The final publication is available at:
http://www.springerlink.com/content/d753r27550257252

Milepost GCC: machine learning enabled self-tuning compiler

Grigori Fursin[1,2] · **Yuriy Kashnikov**[2] · **Abdul Wahid Memon**[2] · **Zbigniew Chamski**[1] · **Olivier Temam**[1] · **Mircea Namolaru**[3] · **Elad Yom-Tov**[3] · **Bilha Mendelson**[3] · **Ayal Zaks**[3] · **Eric Courtois**[4] · **Francois Bodin**[4] · **Phil Barnard**[5] · **Elton Ashton**[5] · **Edwin Bonilla**[6] · **John Thomson**[6] · **Christopher K. I. Williams**[6] · **Michael O'Boyle**[6]

Received: 12 February 2009; Accepted: 30 November 2010

Contact: grigori.fursin@unidapt.org

Abstract Tuning compiler optimizations for rapidly evolving hardware makes porting and extending an optimizing compiler for each new platform extremely challenging. Iterative optimization is a popular approach to adapting programs to a new architecture automatically using feedback-directed compilation. However, the large number of evaluations required for each program has prevented iterative compilation from widespread take-up in production compilers. Machine learning has been proposed to tune optimizations across programs systematically but is currently limited to a few transformations, long training phases and critically lacks publicly released, stable tools.

Our approach is to develop a modular, extensible, self-tuning optimization infrastructure to automatically learn the best optimizations across multiple programs and architectures based on the correlation between program features, run-time behavior and optimizations. In this paper we describe Milepost GCC, the first publicly-available open-source machine learning-based compiler. It consists of an Interactive Compilation Interface (ICI) and plugins to extract program features and exchange optimization data with the cTuning.org open public repository. It automatically adapts the internal optimization heuristic at function-level granularity to improve execution time, code size and compilation time of a new program on a given architecture. Part of the Milepost technology together with low-level ICI-inspired plugin framework is now included in the mainline GCC.

We developed machine learning plugins based on probabilistic and transductive approaches to predict good combinations of optimizations. Our preliminary experimental results show that it is possible to automatically reduce the execution time of individual MiBench programs, some by more than a factor of 2, while also improving compilation

[1] INRIA Saclay, France (HiPEAC member) · [2] University of Versailles Saint Quentin en Yvelines, France · [3] IBM Haifa, Israel (HiPEAC member) · [4] CAPS Entreprise, France (HiPEAC member) · [5] ARC International, UK · [6] University of Edinburgh, UK (HiPEAC member) ·

time and code size. On average we are able to reduce the execution time of the MiBench benchmark suite by 11% for the ARC reconfigurable processor. We also present a realistic multi-objective optimization scenario for Berkeley DB library using Milepost GCC and improve execution time by approximately 17%, while reducing compilation time and code size by 12% and 7% respectively on Intel Xeon processor.

Keywords machine learning compiler · self-tuning compiler · adaptive compiler · automatic performance tuning · machine learning · program characterization · program features · collective optimization · continuous optimization · multi-objective optimization · empirical performance tuning · optimization repository · iterative compilation · feedback-directed compilation · adaptive compilation · optimization prediction · portable optimization

1 Introduction

Designers of new processor architectures attempt to bring higher performance and lower power across a wide range of programs while keeping time to market as short as possible. However, static compilers fail to deliver satisfactory levels of performance as they cannot keep pace with the rate of change in hardware evolution. Fixed heuristics based on simplistic hardware models and lack of run-time information means that much manual retuning of the compiler is needed for each new hardware generation. Typical systems now have multiple heterogeneous reconfigurable cores making such manual compiler tuning increasingly infeasible.

The difficulty of achieving portable performance has led to empirical iterative compilation for statically compiled programs [22, 69, 30, 52, 29, 54, 31, 77, 37, 70, 46, 47, 36, 39, 28, 40], applying automatic compiler tuning based on feedback-directed compilation. Here the static optimization model of a compiler is replaced by an iterative search of the optimization space to empirically find the most profitable solutions that improve execution time, compilation time, code size, power and other metrics. Needing little or no knowledge of the current platform, this approach can adapt programs to any given architecture. This approach is currently used in library generators and adaptive tools [84, 63, 71, 14, 1, 6]. However it is generally limited to searching for combinations of global compiler optimization flags and tweaking a few fine-grain transformations within relatively narrow search spaces. The main barrier to its wider use is the currently excessive compilation and execution time needed in order to optimize each program. This prevents a wider adoption of iterative compilation for general purpose compilers.

Our approach to solve this problem is to use machine learning which has the potential of reusing knowledge across iterative compilation runs, gaining the benefits of iterative compilation while reducing the number of executions needed. The objective of the Milepost project [11] is to develop compiler technology that can automatically learn how to best optimize programs for configurable heterogeneous processors based on the correlation between program features, run-time behavior and optimizations. It also aims to dramatically reduce the time to market configurable or frequently evolving systems. Rather than developing a specialized compiler by hand for each configuration, Milepost aims to produce optimizing compilers automatically.

A key goal of the project is to make machine learning based compilation a realistic technology for general-purpose production compilers. Current approaches [65, 27, 72, 17, 26] are highly preliminary, limited to global compiler flags or a few transformations

considered in isolation. GCC was selected as the compiler infrastructure for Milepost as it is currently the most stable and robust open-source compiler. GCC is currently the only production compiler that supports more than 30 different architectures and has multiple aggressive optimizations making it a natural vehicle for our research. Each new version usually features new transformations and it may take months to adjust each optimization heuristic, if only to prevent performance degradation in any of the supported architectures. This further emphasizes the need for an automated approach.

We use the Interactive Compilation Interface (ICI) [8,49] that separates the optimization process from a particular production compiler. ICI is a plugin system that acts as a "middleware" interface between production compilers such as GCC and user-definable research plugins. ICI allowed us to add a program feature extraction module and to select arbitrary optimization passes in GCC. In the future, compiler independent ICI should help transfer Milepost technology to other compilers. We connected Milepost GCC to a public collective optimization database at cTuning.org [3,38,42]. This provides a wealth of continuously updated training data from multiple users and environments.

In this paper we present experimental results showing that it is possible to improve the performance of the well-known MiBench [45] benchmark suite automatically using iterative compilation and machine learning on several platforms including x86: Intel and AMD, and the ARC configurable core family. Using Milepost GCC, after a few weeks training, we were able to learn a model that automatically improves the execution time of some individual MiBench programs by a factor of more than 2 while improving the overall MiBench suite by 11% on reconfigurable ARC architecture, often without sacrificing code size or compilation time. Furthermore, our approach supports general multi-objective optimization where a user can choose to minimize not only execution time but also code size and compilation time.

This paper is organized as follows: this section provided motivation for our research and developments. It is followed by Section 2 describing the experimental setup used throughout the article. Section 3 describes how iterative compilation can deliver multi-objective optimization. Section 4 describes the overall Milepost collaborative infrastructure while Section 5 describes our machine learning techniques used to predict good optimizations based on program features and provides experimental evaluation. It is followed by the sections on related and future work.

2 Experimental setup

The tools, benchmarks, architectures and environment used throughout the article are briefly described in this section.

2.1 Compiler

We considered several compilers for our research and development including Open64 [12], LLVM [9], ROSE [16], Phoenix [15], GCC [7]. GCC was selected as it is a mature and popular open-source optimizing compiler that supports many languages, has a large community, is competitive with the best commercial compilers, and features a large number of program transformation techniques including advanced optimizations such as the polyhedral transformation framework (GRAPHITE) [78]. Furthermore, GCC is

the only extensible open-source optimizing compiler that supports more than 30 processor families. However, our developed techniques are not compiler dependent. We selected the latest GCC 4.4.x as the base for our machine-learning enabled self-tuning compiler.

2.2 Optimizations

There are approximately 100 flags available for tuning in the most recent version of GCC, all of which are considered by our framework. However, it is impossible to validate all possible combinations of optimizations due to their number. Since GCC has not been originally designed for iterative compilation it is not always possible to explore the entire optimization space by simply combining multiple compiler optimization flags, because some of them are initiated only with a given global GCC optimization level (-Os,-O1,-O2,-O3). We overcome this issue by selecting a global optimization level -O1 .. -O3 first and then either turning on a particular optimization through a corresponding flag `-f<optimization name>` or turning it off using `-fno-<optimization name>` flag. In some cases, certain combinations of compiler flags or passes cause the compiler to crash or produce incorrect program execution. We reduce the probability of such cases by comparing outputs of programs with reference outputs.

2.3 Platforms

We selected two general-purpose and one embedded processor for evaluation:

- *AMD* – a cluster of 16 AMD Opteron 2218, 2.6GHz, 4GB main memory, 2MB L2 cache, running Debian Linux Sid x64 with kernel 2.6.28.1 (provided by GRID5000 [44])
- *Intel* – a cluster of 16 Intel Xeon EM64T, 3GHz, 2GB main memory, 1MB L2 cache, running Debian Linux Sid x64 with kernel 2.6.28.1 (provided by GRID5000)
- *ARC* – FPGA implementation of the ARC 725D reconfigurable processor, 200MHz, 32KB L1 cache, running Linux ARC with kernel 2.4.29

We specifically selected platforms that have been in the market for some time but not outdated to allow a fair comparison of our optimization techniques with default compiler optimization heuristics that had been reasonably hand-tuned.

2.4 Benchmarks and experiments

We use both embedded and server processors so we selected MiBench/cBench [45,39, 38] benchmark suite for evaluation, covering a broad range of applications from simple embedded functions to larger desktop/server programs. Most of the benchmarks have been rewritten to be easily portable to different architectures; we use dataset 1 in all cases. We encountered problems while compiling 4 tiff programs on the *ARC* platform and hence used them only on *AMD* and *Intel* platforms.

We use OProfile [13] with hardware counters support to perform non intrusive function-level profiling during each run. This tool may introduce some overhead, so we execute each compiled program three times and averaged the execution and compilation time. In the future, we plan to use more statistically rigorous approaches [75,43]. For

this study, we selected the most time consuming function from each benchmark for further analysis and optimization. If a program has several hot functions depending on a dataset, we analyze and optimize them one by one and report separately. Analyzing the effects of interactions between multiple functions on optimization is left for future work.

2.5 Collective optimization database

All experimental results were recorded in the public Collective Optimization Database [3, 38, 42] at cTuning.org, allowing independent analysis of our results.

3 Motivation

This section shows that tuning optimization heuristics of an existing real-world compiler for multiple objectives such as execution time, code size and compilation time is a non-trivial task. We demonstrate that iterative compilation can effectively solve this problem, however often with excessive search costs that motivate the use of machine learning to mitigate the need for per-program iterative compilation and learn optimizations across programs based on their features.

3.1 Multi-objective empirical iterative optimization

Iterative compilation is a popular method to explore different optimizations by executing a given program on a given architecture and finding good solutions to improve program execution time and other characteristics based on empirical search.

We selected 88 program transformations of GCC known to influence performance, including inlining, unrolling, scheduling, register allocation, and constant propagation. We selected 1000 combinations of optimizations using a random search strategy with 50% probability to select each flag and either turn it on or off. We use this strategy to allow uniform unbiased exploration of unknown optimization search spaces. In order to validate the resulting diversity of program transformations, we checked that no two combinations of optimizations generated the same binary for any of the benchmarks using the MD5 checksum of the assembler code obtained through the `objdump -d` command. Occasionally, random selection of flags in GCC may result in an invalid code. In order to avoid such situations, we validated all generated combinations of optimizations by comparing the outputs of all benchmarks used in our study with the recorded outputs during reference runs when compiled with -O3 global optimization level.

Figure 1 shows the best execution time speedup achieved for each benchmark over the highest GCC optimization level (-O3) after 1000 iterations across 3 selected architectures. It confirms results from previous research on iterative compilation and demonstrates that it is possible to outperform GCC's highest default optimization level for most programs using random iterative search for good combinations of optimizations. Several benchmarks achieve more than 2 times speedup while on average we reached speedups of 1.33 and 1.4 for *Intel* and *AMD* respectively and a smaller

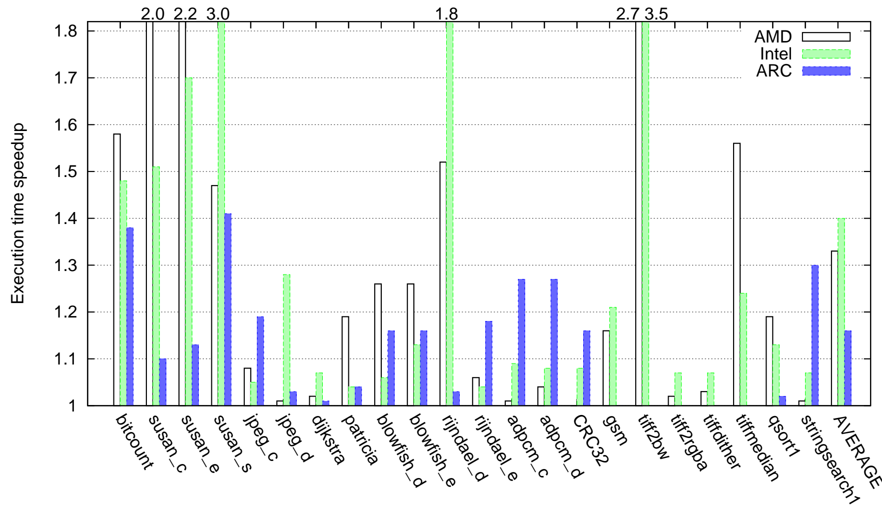

Fig. 1 *Maximum execution time speedups over the highest GCC optimization level (-O3) using iterative compilation with uniform random distribution after 1000 iterations on 3 selected architectures.*

speedup of 1.15 for *ARC*. This is likely due to simpler architecture and less sensitivity to program optimizations.

However, the task of an optimizing compiler is not only to improve execution time but also to balance code size and compilation time across a wide range of programs and architectures. The violin graphs [1] in Figure 2 show high variation of execution time speedups, code size improvements and compilation time speedups during iterative compilation across all benchmarks on *Intel* platform. Multi-objective optimization in such cases depend on end-user usage scenarios: improving both execution time and code size is often required for embedded applications, improving both compilation and execution time is important for data centers and real-time systems, while improving only execution time is common for desktops and supercomputers.

As an example, in Figure 3, we present the execution time speedups vs code size improvements and vs compilation time for *susan_c* on the *AMD* platform. Naturally, depending on optimization scenario, users are interested in optimization cases on the frontier of the program optimization area. Circles on these graphs show the 2D frontier that improves at least two metrics, while squares show optimization cases where the speedup is also achieved on the third optimization metric and is more than some threshold (compilation time speedup is more than 2 in the first graph and code size improvement is more than 1.2 in the second graph). These graphs demonstrate that for this selected benchmark and architecture there are relatively many optimization cases that improve execution time, code size and compilation time simultaneously. This is because many flags turned on for the default optimization level (-O3) do not influence this program or even degrade performance and take considerable compilation time.

[1] Violin graphs are similar to box graphs, showing the probability density in addition to min, max and interquartile.

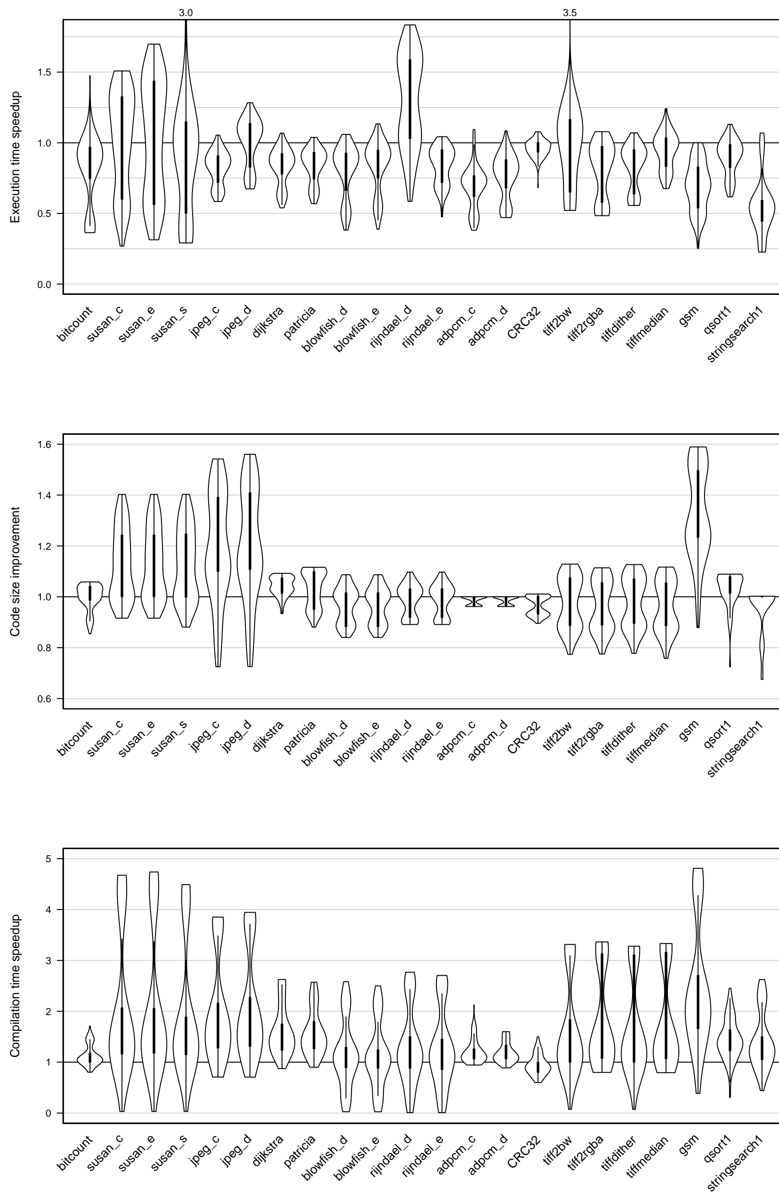

Fig. 2 Distribution of execution time speedups, code size improvements and compilation time speedups on *Intel* platform during iterative compilation (1000 iterations).

Figure 4 summarizes code size improvements and compilation time speedups achievable on *Intel* platform across evaluated programs with the execution time speedups within 95% of the maximum available during iterative compilation. We can observe that in some cases we can improve both execution time, code size and compilation

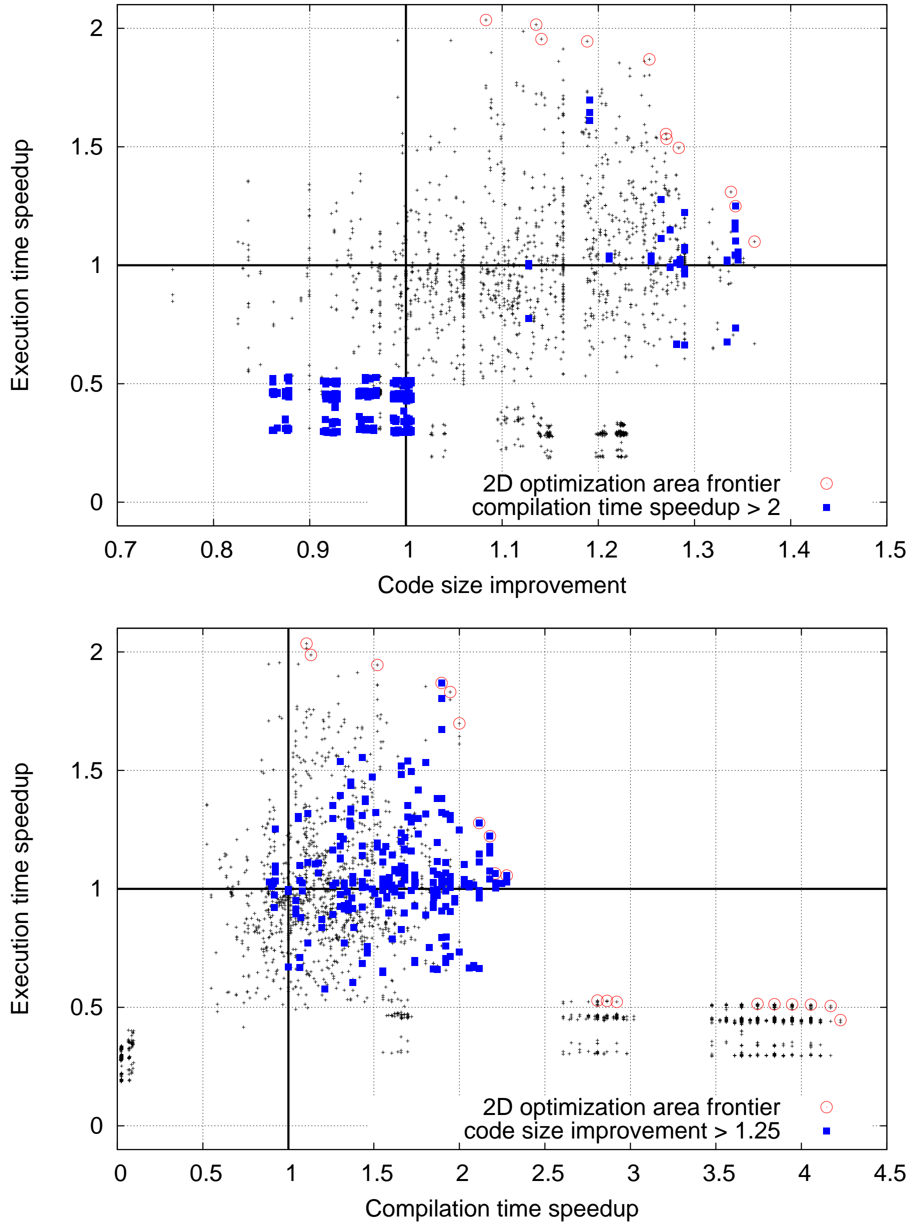

Fig. 3 Distribution of execution time speedups, code size improvements and compilation time speedups for benchmarks *susan_c* on *AMD* platform during iterative compilation. Depending on optimization scenarios, good optimization cases are depicted with circles on 2D optimization area frontier and with squares where third metric is more than some threshold (compilation time speedup > 2 or code size improvement > 1.2).

time such as for *susan_c* and *dijkstra* for example. In some other cases, without avoiding degradation of execution time for the default optimization level (-O3), we can

improve compilation time considerably (more than 1.7 times) and code size such as for *jpeg_c* and *patricia*. Throughout the rest of the article, we will consider improving execution time of primary importance, then code size and compilation time. However, our self-tuning compiler can work with other arbitrary optimization scenarios. Users may provide their own plugins to choose optimal solutions, for example using a Pareto distribution as shown in [46,47].

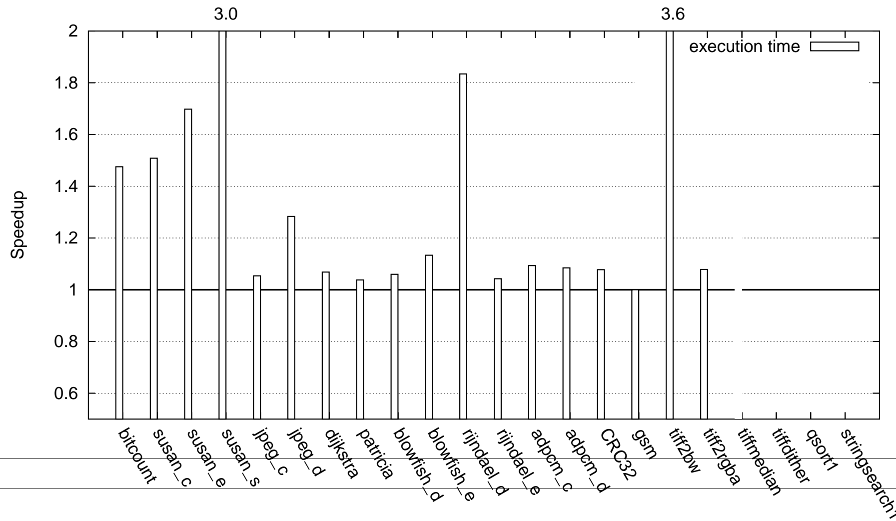

Fig. 4 Code size improvements and compilation time speedups for optimization cases with execution time speedups within 95% of the maximum available on *Intel* platform (as found by iterative compilation).

The combinations of flags corresponding to Figure 4 across all programs and architectures are presented [2] in Table 1. Some combinations can reduce compilation time by 70% which can be critical when compiling large-scale applications or for cloud computing services where a quick response time is critical. The diversity of compiler optimizations involved demonstrates that the compiler optimization space is not trivial and the compiler best optimization heuristic (-O3) is far from optimal. All combinations of flags found per program and architecture during this research are available on-line in the Collective Optimization Database [3] to allow end-users to optimize their programs or enable further collaborative research.

Finally, Figure 5 shows that it may take on average 70 iterations before reaching 95% of the speedup available after 1000 iterations (averaged over 10 repetitions) and is heavily dependent on the programs and architectures. Such a large number of iterations is needed due to an increasing number of aggressive optimizations available in the compiler where multiple combinations of optimizations can both considerably increase or decrease performance, change code size and compilation time.

[2] The flags that do not influence execution time, code size or compilation time have been iteratively and automatically removed from the original combination of random optimizations using CCC framework to simplify the analysis of the results.

-O1 -fcse-follow-jumps -fno-tree-ter -ftree-vectorize
-O1 -fno-cprop-registers -fno-dce -fno-move-loop-invariants -frename-registers -fno-tree-copy-prop -fno-tree-copyrename
-O1 -freorder-blocks -fschedule-insns -fno-tree-ccp -fno-tree-dominator-opts
-O2
-O2 -falign-loops -fno-cse-follow-jumps -fno-dce -fno-gcse-lm -fno-inline-functions-called-once -fno-schedule-insns2 -fno-tree-ccp -fno-tree-copyrename -funroll-all-loops
-O2 -finline-functions -fno-omit-frame-pointer -fschedule-insns -fno-split-ivs-in-unroller -fno-tree-sink -funroll-all-loops
-O2 -fno-align-jumps -fno-early-inlining -fno-gcse -fno-inline-functions-called-once -fno-move-loop-invariants -fschedule-insns -fno-tree-copyrename -fno-tree-loop-optimize -fno-tree-ter -fno-tree-vrp
-O2 -fno-caller-saves -fno-guess-branch-probability -fno-ira-share-spill-slots -fno-tree-reassoc -funroll-all-loops -fno-web
-O2 -fno-caller-saves -fno-ivopts -fno-reorder-blocks -fno-strict-overflow -funroll-all-loops
-O2 -fno-cprop-registers -fno-move-loop-invariants -fno-omit-frame-pointer -fpeel-loops
-O2 -fno-dce -fno-guess-branch-probability -fno-strict-overflow -fno-tree-dominator-opts -fno-tree-loop-optimize -fno-tree-reassoc -fno-tree-sink
-O2 -fno-ivopts -fpeel-loops -fschedule-insns
-O2 -fno-tree-loop-im -fno-tree-pre
-O3 -falign-loops -fno-caller-saves -fno-cprop-registers -fno-if-conversion -fno-ivopts -freorder-blocks-and-partition -fno-tree-pre -funroll-all-loops
-O3 -falign-loops -fno-cprop-registers -fno-if-conversion -fno-peephole2 -funroll-all-loops
-O3 -falign-loops -fno-delete-null-pointer-checks -fno-gcse-lm -fira-coalesce -floop-interchange -fsched2-use-superblocks -fno-tree-pre -fno-tree-vectorize -funroll-all-loops -funsafe-loop-optimizations -fno-web
-O3 -fno-gcse -floop-strip-mine -fno-move-loop-invariants -fno-predictive-commoning -ftracer
-O3 -fno-inline-functions-called-once -fno-regmove -frename-registers -fno-tree-copyrename
-O3 -fno-inline-functions -fno-move-loop-invariants

Table 1 Best found combinations of Milepost GCC flags to improve execution time, code size and compilation time after iterative compilation (1000 iterations) across all evaluated benchmarks and platforms.

The experimental results of this section suggest that iterative compilation can effectively generalize and automate the program optimization process but can be too time consuming. Hence it is important to speed up iterative compilation. In the next section, we present the Milepost framework which speeds up program optimization through machine learning.

4 Milepost optimization approach and framework

As shown in the previous section, iterative compilation can considerably outperform existing compilers but at the cost of excessive recompilation and program execution during optimization search space exploration. Multiple techniques have been proposed to speed up this process. For example, ACOVEA tool [1] utilizes genetic algorithms; hill-climbing search [37] and run-time function-level per-phase optimization evaluation [40] have been used, as well as the use of Pareto distribution [46, 47] to find multi-objective solutions. However, these approaches start their exploration of optimizations for a new program from scratch and do not reuse any prior optimization knowledge across different programs and architectures.

The Milepost project takes an orthogonal approach based on the observation that similar programs may exhibit similar behavior and require similar optimizations so it

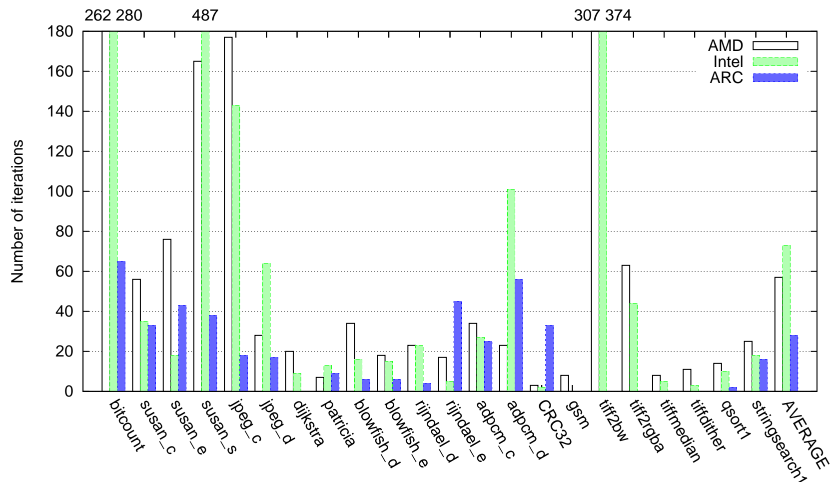

Fig. 5 *Number of iterations needed to obtain 95% of the available speedup using iterative compilation with uniform random distribution.*

is possible to correlate program features and optimizations, thereby predicting good transformations for unseen programs based on previous optimization experience [65, 27, 72, 17, 48, 26, 42]. In the current version of Milepost GCC we use static program features (such as the number of instructions in a method, number of branches, etc) to characterize programs and build predictive models. Naturally, since static features may not be enough to capture run-time program behavior, we plan to add plugins to improve program and optimization correlation based on dynamic features (performance counters [26], microarchitecture-independent characteristics [48], reactions to transformations [42] or semantically non-equivalent program modifications [41]).

The next section describes the overall framework and is followed by a detailed description of Milepost GCC and the Interactive Compiler Interface. This is then followed by a discussion of the features used to predict good optimizations.

4.1 Milepost adaptive optimization framework

The Milepost framework shown in Figure 6 uses a number of components including (i) a machine learning enabled Milepost GCC with Interactive Compilation Interface (ICI) to modify internal optimization decisions, (ii) a Continuous Collective Compilation Framework (CCC) to perform iterative search for good combinations of optimizations and (iii) a Collective Optimization Database (COD) to record compilation and execution statistics in the common repository. Such information is later used as training data for the machine learning models. We use public COD that is hosted at cTuning.org [3, 38, 42]. The Milepost framework currently proceeds in two distinct phases, in accordance with typical machine learning practice: training and deployment.

Training During the training phase we need to gather information about the structure of programs and record how they behave when compiled under different optimization

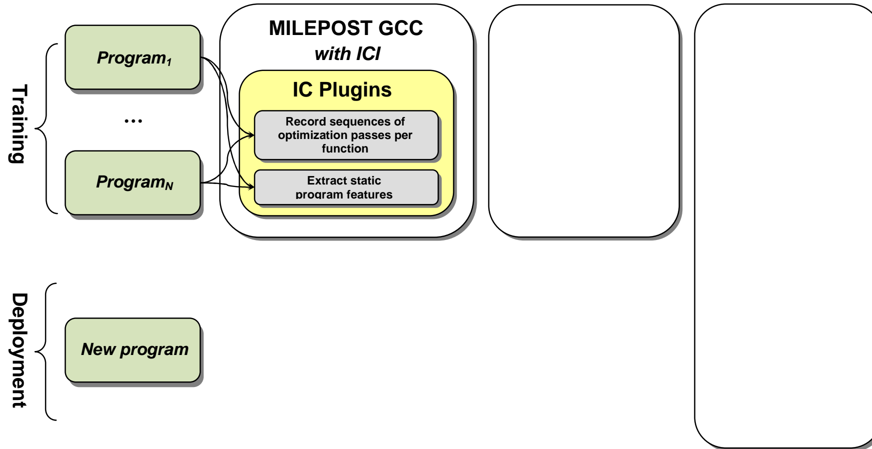

Fig. 6 Open framework to automatically tune programs and improve default optimization heuristics using predictive machine learning techniques, Milepost GCC with Interactive Compilation Interface (ICI) and program features extractor, CCC Framework to train ML model and predict good optimization passes, and COD optimization repository at cTuning.org.

settings. Such information allows machine learning tools to correlate aspects of program structure, or *features*, with optimizations, building a strategy that predicts good combinations of optimizations.

In order to train a useful model, a large number of compilations and executions are needed as training examples. These training examples are generated by CCC [2, 38], which evaluates different combinations of optimizations and stores execution time, profiling information, code size, compilation time and other metrics in a database. The features of the program are also extracted from Milepost GCC and stored in the COD. Plugins allow fine grained control and examination of the compiler, driven externally through shared libraries.

Deployment Once sufficient training data is gathered, multiple machine learning models can be created. Such models aim to correlate a given set of program features with profitable program transformations to predict good optimization strategies. They can later be re-inserted as plugins back to Milepost GCC or deployed as web-service at cTuning.org. The last method allows continuous update of the machine learning model based on collected information from multiple users. When encountering a new program, Milepost GCC determines the program's features and passes them to the model to predict the most profitable optimizations to improve execution time or other metrics depending on the user's optimization requirements.

4.2 Milepost GCC and Interactive Compilation Interface

Current production compilers often have fixed and black-box optimization heuristics without the means to fine-tune the application of transformations. This section describes the Interactive Compilation Interface (ICI) [49] which unveils a compiler and

provides opportunities for external control and examination of its optimization decisions with minimal changes. To avoid the pitfall of revealing intermediate representation and libraries of the compiler to a point where it would overspecify to many internals details and prevent further evolution, we choose to control the decision process itself, granting access only to the high-level features needed for effectively taking a decision. Optimization settings at a fine-grained level, beyond the capabilities of command line options or pragmas, can be managed through external shared libraries, leaving the compiler uncluttered. By replacing default optimization heuristics, execution time, code size and compilation time can be improved.

We decided to implement ICI for GCC and transform it into a research self-tuning compiler to provide a common stable extensible compiler infrastructure shared by both academia and industry, aiming to improve the quality, practicality and reproducibility of research, and make experimental results immediately useful to the community.

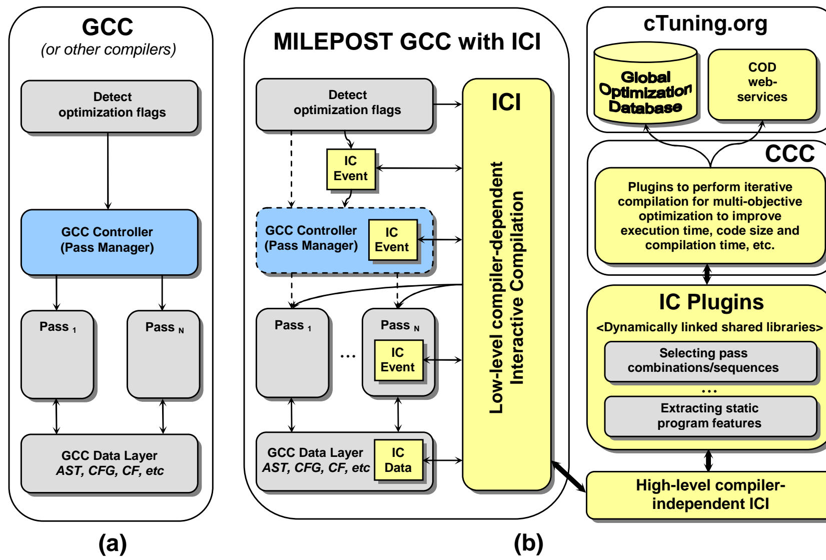

Fig. 7 GCC Interactive Compilation Interface: a) original GCC, b) Milepost GCC with ICI and plugins

The internal structure of ICI is shown in Figure 7. We separate ICI into two parts: low-level compiler-dependent and high-level compiler independent, the main reason being to keep high-level iterative compilation and machine learning plugins invariant when moving from one compiler to another. At the same time, since plugins now extend GCC through external shared libraries, experiments can be performed with no further modifications to the underlying compiler.

External plugins can transparently monitor execution of passes or replace the GCC Controller (Pass Manager), if desired. Passes can be selected by an external plugin which may choose to drive them in a very different order than that currently used in GCC, even choosing different pass orderings for each and every function in the

program being compiled. This mechanism simplifies the introduction of new analysis and optimization passes to the compiler.

In an additional set of enhancements, a coherent event and data passing mechanism enables external plugins to discover the state of the compiler and to be informed as it changes. At various points in the compilation process events (IC Event) are raised indicating decisions about transformations. Auxiliary data (IC Data) is registered if needed.

Using ICI, we can now substitute all default optimization heuristics with external optimization plugins to suggest an arbitrary combination of optimization passes during compilation without the need for any project or Makefile changes. Together with additional routines needed for machine learning, such as program feature extraction, our compiler infrastructure forms the Milepost GCC. We added a '-Oml' flag which calls a plugin to extract features, queries machine learning model plugins and substitutes the default optimization levels.

In this work, we do not investigate optimal orders of optimizations since that requires detailed information about dependencies between passes to detect legal orders; we plan to provide this information in the future. Hence, we examine the pass orders generated by compiler flags during iterative compilation and focus on selecting or deselecting appropriate passes that improve program execution time, compilation time or code size. In the future, we will focus on fine-grain parametric transformations in MILEPOST GCC [49] and combine them with the POET scripting language [86].

4.3 Static program features

Our machine learning models predict the best GCC optimization to apply to an input program based on its program structure or *program features*. The program features are typically a summary of the internal program representation and characterize essential aspects of a program that help to distinguish between good and bad optimizations.

The current version of ICI allows to invoke auxiliary passes that are not part of GCC's default compiler optimization heuristics. These passes can monitor and profile the compilation process or extract data structures needed for generating program features.

During compilation, a program is represented by several data structures, implementing the intermediate representation (tree-SSA, RTL etc), control flow graph (CFG), def-use chains, loop hierarchy, etc. The data structures available depend on the compilation pass currently being performed. For statistical machine learning, the information about these data structures is encoded in a constant size vector of numbers (i.e features) — this process is called *feature extraction* and facilitates reuse of optimization knowledge across different programs.

We implemented an additional *ml-feat* pass in GCC to extract static program features. This pass is not invoked during default compilation but can be called using an *extract_program_static_features* plugin after any arbitrary pass, when all data necessary to produce features is available.

In Milepost GCC, feature extraction is performed in two stages. In the first stage, a relational representation of the program is extracted; in the second stage, the vector of features is computed from this representation. In the first stage, the program is considered to be characterized by a number of entities and relations over these entities.

ft1	Number of basic blocks in the method
ft2	Number of basic blocks with a single successor
ft3	Number of basic blocks with two successors
ft4	Number of basic blocks with more then two successors
ft5	Number of basic blocks with a single predecessor
ft6	Number of basic blocks with two predecessors
ft7	Number of basic blocks with more then two predecessors
ft8	Number of basic blocks with a single predecessor and a single successor
ft9	Number of basic blocks with a single predecessor and two successors
ft10	Number of basic blocks with a two predecessors and one successor
ft11	Number of basic blocks with two successors and two predecessors
ft12	Number of basic blocks with more then two successors and more then two predecessors
ft13	Number of basic blocks with number of instructions less then 15
ft14	Number of basic blocks with number of instructions in the interval [15, 500]
ft15	Number of basic blocks with number of instructions greater then 500
ft16	Number of edges in the control flow graph
ft17	Number of critical edges in the control flow graph
ft18	Number of abnormal edges in the control flow graph
ft19	Number of direct calls in the method
ft20	Number of conditional branches in the method
ft21	Number of assignment instructions in the method
ft22	Number of binary integer operations in the method
ft23	Number of binary floating point operations in the method
ft24	Number of instructions in the method
ft25	Average of number of instructions in basic blocks
ft26	Average of number of phi-nodes at the beginning of a basic block
ft27	Average of arguments for a phi-node
ft28	Number of basic blocks with no phi nodes
ft29	Number of basic blocks with phi nodes in the interval [0, 3]
ft30	Number of basic blocks with more then 3 phi nodes
ft31	Number of basic block where total number of arguments for all phi-nodes is in greater then 5
ft32	Number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5]
ft33	Number of switch instructions in the method
ft34	Number of unary operations in the method
ft35	Number of instruction that do pointer arithmetic in the method
ft36	Number of indirect references via pointers ("*" in C)
ft37	Number of times the address of a variables is taken ("&" in C)
ft38	Number of times the address of a function is taken ("&" in C)
ft39	Number of indirect calls (i.e. done via pointers) in the method
ft40	Number of assignment instructions with the left operand an integer constant in the method
ft41	Number of binary operations with one of the operands an integer constant in the method
ft42	Number of calls with pointers as arguments
ft43	Number of calls with the number of arguments is greater then 4
ft44	Number of calls that return a pointer
ft45	Number of calls that return an integer
ft46	Number of occurrences of integer constant zero
ft47	Number of occurrences of 32-bit integer constants
ft48	Number of occurrences of integer constant one
ft49	Number of occurrences of 64-bit integer constants
ft50	Number of references of a local variables in the method
ft51	Number of references (def/use) of static/extern variables in the method
ft52	Number of local variables referred in the method
ft53	Number of static/extern variables referred in the method
ft54	Number of local variables that are pointers in the method
ft55	Number of static/extern variables that are pointers in the method
ft56	Number of unconditional branches in the method

Table 2 List of static program features currently available in Milepost GCC V2.1

The entities are a direct mapping of similar entities defined by the language reference, or generated during compilation. Such examples of entities are variables, types, instructions, basic blocks, temporary variables, etc.

A relation over a set of entities is a subset of their Cartesian product. The relations specify properties of the entities or the connections among them. We use a notation based on logic for describing the relations — Datalog is a Prolog-like language but with a simpler semantics, suitable for expressing relations and operations upon them [83, 79].

To extract the relational representation of the program, we used a simple method based on the examination of the *include* files. The main data structures of the compiler are built using *struct* data types, having a number of $fields$. Each such *struct* data type may introduce an entity, and its $fields$ may introduce relations over the entity, representing the including *struct* data type and the entity representing the data type of the $field$. This data is collected by the *ml-feat* pass.

In the second stage, we provide a Prolog program defining the features to be computed from the Datalog relational representation, extracted from the compiler's internal data structures in the first stage. The *extract_program_static _features* plugin invokes a Prolog compiler to execute this program, resulting in a vector of features (as shown in Table 2) which later serves to detect similarities between programs, build machine learning models and predict the best combinations of passes for new programs. We provide more details about aggregation of semantical program properties for machine learning based optimization in [68].

5 Using machine learning to predict good optimization passes

The Milepost approach to learning optimizations across programs is based on the observation that programs may exhibit similar behavior for a similar set of optimizations [17, 42], and hence we try to apply machine learning techniques to correlate their features with most profitable program optimizations. In this case, whenever we are given a new unseen program, we can search for similar programs within the training set and suggest good optimizations based on their optimization experience.

In order to test this assumption, we selected the combination of optimizations which yields the best performance for a given program on *AMD*, see `reference` in Figure 8. We then applied all these "best" combinations to all other programs and reported the performance difference, see `applied to`. It is possible to see that there is a fairly large amount of programs that share similar optimizations.

In the next subsections we introduce two machine learning techniques to select combinations of optimization passes based on construction of a *probabilistic model* or a *transductive model* on a set of M training programs, and then use these models to predict "good" combinations of optimization passes for unseen programs based on their features.

There are several differences between the two models: first, in our implementation the probabilistic model assumes each attribute is independent, whereas the proposed transductive model also analyzes interdependencies between attributes. Second, the probabilistic model finds the closest programs from the training set to the test program, whereas the transductive model attempts to generalize and identify good combinations of flags and program attributes. Therefore, it is expected that in some settings programs will benefit more from the probabilistic approach, whereas in others programs will

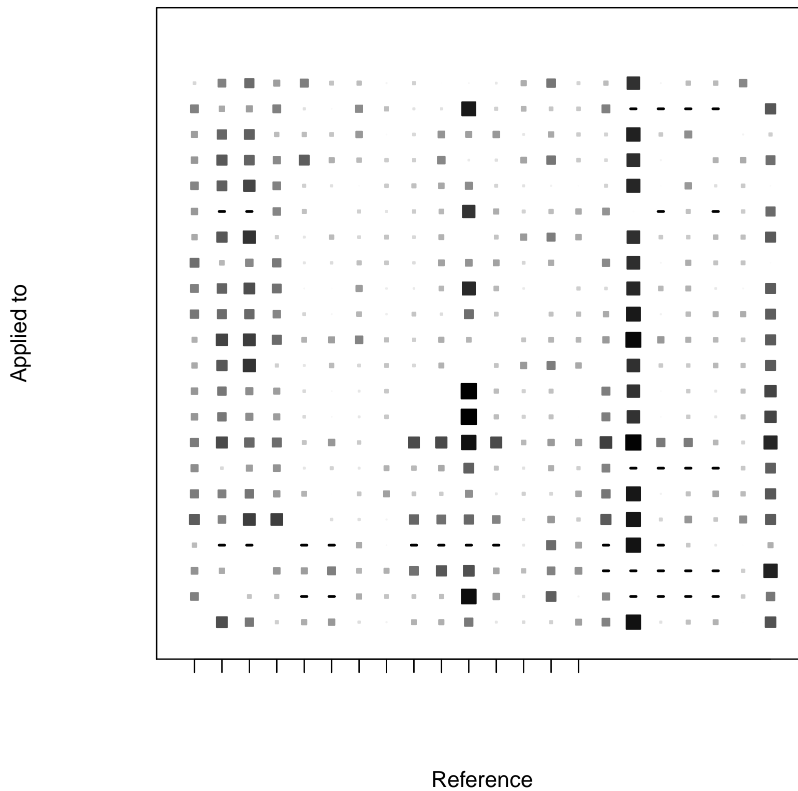

Fig. 8 % difference between speedup achievable after iterative compilation for *"applied to"* program and speedup obtained when applying best optimization from *"reference"* program to *"applied to"* program on *AMD.*"-" means that best optimization was not found for this program.

be improved more by using the transductive method, depending on training set size, number of samples of the program space, as well as program and architecture attributes.

In order to train the two machine learning models, we generated 1000 random combinations of flags turned either on or off as described in Section 3. Such a number of runs is small relative to the size of the optimization space yet it provides enough optimization cases and sufficient information to capture good optimization choices. The program features for each benchmark, the flag settings and execution times formed the training data for each model. All experiments were conducted using leave-one-out cross-validation. This means that for each of the N programs, the other $N-1$ programs are used as training data. This guarantees that each program is unseen when the model predicts good optimization settings to avoid bias.

5.1 Probabilistic machine learning model

Our probabilistic machine learning method is similar to that of [17] where a probability distribution over "good" solutions (i.e. optimization passes or compiler flags) is learnt across different programs. This approach has been referred to as Predictive Search Distributions (PSD) [23]. However, unlike prior work [17, 23] where such a distribution is used to focus the search of compiler optimizations on a new program, we use the learnt distribution to make *one-shot* predictions on unseen programs. Thus we do not search for the best optimization, we automatically predict it.

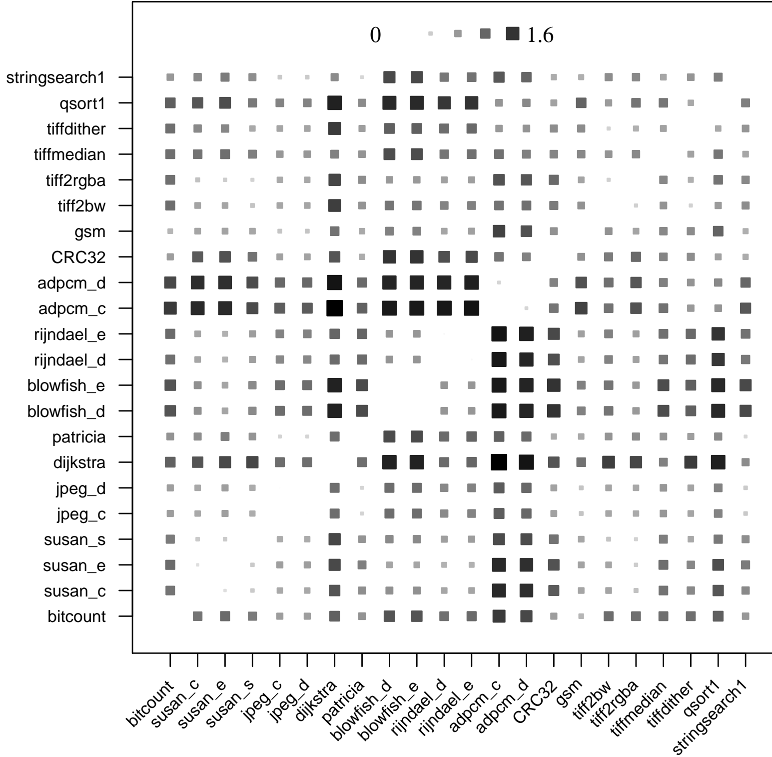

Fig. 9 Euclidean distance for all programs based on static program features normalized by feature 24 (number of instructions in a method).

Given a set of training programs $T^1, \ldots, T^M$, which can be described by feature vectors $\mathbf{t}^1 \ldots, \mathbf{t}^M$, and for which we have evaluated different combinations of optimization passes ($\mathbf{x}$) and their corresponding execution times (or speed-ups) y so that we have for each program T^j an associated dataset $\mathcal{D}^j = \{(\mathbf{x}^i, y^i)\}_{i=1}^{N^j}$, with $j = 1, \ldots M$, our goal is to predict a good combination of optimization passes $\mathbf{x}^*$ minimizing y^* when a new program T^* is presented.

We approach this problem by learning a mapping from the features of a program $\mathbf{t}$ to a *distribution over good solutions* $q(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters of the distribution. Once this distribution has been learnt, prediction for a new program T^* is straightforward and is achieved by sampling at the mode of the distribution. In other words, we obtain the predicted combination of flags by computing:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}}\, q(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta}). \tag{1}$$

In order to learn the model it is necessary to fit a distribution over good solutions to each training program beforehand. These solutions can be obtained, for example, by using uniform sampling or by running an estimation of distribution algorithm (EDA, see [55] for an overview) on each of the training programs. In our experiments we use uniform sampling and we choose the set of good solutions to be those optimization settings that achieve at least 98% of the maximum speed-up available in the corresponding program-dependent dataset.

Let us denote the distribution over good solutions on each training program by $P(\mathbf{x}|T^j)$ with $j = 1, \ldots, M$. In principle, these distributions can belong to any parametric family. However, in our experiments we use an IID model where each of the elements of the combination are considered independently. In other words, the probability of a "good" combination of passes is simply the product of each of the individual probabilities corresponding to how likely each pass is to belong to a good solution:

$$P(\mathbf{x}|T^j) = \prod_{\ell=1}^{L} P(x_\ell|T^j), \tag{2}$$

where L is the length of the combination.

Once the individual training distributions $P(\mathbf{x}|T^j)$ are obtained, the predictive distribution $q(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta})$ can be learnt by maximization of the conditional likelihood or by using *k-nearest neighbor* methods. In our experiments we use a 1-nearest neighbor approach (Figure 9 shows Euclidean distances between all programs with a visible clustering). In other words, we set the predictive distribution $q(\mathbf{x}|\mathbf{t}, \boldsymbol{\theta})$ to be the distribution corresponding to the training program that is closest in feature space to the new (test) program.

Figure 10 compares the speedups achieved after iterative compilation using 1000 iterations and 50% probability of selecting each optimization on *AMD* and *Intel* after one-shot prediction using probabilistic model or simply after selecting best combination of optimizations from the closest program. Interestingly, the results suggest that simply selecting best combination of optimizations from a similar program may not perform well in many cases; this may be due to our random optimization space exploration technique - each "good" combination of optimizations includes multiple flags that do not influence performance or other metrics on a given program, however some of them can considerably degrade performance on other programs. On the contrary, probabilistic approach helps to filter away non-influential flags statistically and thereby improve predictions.

5.2 Transductive machine learning model

In this subsection we describe a new transductive approach where optimization combinations themselves are used, as features for the learning algorithm, together with

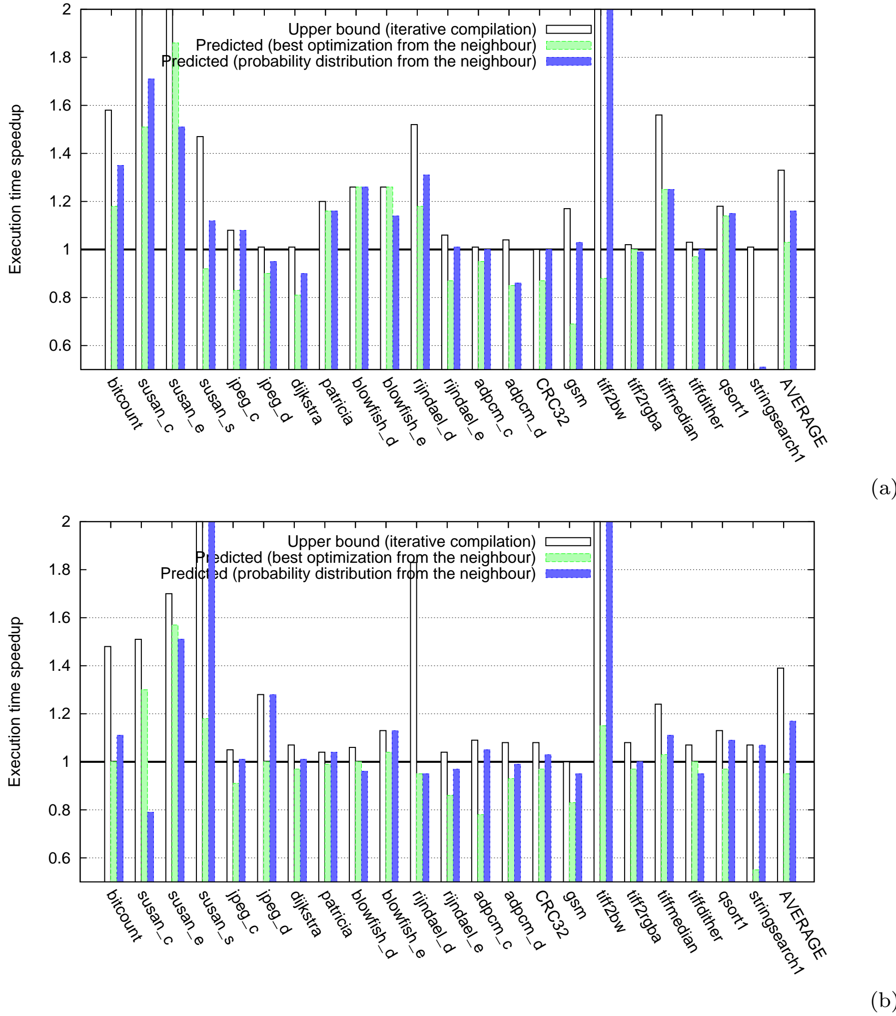

Fig. 10 Speedups achieved when using iterative compilation on (a) *AMD* and (b) *Intel* with random search strategy (1000 iterations; 50% probability to select each optimization;), when selecting best optimization from the nearest program and when predicting optimization using *probabilistic ML model* based on program features.

program features. The model is then queried for the best combination of optimizations out of the set of optimizations that the program was compiled with. Many learning algorithms can be used for building the ML model. In this work we used a decision tree model [34] to ease analysis of the resulting model.

As in the previous section, we try to predict whether a specific optimization combination will obtain at least 95% of the maximal speedup possible. The feature set is comprised of the flags/passes and the extracted program features, obtained from Milepost GCC. Denoting the vector of extracted features from the i-th program by $\mathbf{t}^i, i = 1, \ldots, M$ and the possible optimization passes by $\mathbf{x}^j, j = 1, \ldots, N$, we train the ML model with a set of features which is the cross-product of $\mathbf{x} \times \mathbf{t}$, such that

each feature vector is a concatenation of $\mathbf{x}^j$ and $\mathbf{t}^i$. This is akin to multi-class methods which rely on single binary classifiers (see [35] for a detailed discussion of such methods). The target for the predictor is whether this combination of program features and flags/passes combination will give a speedup of at least 95% of the maximal speedup.

Once a program is compiled with different optimization settings (either an exhaustive sample, or a random sample of optimization combinations), all successfully compiled program settings are used as a query for the learned model together with the program features, and the flag setting which is predicted to have the best speedup is used. If several settings are predicted to have the same speedup, the one which exhibited, on average, the best speedup with the training set programs, is used.

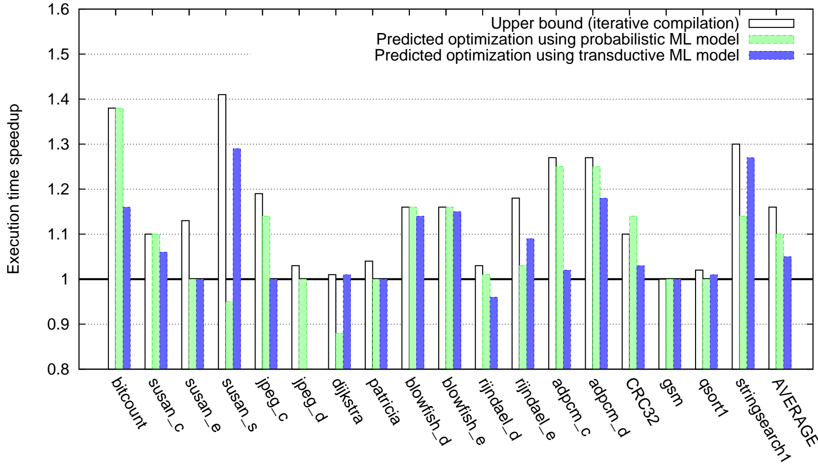

Fig. 11 Speedups achieved when using iterative compilation on *ARC* with random search strategy (1000 iterations; 50% probability to select each optimization;) and when predicting best optimizations using *probabilistic ML model* and *transductive ML model* based on program features

Figure 11 compares the speedups achieved after iterative compilation using 1000 iterations and 50% probability of selecting each optimization on *ARC* and after one-shot prediction using probabilistic and transductive models. It shows that our probabilistic model can automatically improve the default optimization heuristics of GCC by 11% on average while reaching 100% of the achievable speedup in some cases. On the other hand, transductive model improves GCC by only a modest 5%. However, in several cases it outperforms the probabilistic model: susan_s, dijkstra, rijndael_e, qsort1 and strinsearch1 likely due to a different mechanism of capturing the importance of program features and optimizations. Moreover, transductive (decision tree) model has an advantage that it is much easier to analyze the results. For example, Figure 12 shows the top levels of the decision trees learnt for *ARC*. The leafs indicate the probability that the optimization and program feature combinations which reached these nodes will be in the top 95% of the speedup for a benchmark. Most of these features found at the top level characterize the control flow graph (CFG). This is somehow expected, since the structure of the CFG is one of the major factors that may affect the efficiency of

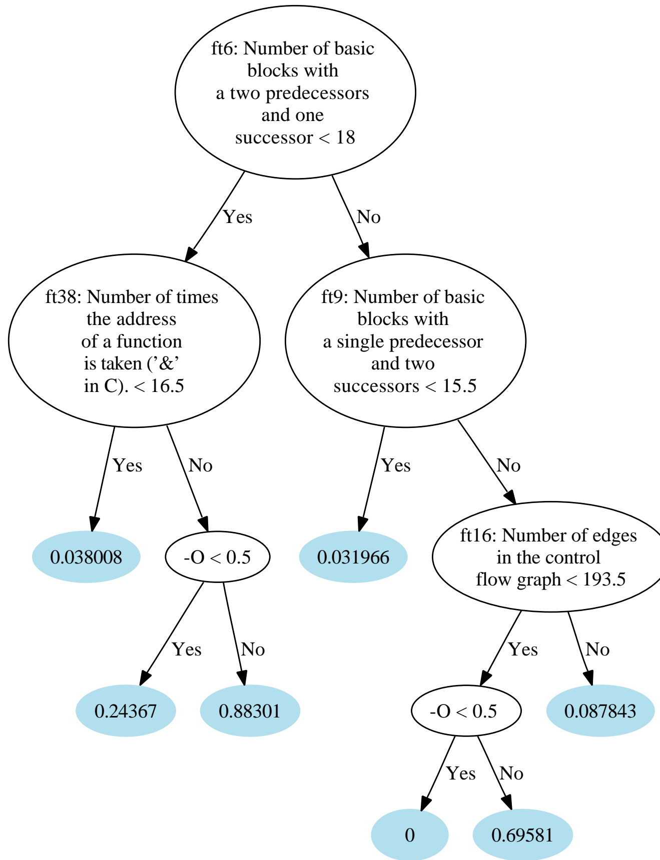

Fig. 12 Top levels of decision trees learnt for *ARC*.

several optimizations. Other features relate to the applicability of the "address-taken" operator to functions that may affect the accuracy of the call-graph and of subsequent analysis using it. To improve the performance of both models, we intend to analyze the quality and importance of program features and their correlation with optimizations in the future.

5.3 Realistic optimization scenario of a production application

Experimental results from the previous section show how to optimize several standard benchmarks using Milepost GCC. In this section we show how to optimize a real production application using Milepost technology combined with machine learning model from Section 5.1. For this purpose, we selected the open-source Berkeley DB library (BDB) which is a popular high-performance database written in C with APIs to most other languages. For evaluation purposes we used an official internal benchmarking suite and provided support of the CCC framework to perform iterative compilation in a same manner as described in Section 3, in order to find the upper bounds for execution time, code size and compilation time.

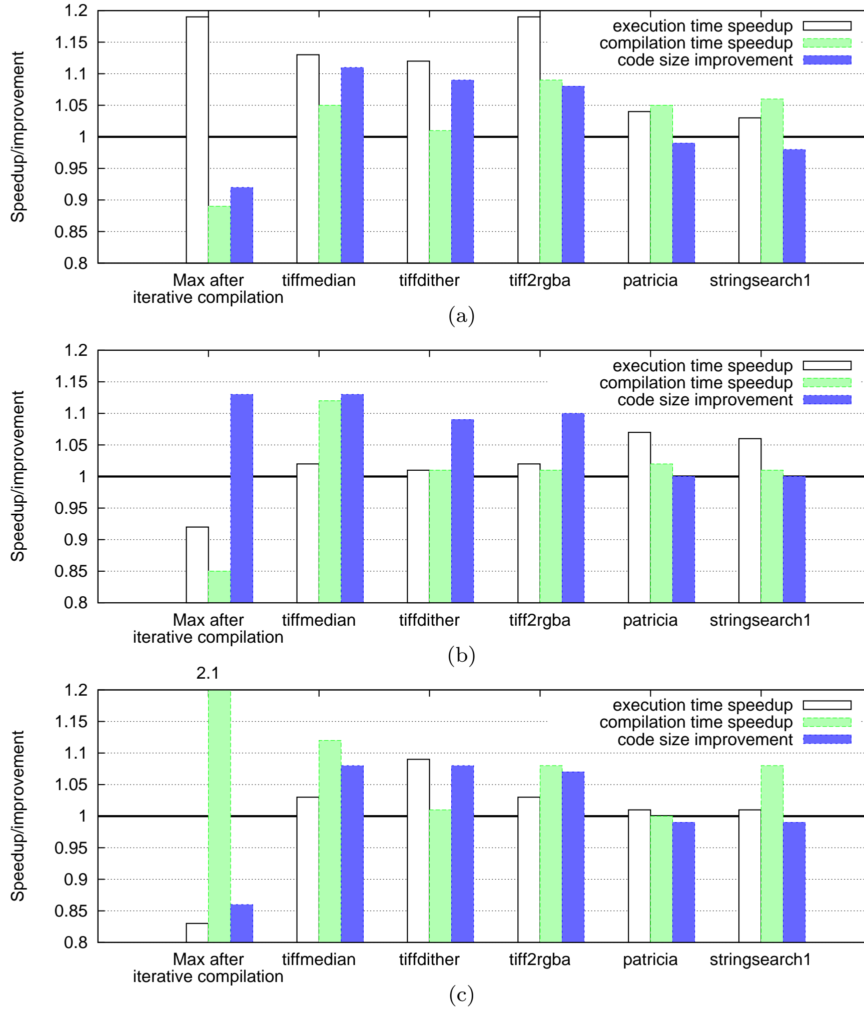

Fig. 13 *Execution time speedups (a), code size improvements (b) and compilation time speedup (c) for BerkeleyDB on* Intel *when applying optimizations from 5 closest programs from MiBench/cBench (based on Euclidean distance using static program features of 3 hottest functions) using several optimization scenarios.*

For simplicity, we decided to use a probabilistic machine learning model from Section 5.1. Since BDB is relatively large (around 200,000 lines of code) we selected the 3 hottest functions, extracted features for each function using Milepost GCC and calculated Euclidean distance with all programs from our training set (MiBench/cBench) to find the five most similar programs. Then, depending on the optimization scenario, we selected the best optimizations from those programs to (a) improve execution time while not degrading compilation time (b) improve code size while not degrading execution time and (c) improve compilation time while not degrading execution time. Figure 13 shows the achieved execution time speedups, code size improvements and compilation time speedups over -O3 optimization level when applying selected opti-

mizations from the most similar programs to BerkeleyDB for these three optimization scenarios. These speedups are compared to the upper bound for the respective metrics achieved after iterative compilation (200 iterations) for the whole program. The programs on the X-axis are sorted by distances starting from the closest program. In the case of improving execution time, we show significant speedup across the functions. For improving compilation time we are far from the optimal solution because it is naturally associated with the lowest optimization level, while we have been focusing also on not degrading execution time of -O3. Overall, the best results were achieved when applying optimizations from tiff programs that are closer in the feature space to the hot functions selected from BerkeleyDB, than any other program of the training set.

We added information about the best optimizations from these 3 optimization scenarios to the open online Collective Optimization Database [3] to help users and researchers validate and reproduce such results. These optimization cases are referenced by the following cTuning RUN_ID reference numbers: 24857532370695782, 17268781782733561 and 9072658980980875. The default run related to -O3 optimization level is referenced by 965827379437489142. We also added support for pragma `#ctuning-opt-case UID` that allows end-users to explicitly force Milepost GCC to connect combinations of optimizations found by other users during empirical collective search and referenced by UID in COD to a given code section instead of using machine learning.

6 Related work

Automatic performance tuning techniques are now widely adopted to improve different characteristics of a code empirically. They search automatically for good optimization settings, applying multiple compilations and executions of a given program while requiring little or no knowledge of the current platform, so programs can be adapted to any given architecture. Originally developed to improve performance of various small kernels using a few parametric transformations across multiple architectures, where static compilers fail to deliver best performance [84, 53, 63, 71, 81, 21, 85], these techniques have been extended to larger applications and richer set of optimizations [22, 52, 29, 54, 31, 77, 37, 32, 59, 70, 86, 6, 14, 78].

Though popular for library generators and embedded systems, iterative compilation is still not widely adopted by general purpose compilers mainly due to excessively long optimization time. Multiple genetic and probabilistic approaches have been developed to speed up optimization of a given program on a given architecture [69, 30, 73, 46, 19, 47, 36]. Furthermore, novel dynamic and hybrid (static and dynamic) adaptation techniques have been proposed to speed up evaluation of optimizations at run-time [80, 57]. In [40], we have shown the possibility to speed up iterative compilation by several orders of magnitude using static function cloning with pre-optimized versions for various objectives and run-time low-overhead optimization evaluation that also enabled adaptive binaries reactive to run-time changes in the program and environment. In [37, 41], we introduced a new technique to quickly detect realistic lower bound of the execution time of memory intensive applications by converting array accesses to scalars in various ways without preserving the semantics of the code to quickly detect performance anomalies and identify code sections that can benefit from empirical optimizations. All these techniques can effectively learn the optimization space of an individual program

and optimization process but they still do not learn optimizations across programs and architectures.

Calder et al. [25] presented a new approach to predict branches for a new program based on behavior of other programs. They used neural networks and decision trees to map static features associated with each branch to a prediction that the branch will be taken, and managed to slightly reduce the branch misprediction rate on a set of C and Fortran programs. Moss and McGovern et al. [67,64] incorporated a reinforcement learning model with a compiler to improve code scheduling, however no absolute performance improvements were reported. Monsifrot et al. [66] presented a classifier based on decision tree learning to determine which loops to unroll. Mark Stephenson and Saman Amarasinghe [72] also predict unroll factors using nearest neighbor classification and support vector machines. In our previous work [17,26] we used static or dynamic (performance counters) code features with SUIF, Intel and PathScale compilers to predict a set of multiple optimizations that improve execution time for new programs based on similarities between previously optimized programs. Liao et al. [82] used machine learning to performance counters and decision trees to choose hardware prefetcher configurations. Several researchers [24,58,62] attempted to characterize program input in order to predict best code variant at run-time using several machine learning methods, including automatically generated decision trees and statistical modeling. Other works [50,48,33] used machine learning for performance prediction and hardware-software co-design.

Though machine learning techniques demonstrate a good potential to speed up the iterative compilation process and facilitate reuse of optimization knowledge across different programs and architectures, the training phase can still be very long. Techniques for continuous optimization can effectively speed up training of machine learning models. Anderson et al. [18] presented a practical framework for continuous and transparent profiling and analysis of computing systems, though unfortunately this work did not continue and no machine learning has been used. Lattner and Adve [56] and Lu et al. [61] describe frameworks for lifelong program optimization, but without providing details on practical collection of data and optimization strategies across runs. Other frameworks [20,74] can collect profile information across multiple runs of users and continuously alter run-time decisions in Java virtual machines, while we focus on production static compilers and predictive modeling to correlate program features with program optimizations. In previous work [38,42] we presented an open source framework for statistical collective optimization that can leverage experience of multiple users with static compilers and collect run-time profile data transparently in an open public database for further machine learning processing. In [42], we also presented a new technique to characterize programs based on reaction to transformations, which can be an alternative portable approach to program characterization using static or dynamic program features.

We found many of the above approaches highly preliminary, limited to a few transformations and global flags, rarely with publicly released open source tools or experimental data to reproduce results. In contrast, the main goal of the Milepost project is to make machine learning based multi-objective optimization a realistic, automatic, reproducible and portable technology for general-purpose production compilers.

7 Conclusions and future work

The main contribution of this article is the first practical attempt to move empirical multi-objective iterative optimization and machine learning research techniques into production compilers, deliver open collaborative R&D infrastructure based on the popular GCC compiler and connect it to the cTuning.org optimization repository to help end-users optimize their applications and allow researchers to reproduce and improve experimental results. We show that Milepost GCC has a potential to automate the tuning of compiler heuristics for a wide range of architectures and multi-objective optimization such as improving execution time, code size, compilation time and other constraints while considerably simplifying overall compiler design and time to market.

We released all Milepost/cTuning infrastructure and experimental data as open source at cTuning.org [10,4,5] to be immediately useful to end users and researchers. We hope that Milepost GCC connected to cTuning.org's public collaborative tools and databases with common API will open many new practical opportunities for systematic and reproducible research in the area of empirical multi-objective optimization and machine learning. Some of Milepost's technology is now included in mainline GCC.

We continue to extend the Interactive Compilation Interface [8,49] to abstract high-level optimization processes from compiler internals and provide finer grain tuning for performance, power, compilation time and code size. We also expect to combine ICI with the POET scripting language [86] and pragmas to unify fine-grain program tuning. Future work will connect LLVM, ROSE, Path64 and other compilers to our framework. We are also integrating our framework with the collective optimization approach [42] to reduce or completely remove training stage overheads with limited benchmarks, architectures and datasets. Collective optimization also allows to define truly representative benchmarks based on classical clustering techniques.

Our framework now facilitates deeper analysis of interactions among optimizations and investigation of the influence of program inputs and run-time state on program optimizations in large applications. We also extend Milepost/cTuning technology to improve machine learning models and analyze the quality of program features to search for optimal sequences of optimization passes or polyhedral transformations [59, 78]. We started combining Milepost technology with machine-learning based auto-parallelization and predictive scheduling techniques [60,51,76]. We have also started investigating staged compilation techniques to balance between static and dynamic optimizations using machine learning in LLVM or Milepost GCC4CIL connected to Mono virtual machine. We plan to connect architecture simulators to our framework to enable software and hardware co-optimization. Finally, we will investigate adaptive and machine learning techniques for parallelization on heterogeneous multi-core architectures and power saving prediction for large data centers and supercomputers.

8 Acknowledgments

This research was generously supported by the EU FP6 035307 Project Milepost (MachIne Learning for Embedded PrOgramS opTimization) [11]. We would like to thank GRID5000 [44] community for providing computational resources to help validate results of this paper. We are grateful to Prof. William Jalby for providing financial support to Abdul Wahid Memon and Yuriy Kashnikov to work on this project. We would

like to thank Ari Freund, Björn Franke, Hugh Leather, our colleagues from the Institute of Computing Technology of Chinese Academy of Science and users from GCC, cTuning and HiPEAC communities for interesting discussions and feedback during this project. We would like to thank Cupertino Miranda and Joern Rennecke for their help to improve the Interactive Compilation Interface [8]. We would also like to thank Diego Novillo and GCC developers for practical discussions about the implementation of the ICI-compatible plugin framework in GCC. We are grateful to Google for their support to extend Milepost GCC during Google Summer of Code'09 program. We would also like to thank Yossi Gil for proof-reading this paper and anonymous reviewers for their insightful comments.

References

1. ACOVEA: Using Natural Selection to Investigate Software Complexities. http://www.coyotegulch.com/products/acovea.
2. CCC: Continuous Collective Compilation Framework for iterative multi-objective optimization. http://cTuning.org/ccc.
3. COD: Public collaborative repository and tools for program and architecture characterization and optimization. http://cTuning.org/cdatabase.
4. cTuning CC: cTuning Compiler Collection that can convert any traditional compiler into adaptive, machine learning enabled self-tuning infrastructure using Milepost GCC with ICI, CCC framework, cBench, COD public repository and cTuning.org web-services. http://cTuning.org/ctuning-cc.
5. cTuning.org: public collaborative optimization center with open source tools and repository to systematize, simplify and automate design and optimization of computing systems while enabling reproducibility of results.
6. ESTO: Expert System for Tuning Optimizations. http://www.haifa.ibm.com/projects/systems/cot/esto/index.html.
7. GCC: the GNU Compiler Collection. http://gcc.gnu.org.
8. ICI: Interactive Compilation Interface is a unified plugin system to convert black-box production compilers into interactive research toolsets for application and architecture characterization and optimization. http://cTuning.org/ici.
9. LLVM: the low level virtual machine compiler infrastructure. http://llvm.org.
10. MILEPOST GCC: public collaborative R&D website. http://cTuning.org/milepost-gcc.
11. MILEPOST project archive (MachIne Learning for Embedded PrOgramS opTimization). http://cTuning.org/project-milepost.
12. Open64: an open source optimizing compiler suite. http://www.open64.net.
13. OProfile: system-wide profiler for Linux systems, capable of profiling all running code at low overhead. http://oprofile.sourceforge.net.
14. PathScale EKOPath Compilers. http://www.pathscale.com.
15. Phoenix: software optimization and analysis framework for microsoft compiler technologies. https://connect.microsoft.com/Phoenix.
16. ROSE: an open source compiler infrastructure to build source-to-source program transformation and analysis tools. http://www.rosecompiler.org/.
17. F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.
18. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 30th Symposium on Microarchitecture (MICRO-30)*, 1997.
19. A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL'08)*, 2008.
20. M. Arnold, A. Welc, and V.T.Rajan. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05)*, 2005.

21. D. Barthou, S. Donadio, P. Carribault, A. Duchateau, and W. Jalby. Loop optimization using hierarchical compilation and kernel decomposition. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.
22. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.
23. E. V. Bonilla, C. K. I. Williams, F. V. Agakov, J. Cavazos, J. Thomson, and M. F. P. O'Boyle. Predictive search distributions. In *Proceedings of the 23rd International Conference on Machine learning*, pages 121–128, New York, NY, USA, 2006.
24. E. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, 1995.
25. B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1997.
26. J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March 2007.
27. J. Cavazos and J. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.
28. Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2010.
29. K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
30. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
31. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
32. S. Donadio, J. C. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzaran, D. A. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Proceedings of the International Workshop on Languages and Compilers for Parallel computing (LCPC)*, 2005.
33. C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimization across embedded programs and microarchitectures using machine learning. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, December 2009.
34. R. Duda, P. Hart, and D. Stork. *Pattern classification*. John Wiley and Sons, Inc, New-York, USA, 2001.
35. R. El-Yaniv, D. Pechyony, and E. Yom-Tov. Better multiclass classification via a margin-optimized single binary problem. *Pattern Recognition Letters*, 29(14):1954–1959, 2008.
36. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
37. G. Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.
38. G. Fursin. Collective Tuning Initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009.
39. G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.
40. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.

41. G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency: Practice and Experience*, 16(2-3):271–292, 2004.
42. G. Fursin and O. Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.
43. A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the Twenty-Second ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2007.
44. GRID5000. A nationwide infrastructure for large scale parallel and distributed computing research. `http://www.grid5000.fr`.
45. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.
46. K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.
47. K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.
48. K. Hoste and L.Eeckhout. Comparing benchmarks using key microarchitecture-independent characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 83–92, California,USA, October 2006.
49. Y. Huang, L. Peng, C. Wu, Y. Kashnikov, J. Renneke, and G. Fursin. Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In *2nd International Workshop on GCC Research Opportunities (GROW), colocated with HiPEAC'10 conference*, January 2010.
50. E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 195–206, 2006.
51. V. Jimenez, I. Gelado, L. Vilanova, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.
52. T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 237–246, 2000.
53. T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 237–246, 2000.
54. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.
55. P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
56. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.
57. J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the ACM SIGPLAN Conference on Programming Languaged Design and Implementation (PLDI'06)*, 2006.
58. X. Li, M. J. Garzaran, and D. A. Padua. Optimizing sorting with machine learning algorithms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.
59. S. Long and G. Fursin. A heuristic search algorithm based on unified transformation framework. In *Proceedings of the 7th International Workshop on High Performance Scientific and Engineering Computing (HPSEC-05)*, pages 137–144, 2005.
60. S. Long, G. Fursin, and B. Franke. A cost-aware parallel workload allocation approach based on machine learning techniques. In *Proceedings of the IFIP International Conference*

on Network and Parallel Computing (NPC 2007), number 4672 in LNCS, pages 506–515. Springer Verlag, September 2007.

61. J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism*, volume 6, 2004.
62. L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'09), colocated with HiPEAC'09 conference*, January 2009.
63. F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.
64. A. McGovern and E. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *Advances in Neural Information Processing Systems (NIPS)*. Morgan Kaufmann, 1998.
65. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.
66. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the Tenth International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA)*, LNCS 2443, pages 41–50, 2002.
67. J. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Advances in Neural Information Processing Systems (NIPS)*, pages 929–935. Morgan Kaufmann, 1997.
68. M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. Practical aggregation of semantical program properties for machine learning based optimization. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2010)*, October 2010.
69. A. Nisbet. Iterative feedback directed parallelisation using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation in conjunction with International Conference on Parallel Architectures and Compilation Technique (PACT)*, 1998.
70. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.
71. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.
72. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.
73. M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 77–90, June 2003.
74. M. W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, MIT, USA, 2006.
75. S. Touati, J. Worms, and S. Briais. The speedup test. In *INRIA Technical Report HAL-inria-00443839*, 2010.
76. G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2009.
77. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
78. K. Trifunovic, A. Cohen, D. Edelsohn, L. Feng, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjoedin, and R. Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *2nd International Workshop on GCC Research Opportunities (GROW)*, 2010.
79. J. Ullman. Principles of database and knowledge systems. *Computer Science Press*, 1, 1988.

80. M. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of International Conference on Parallel Processing*, 2000.
81. R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530, 2005.
82. S. wei Liao, T. han Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC)*, 2009.
83. J. Whaley and M. S. Lam. Cloning based context sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2004.
84. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.
85. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC)*, 2007.
86. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Proceedins of the Workshop on Performance Optimization of High-level Languages and Libraries (POHLL) co-located with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.