)	<10, 169, 2690, 285, 6000, 149, 8>	183.71		179.00		1.03
(445.gobmk,I3)	<10, 157, 2070, 199, 6000, 123, 8>	97.62		91.36		1.07
(445.gobmk,I4)	<10, 134, 2088, 164, 6000, 148, 8>	71.54		70.00		1.02
(445.gobmk,I5)	<10, 156, 2603, 296, 6000, 97, 8>	97.90		95.50		1.03
(464.h264ref,I1)	<10, 154, 2688, 297, 6000, 149, 8>	81.48		81.41		1.00
(464.h264ref,I2)	<10, 169, 2053, 295, 6000, 124, 8>	60.78		60.00		1.01
(464.h264ref,I3)	<10, 120, 2078, 271, 6000, 75, 8>	527.06		515.00		1.02

- **max_inline_insns_auto**: the predicted values suggest that inlining functions indiscriminately is never the best option. However, it is beneficial to assign a value to this parameter that is larger than the default value.
- **large_function_insns**: the predicted values suggest that the presence of larger functions (large in terms of instructions count after inlining) is beneficial to performance. Indeed, it potentially exposes more opportunities for global optimizations.
- **large_function_growth**: the predicted values limits the size of these functions imposing more restrictive constraints than the baseline.
- **large_unit_insns**: the predicted values separate the programs in two groups. In the first group there is 401.bzip2, where the default value corresponds to the predicted one. This indicates that this program is not sensitive to the variation of this parameter. The second group contains the other programs. The value of **large_unit_insns** is lower than that of the baseline suggesting that the inliner must exclude more functions from the inlining decisions to deliver better program performance.
- **large_unit_growth**: the predicted values suggest that the presence of smaller modules after compilation benefits performance on the target architecture.
- **inline_recursive_depth**: this parameter always assumes the default value, although there are recursive functions inlined within both 401.bzip2 or 403.gcc. This indicates the performance variation induced by varying this parameter is predicted to be negligible.

4.7 Performance Results

Our framework feeds the compiler with the inlining vectors in Table 7 and re-compiles the programs in Table 4. The compilation produces one binary file for each combinations (*program, input*). Hence, five versions of the program 401.bzip2, nine versions of the program 403.gcc, five versions of the program 445.gobmk and three versions of the program 464.h264ref are produced by our framework. Performance results - shown in Table 7 - exhibit modest yet positive speedups. This is in part due to the features of the architecture in use. Indeed, when compared with prior generations, Westmere has a deeper pipeline and large micro-architectural buffers to mask more miss-events; consistent chunks of the working set sizes fit into the large last level of cache; the presence of a cross-bar interconnection network, which replace the old front-side bus, provides fast access to main memory. Despite the massive presence of training examples that were not outperforming the baseline performance for each program, the proposed procedure for the hypothesis selection allows leveraging the information embedded in the model and selecting inlining vectors outperforming in both the best inlining vectors seen during the training phase and the baseline performance.

To support the performance results in Table 7 we repeated the execution of the baseline and the optimized program for > 50 times. Then we applied **t-test** [32] to evaluate the hypothesis of equality of the average execution times of the baseline and the predicted inlining vector. As a result, our statistical test rejects the null hypothesis with a confidence level of 95%. This, in turns assesses the significance of performance improvements from 2% to 9%. Furthermore, our technique never provides performance losses to any of the combinations (*program, input*) considered in this work.

5 Related Work

Prior studies acknowledge the potential that function inlining offers at compile time to other optimizations [2–4, 33], parallelization [5] and vectorization [6]. Despite its importance, most of the efforts of prior work on function inlining propose techniques to improve the accuracy of inlining decisions. Sheifler in [7] profile the cost of function calls and provide this information to the inliner. Dean and Chambers in [34] propose inlining trials. After inlining a call site, the caller is compiled and executed so to measure the profitability of the current inlining decisions. The inliner queries a database of inlining trials to decide to inlining a call site. This technique requires a compiler to be capable of compiling, execute and replay pieces of code. Hazelwood and Grove in [35] drive function inlining and specialization using call contexts. The technique proposed in this work focuses on the selection of inlining constraints, i.e., an inlining vector, to minimize the completion time of a program subject to a given workload on any architecture. Little attention has been paid in the past to investigate the influence of inlining constraints on performance of arbitrarily complex programs, i.e., composed of several functions and source files. In particular, the techniques referenced above do not account for the mutual influence among inlining decisions

during the compilation of a module. In this work, we deal explicitly with such a complex aspect of automatic inlining with the assistance of a machine learning model to select inlining vectors for program optimization.

Cavazos and O’Boyle in [9] presented a technique that uses genetic search to dynamically tune JikesRVM inlining heuristics and selectively inline frequently used methods of Java programs. Profiling of frequently used methods in programs with long uptime - e.g., application server, ensures in practice the convergence of the search within the uptime of the application. Cooper et al. in [10] extended the source-to-source C inliner proposed by Davidson and Holler in [1], introducing the notion of **condition string** - a vector of static and dynamic program properties designed to characterize the call sites of a program and to allow fine-grain inlining decisions. Selecting condition strings is as complex as selecting inlining vectors and specialized search methods may represent a solution to the problem. Hill climbing is used in [10] to explore the space of condition strings. The work in [10] is concerned on characterizing the space of condition strings rather than reducing the time of the search. However, the large number of compilations and runs involved in the search, for each program, shows that the direct application of specialized search techniques is not practical for optimize statically compiled programs. Furthermore, the technique proposed in [10] does not account for separate compilation, which further reduces attainable performance.

In contrast, the technique proposed in this work - that extends our prior work in [36] - deals with separate compilation and approaches the problem of selecting inlining constraints in two passes. First our technique selects and builds a performance prediction model using a small number of training examples and then uses the model to search the inlining vector which optimizes performance of a program.

Prior work [37–39] interested in predicting near-optimal compiler settings for program optimization explore only on-off compiler setting, including those enabling/disabling inlining, but do not account for the selection of inlining constraints that directly influence the behavior of the inliner. To the best of our knowledge this is the first paper investigating automatic selection of inlining constraints, given a compiler and its parameterized heuristic. Differently from prior work utilizing machine learning algorithms to predict near-optimal on-off compiler settings [40–45, 38, 46, 39] our technique leaves the selection of the model unspecified until the training set is available and automatically selects the model using the rigorous and quantitative procedure described in Sections 3.3 and 3.4.

6 Conclusion

We proposed a new technique to determine inlining vectors for program optimization. Our technique uses regression algorithms to learn the relation between inlining vectors and completion time. Such a relation is subsequently used to predict inlining vectors able to optimize programs subject to a given workload and targeting a given architecture.

Key highlights of our technique are as follows: (1) The selection of the regression algorithm is unspecified until the training set is available and many algorithms are evaluated; (2) The rigorous selection of the most suitable regression algorithm follows a procedure based 10-fold cross-validation; (3) Our technique is suitable to be used in industry settings as a support for production compilers. In such settings, applications are routinely tested, hence the training data required to train our technique is often times already available, whereas application performance is often times optimized for specific classes of workloads rather than for any workload.

The effectiveness of our technique is shown by the experimental results with GNU GCC. Our technique was able to select inlining vectors to optimize 22 combinations (*program, input*) on the state-of-the-art Intel Xeon Westmere. Compared with the optimization level `-O3` and its default inlining vector, our technique yielded performance improvements ranging from 2% to 9%.

Acknowledgments. This work was partially supported by Yahoo! Inc, the NSF grant number CCF-0811882 and the NSF Variability Expedition Grant number CCF-1029783. The authors would like to thank the anonymous reviewers for their valuable comments.

References

1. Davidson, J.W., Holler, A.M.: A study of a C function inliner. *Software: Practice and Experience* 18(8) (1988)
2. Ball, J.E.: Predicting the effects of optimization on a procedure body. In: *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction* (1979)
3. Cooper, K.D., Hall, M.W., Torczon, L.: An experiment with inline substitution. *Softw. Pract. Exper.* 21(6) (May 1991)
4. Cooper, K.D., Hall, M.W., Torczon, L.: Unexpected side effects of inline substitution: a case study. *ACM Lett. Program. Lang. Syst.* 1(1) (March 1992)
5. Guo, J., Stiles, M., Yi, Q., Psarris, K.: Enhancing the role of inlining in effective interprocedural parallelization. In: *Proceedings of the International Conference on Parallel Processing* (2011)
6. Allen, R., Johnson, S.: Compiling C for vectorization, parallelization, and inline expansion. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1988)
7. Scheifler, R.W.: An analysis of inline substitution for a structured programming language. *Commun. ACM* 20(9) (September 1977)
8. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman (1979)
9. Cavazos, J., O'Boyle, M.F.P.: Automatic tuning of inlining heuristics. In: *Proceedings of the ACM/IEEE Conference on Supercomputing* (2005)
10. Cooper, K.D., Harvey, T.J., Waterman, T.: An Adaptive Strategy for Inline Substitution. In: Hendren, L. (ed.) *CC 2008. LNCS*, vol. 4959, pp. 69–84. Springer, Heidelberg (2008)

11. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.* 44(2) (January 2005)
12. Stone, M.: Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society B* 36(1), 111–147 (1974)
13. Henning, J.L.: Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34(4) (September 2006)
14. Berube, P., Amaral, J.M.: Aestimo: a feedback-directed optimization evaluation tool. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2006)
15. Quinlan, R.J.: Learning with continuous classes. In: *5th Australian Joint Conference on Artificial Intelligence* (1992)
16. Jalan, R., Kejariwal, A.: Trin-trin: Who's calling? a pin-based dynamic call graph extraction framework. *International Journal of Parallel Programming* 40(4) (2012)
17. Weicker, R.P., Henning, J.L.: Subroutine profiling results for the cpu2006 benchmarks. *SIGARCH Comput. Archit. News* 35(1) (March 2007)
18. Glek, T., Hubicka, J.: Optimizing real-world applications with gcc link time optimization. In: *GCC Developers Summit* (2010)
19. Hubicka, J.: Interprocedural optimization framework in GCC. In: *GCC Developers Summit* (2007)
20. Quinlan, J.R.: *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco (1993)
21. Smola, A.J., Schölkopf, B.: A tutorial on support vector regression. *Statistics and Computing* 14(3) (August 2004)
22. Siegel, S., Castellan, N.J.: *Nonparametric Statistics for the Behavioral Sciences*, 2nd edn. McGraw-Hill (1988)
23. Wall, L., Christiansen, T., Orwant, J.: *Programming Perl*, 3rd edn. O'Reilly Media (2000)
24. Hornik, K., Buchta, C., Zeileis, A.: Open-source machine learning: R meets Weka. *Computational Statistics* 24(2) (2009)
25. Williams, G.J.: Rattle: A data mining GUI for R. *The R Journal* 1(2) (December 2009)
26. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria (2011)
27. Zapanu, D., Jovic, M., Hauswirth, M.: Accuracy of performance counter measurements. In: *IEEE International Symposium on Performance Analysis of Systems and Software* (2009)
28. Levinthal, D.: *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors* (2009)
29. Reinders, J.: *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Engineer to Engineer Series. Intel Press (2005)
30. Rousseeuw, P.J., Leroy, A.M.: *Robust regression and outlier detection*. John Wiley & Sons, Inc. (1987)
31. Moody, J., Darken, C.J.: Fast learning in networks of locally-tuned processing units. *Neural Comput.* 1(2) (1989)
32. Student: The Probable Error of a Mean. *Biometrika* 6(1) (1908)
33. Ayers, A., Schooler, R., Gottlieb, R.: Aggressive inlining. In: *Proceedings of the ACM Conference on Programming Language Design and Implementation* (1997)

34. Dean, J., Chambers, C.: Towards better inlining decisions using inlining trials. In: Proceedings of the ACM Conference on LISP and Functional Programming (1994)
35. Hazelwood, K., Grove, D.: Adaptive online context-sensitive inlining. In: Proceedings of the International Symposium on Code Generation and Optimization (2003)
36. Cammarota, R., Kejariwal, A., Donato, D., Nicolau, A., Veidenbaum, A.V.: Selective search of inlining vectors for program optimization. In: Proceedings of the 9th Conference on Computing Frontiers (2012)
37. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (2006)
38. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of the International Symposium on Code Generation and Optimization (2007)
39. Fursin, G., Temam, O.: Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.* 7(4) (December 2010)
40. Monsifrot, A., Bodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Scott, D. (ed.) *AIMSA 2002. LNCS (LNAI)*, vol. 2443, pp. 41–50. Springer, Heidelberg (2002)
41. Stephenson, M., Amarasinghe, S., Martin, M., O'Reilly, U.: Meta optimization: improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (2003)
42. Pan, Z., Eigenmann, R.: Rating compiler optimizations for automatic performance tuning. In: Proceedings of the ACM/IEEE Conference on Supercomputing (2004)
43. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Optimizing general purpose compiler optimization. In: Proceedings of the 2nd Conference on Computing Frontiers (2005)
44. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of the International Symposium on Code Generation and Optimization (2005)
45. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: On the impact of data input sets on statistical compiler tuning. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing (2006)
46. Dubach, C., Jones, T.M., Bonilla, E.V., Fursin, G., O'Boyle, M.F.P.: Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In: Proceedings of the IEEE/ACM International Symposium on Microarchitecture (2009)
47. Pearson, K.: On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *Philosophical Magazine Series 5* 50 (1900)
48. Pearson, K.: Notes on the History of Correlation. *Biometrika* 13(1) (October 1920)
49. Rodgers, J.L., Nicewander, A.W.: Thirteen ways to look at the correlation coefficient. *The American Statistician* 42 (1988)

APPENDIX A - Example of Cross-Validation

Let us assume that our training set is composed of 1000 (size of $\mathcal{M}$) samples and that we want to compare the two hypotheses $\hat{h}_I$ and $\hat{h}_{II}$ corresponding to the regression algorithms I and II respectively. We carry out k -fold cross-validation using $k = 10$. We divide our training set into 10 folds. Each fold is composed of 100 examples. The outcomes of the features in the fold j are arranged as a vector as in Equation 1.

$$\mathbf{p}_j = (p_{j,1}; \dots; p_{j,100}), \text{ with } j = 1, 2, \dots, 10 \quad (1)$$

To compare the two hypotheses at the pass j of the procedure, we take the fold j out of the training set, train both the models I and II with the remaining 9 folds and use the corresponding $\hat{h}_{I,j}$ and $\hat{h}_{II,j}$ to predict the outcomes corresponding to the (*unseen*) features in the fold j - see Equation 2.

$$\hat{\mathbf{p}}_j = (\hat{p}_{j,1}; \dots; \hat{p}_{j,100}), \text{ with } j = 1, 2, \dots, 10 \quad (2)$$

The mean absolute error between $\mathbf{p}_j$ and $\hat{\mathbf{p}}_j$ is computed as in Equation 3, whereas the Pearsons coefficient of correlation [47–49] between $\mathbf{p}_j$ and $\hat{\mathbf{p}}_j$ is computed as in Equation 4.

$$\alpha_j = \frac{|p_{j,1} - \hat{p}_{j,1}| + \dots + |p_{j,100} - \hat{p}_{j,100}|}{100} \quad (3)$$

$$\rho_j = \frac{\sum_{s=1}^{100} (p_{j,s} - \mu_j)(\hat{p}_{j,s} - \hat{\mu}_j)}{\sqrt{\sum_{s=1}^{100} (p_{j,s} - \mu_j)^2 \times \sum_{s=1}^{100} (\hat{p}_{j,s} - \hat{\mu}_j)^2}} \quad (4)$$

where

$$\mu_j = \frac{\sum_{s=1}^{100} p_{j,s}}{100} \quad \text{and} \quad \hat{\mu}_j = \frac{\sum_{s=1}^{100} \hat{p}_{j,s}}{100}$$

At the end of the cross-validation process we obtain a sequence of 10 mean absolute errors and a sequence of 10 coefficients of correlation. The two models, I and II , are compared using the average values of the mean absolute errors and of the coefficients of correlation as indicated in Equations 5 and 6.

$$\mu_I^\alpha = \frac{\sum_{s=1}^{10} \alpha_{j,s}}{10} \quad \text{and} \quad \mu_I^\rho = \frac{\sum_{s=1}^{10} \rho_{j,s}}{10} \quad (5)$$

$$\mu_{II}^\alpha = \frac{\sum_{s=1}^{10} \alpha_{j,s}}{10} \quad \text{and} \quad \mu_{II}^\rho = \frac{\sum_{s=1}^{10} \rho_{j,s}}{10} \quad (6)$$

By definition $\alpha > 0$ and $|\rho| \leq 1$ and too are the quantities in Equations 5 and 6. Model selection occur as follows. Let assume, for example, that $\mu_I^\alpha < \mu_{II}^\alpha$ and $\mu_I^\rho > \mu_{II}^\rho > 0$. Both models have positive coefficients of correlation - indicating the presence of a linear relation between the real and the estimated values during cross-validation. However, the mean absolute error of the model I is lower than that of the model II . Consequently, one would select/prefer model I to model II .