Crowdtuning : towards practical and reproducible auto-tuning via crowdsourcing and predictive analytics

Abdul Wahid Memon

▶ **To cite this version:**

Abdul Wahid Memon. Crowdtuning : towards practical and reproducible auto-tuning via crowdsourcing and predictive analytics. Intelligence artificielle [cs.AI]. Université Paris Saclay (COmUE), 2016. Français. NNT : 2016SACLV037 . tel-01395556

HAL Id: tel-01395556

https://theses.hal.science/tel-01395556

Submitted on 10 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2016SACLV037

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A
"UNIVERSITE DE VERSAILLES SAINT QUENTIN"

ECOLE DOCTORALE N° 580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat Informatique

Par

M. Abdul Wahid Memon

Crowdtuning: Towards Practical and Reproducible Auto-tuning via Crowdsourcing and Predictive Analytics

Thèse présentée et soutenue à Versailles, le 17 Juin 2016 :

Composition du Jury :

Monsieur BARTHOU Denis	Professeur, Université de Bordeaux	Président
Monsieur BARTHOU Denis	Professeur, Université de Bordeaux	Rapporteur
Monsieur BASTOUL Cédric	Professeur, Université de Strasbourg	Rapporteur
Monsieur CASTRO Pablo de Oliveira	Maître de conférences, Université de Versailles	Examinateur
HEYDEMANN Karine	Maître de conférences, Université Pierre et Marie Curie	Examinatrice
Monsieur JALBY William	Professeur, Université de Versailles	Directeur de thèse
Monsieur FURSIN Grigori	Chercheur, cTuning Foundation, France	Co-directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Crowdtuning: Auto-tuning Pragmatique et Reproductible via Crowdsourcing et Analyses Prédictives

Mots clés : réglage automatique de compilateur, gestion des connaissances, reproductibilité des ex- périmentations, l'optimisation du programme par crowdsourcing, apprentissage automatique, partage de code et des données

Résumé : Le réglage des heuristiques d'optimisation de compilateur pour de multiples cibles ou implémentations d'une même architecture est devenu complexe. De plus, ce problème est généralement traité de façon ad-hoc et consomme beaucoup de temps sans être nécessairement reproductible. Enfin, des erreurs de choix de paramétrage d'heuristiques sont fréquentes en raison du grand nombre de possibilités d'optimisation et des interactions complexes entre tous les composants matériels et logiciels. La prise en compte de multiples exigences, comme la performance, la consommation d'énergie, la taille de code, la fiabilité et le coût, peut aussi nécessiter la gestion de plusieurs solutions candidates. La compilation itérative avec profil d'exécution (profiling feedback), le réglage automatique (auto tuning) et l'apprentissage automatique ont montré un grand potentiel pour résoudre ces problèmes. Par exemple, nous les avons utilisés avec succès pour concevoir le premier compilateur qui utilise l'apprentissage pour l'optimisation automatique de code. Il s'agit du compilateur Milepost GCC, qui apprend automatiquement les meilleures optimisations pour plusieurs programmes, données et architectures en se basant sur les caractéristiques statiques et dynamiques du programme. Malheureusement, son utilisation en pratique, a été très limitée par le temps d'apprentissage très long et le manque de benchmarks et de données représentatives. De plus, les modèles d'apprentissage « boîte noire » ne pouvaient pas représenter de façon pertinente les corrélations entre les caractéristiques des programme ou architectures et les meilleures optimisations.

Dans cette thèse, nous présentons une nouvelle méthodologie et un nouvel écosystème d'outils (framework) sous la nomination Collective Mind (cM). L'objectif est de permettre à la communauté de partager les différents benchmarks, données d'entrée, compilateurs, outils et autres objets tout en formalisant et facilitant la contribution participative aux boucles d'apprentissage. Une contrainte est la reproductibilité des expérimentations pour l'ensemble des utilisateurs et plateformes. Notre cadre de travail open-source et notre dépôt (repository) public permettent de rendre le réglage automatique et l'apprentissage d'optimisations praticable. De plus, cM permet à la communauté de valider les résultats, les comportements inattendus et les modèles conduisant à de mauvaises prédictions. cM permet aussi de fournir des informations utiles pour l'amélioration et la personnalisation des modules de réglage automatique et d'apprentissage ainsi que pour l'amélioration des modèles de prévision et l'identification des éléments manquants.

Notre analyse et évaluation du cadre de travail proposé montre qu'il peut effectivement exposer, isoler et identifier de façon collaborative les principales caractéristiques qui contribuent à la précision de la prédiction du modèle. En même temps, la formalisation du réglage automatique et de l'apprentissage nous permettent d'appliquer en permanence des techniques standards de réduction de complexité. Ceci permet de se contenter d'un ensemble minimal d'optimisations pertinentes ainsi que de benchmarks et de données d'entrée réellement représentatifs.

Nous avons publié la plupart des résultats expérimentaux, des benchmarks et des données d'entrée à l'adresse http://c-mind.org tout en validant nos techniques dans le projet EU FP6 Milepost et durant un stage de thèse HiPEAC avec STMicroelectronics.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Crowdtuning: Towards Practical and Reproducible Auto-tuning via Crowdsourcing and Predictive Analytics

Keywords : compiler auto-tuning, knowledge management, reproducible expeirments, crowdsourcing program optimization, machine learning, code and data sharing

Tuning general compiler optimization heuristics or optimizing software for rapidly evolving hardware has become intolerably complex, ad-hoc, time consuming and error prone due to enormous number of available design and optimization choices, complex interactions between all software and hardware components, and multiple strict requirements placed on performance, power consumption, size, reliability and cost. Iterative feedback-directed compilation, auto-tuning and machine learning have been showing a high potential to solve above problems. For example, we successfully used them to enable the world's first machine learning based self-tuning compiler, Milepost GCC, which automatically learns the best optimizations across multiple programs, data sets and architectures based on static and dynamic program features. Unfortunately, its practical use was very limited by very long training times and lack of representative benchmarks and data sets. Furthermore, « black box » machine learning models alone could not get full insight into correlations between features and best optimizations.

In this thesis, we present the first to our knowledge methodology and framework, called Collective Mind (cM), to let the community share various bench marks, datasets, compilers, tools and other artifacts while formalizing and crowdsourcing optimization and learning in reproducible way across many users (platforms). Our open-source framework and public optimization repository helps make auto-tuning and machine learning practical. Furthermore, cM let the community validate optimization results, share unexpected run-time behavior or model mispredictions, provide useful feedback for improvement, customize common auto-tuning and learning modules, improve predictive models and find missing features. Our analysis and evaluation of the proposed framework demonstrates that it can effectively expose, isolate and collaboratively identify the key features that contribute to the model prediction accuracy. At the same time, formalization of auto-tuning and machine learning allows us to continuously apply standard complexity reduction techniques to leave a minimal set of influential optimizations and relevant features as well as truly representative benchmarks and data sets.

We released most of the experimental results, benchmarks and data sets at http://c-mind.org while validating our techniques in the EU FP6 MILEPOST project and during HiPEAC internship at STMicroelectronics.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Dedicated to my family and friends.

Publications

[1] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael F. P. O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[2] Grigori Fursin, Abdul Wahid Memon, Christophe Guillon, and Anton Lokhmotov. Collective mind, part II: Towards performance-and cost-aware software engineering as a natural science. *arXiv preprint arXiv:1506.06256, 18th International Workshop on Compilers for Parallel Computing (CPC'15) London, UK*, January 2015.

[3] Abdul Wahid Memon and Grigori Fursin. Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing. In *PARCO mini-symposium on "Application Autotuning for HPC (Architectures)"*, Munich, Allemagne, September 2013.

Shared Artifacts

All the benchmarks, datasets and tools pertaining to Collective Mind experimentations have been shared for validation by community at c-mind.org/repo.

Acknowledgements

First and foremost, I am grateful for the scientific and technical support of Dr. Grigori Fursin to successfully complete this thesis. He has been very kind and cooperative throughout my research work. His vast expertise in the domain of optimizing compilers, machine learning, auto-tuning software and reproducibility of experiments has been a great source of inspiration for me right from the beginning. His continuous support and coordination always kept me on the right path throughout my PhD. I wish him all the very best for his career.

I am thankful to Prof. William Jalby, my thesis director, for his scientific and administrative support. His polite attitude and cooperation has been a tremendous source of encouragement for me.

I extend my gratitude to Pablo de Oliveira, Christophe Guillon, Umair Ali Khan, Christian Bertin, Yvan Roux, Yuriy Kashnikov, and Abdelhafid Mazouz for their technical and moral support throughout my research work and thesis write-up. They have been great friends and mentors for me throughout my PhD.

I am also very thankful to HEC[1], Pakistan for awarding me Masters leading to PhD scholarship and HiPEAC[2] for providing me the research grant to conduct research at STMicroelectronics, France from August to December 2013.

Lastly, I am in debt to the continuous moral support of my parents, family and friends during my PhD. I wish I could tell them how much I love them.

[1] Higher Education Commission

[2] European Network of Excellence on High Performance and Embedded Architecture and Compilation

Contents

List of Figures

List of Tables

1

Introduction

1.1 Introduction

Designers of new embedded architectures attempt to bring higher performance and lower power across a wide range of programs while keeping time to market as short as possible. These embedded architectures vary in performance, size, power consumption, reliability, price and other characteristics depending on numerous available hardware features such as processor architecture, number of cores, availability of specialized hardware accelerators, working frequency, memory hierarchy and available storage. However, delivering such resources for high performance computing or ultra low-power for embedded systems is becoming intolerably complex, costly and error prone due to limitations of available technology, huge number of available designs and optimization choices, complex interactions between all software and hardware components, and growing number of incompatible tools and techniques with ad-hoc, intuition based heuristics. As a result, understanding and modeling of the overall relationship between end-user algorithms, applications, compiler optimizations, hardware designs, data sets and run-time behavior, essential for providing better solutions and computational resources, have become infeasible as confirmed by numerous recent long-term international research visions about future computer systems [5, 71, 38, 68, 67, 118]. On the other hand, engineers often have to develop software that may end up running across different, heterogeneous and possibly virtualized hardware in multiple embedded platforms, desktops, HPC servers, data centers and cloud services. Such a rising complexity of computer systems and limited development time usually force software engineers to rely almost exclusively on existing compilers, operating systems and run-time libraries in a hope to deliver highly optimized, computational-efficient, scalable, reliable and power-efficient executable codes across all available hardware. As a result, peak

performance of the new systems is often achieved only for a few previously optimized and not necessarily representative benchmarks such as SPEC for desktops and servers or LINPACK for TOP500 supercomputer ranking, while leaving most of the systems severely underperforming and wasting expensive resources and power.

Automatic offline and online performance tuning of compilers to achieve satisfactory portable performance is generally performed using empirical iterative compilation for statically compiled programs, applying automatic compiler tuning based on feedback-directed compilation. In this technique, 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.

Bodin et al. [11] studied the applicability of iterative optimization for selecting the best optimization for a program. They showed that by using profile feedback in the form of execution time, a restricted non-linear search space could be effectively searched to outperform the existing approaches.

Nisbet et al. [109] showed that genetic algorithm can be applied to iterative optimization problems. In their proposed approach, referred to as Genetic Algorithm Parallelisation System (GAPS), they evaluate the application and performance benefit of genetic algorithm optimization techniques to the compilation of loop-based programs for parallel architectures. They showed that GAPS can deliver significant performance improvement in parallel execution time.

Cooper et al. [30] targeted to reduce the size of the compiled code by applying the genetic algorithm to find optimization sequence that generates small object codes. Their evaluations on several benchmarks and comparisons with no-optimization or fixed optimization techniques showed significant improvements.

Kisuki et al. [88] investigated the efficacy of iterative compilation on a combined selection of tile sizes and unroll factors. They further evaluate the iterative strategies based on genetic algorithms, random sampling and simulated annealing to select optimal tile sizes and unroll factors simultaneously. They compare their optimization strategies with several existing techniques and showed that their techniques outperform each of them on a variety of architectures.

Cooper et al. [29] focused on compile-execute-analyze feedback loop and proposed a technique to reduce the execution time of iterative compilation. Their proposed technique captures salient information about a program in a single run and uses this information to predict its performance for different optimization sequences. Their experiments showed that their proposed technique can significantly reduce the adaptive compilation time.

Kulkarni et al [89] used genetic algorithm to find effective sequences of optimization phases and providing the user with dynamic and static performance information that can be used during an interactive compilation session to gauge the progress of improving

the code. A peculiar shortcoming of all the techniques using genetic algorithm for iterative compilation is that genetic algorithms can be unstable and their fixed-length representation precludes their use in many problems.

Cooper et al. [31] explored different orders of optimization sequence to discover a program-specific compilation sequence that minimizes an explicit, external objective function. They showed that depending on the objective function selected, their proposed technique can produce significant reduction in code size and execution time. However, their technique requires large number of passes before it can discover a solution.

Triantafyllis et al. [133] came up with improvements over the traditional predictive heuristic techniques by proposing a technique which uses the compiler writer's knowledge encoded in the heuristics to select a small number of promising optimization alternatives for a given code segment. However, this technique is limited to the best sequences categorized into a small tree of compiler options [28].

Fursin et al. [51] proposed a technique for guiding optimizations for numerical applications based on iterative feedback-directed program restructuring. Using the profiling techniques to find the best possible program variant, they showed significant performance improvement as compared to the native static and platform-specific feedback directed compilers.

Pan et al. [113] also focused on reducing the number of evaluations by proposing a technique referred to as Combined Elimination (CE) which selectively turns off optimizations until the best optimization is found for a new program. Their results showed that the performance achieved by CE is close to the upper bound obtained by an exhaustive search algorithm.

Code optimization for embedded devices must take into consideration the important factors of power consumption, performance, memory space, etc. In order to deal with these conflicting objectives, Heydemann et al. [69] formulated the iterative optimization problem as a constraint optimization methodology to find a Pareto-optimal search space among multiple objectives. They showed that they could find a deeper trade-off between these objectives at the expense of minimum possible performance degradation. Hoste et al. [74] further explored the same problem by automatically finding the Pareto-optimal search space of the multiple objectives. They showed that their technique can produce better optimization levels than GCC's manually driven optimization levels and those obtained through random sampling.

Franke et al. [46] further worked on finding best optimization sequences in a large search space. Their technique comprised localization of specific areas of search space to find the best candidate solution, while reducing the search time. Their experiments demonstrated that their technique outperformed the existing techniques by significantly reducing the execution time.

A common assumption prevelant in iterative compilation was that the best configuration found for any arbitrary data set will work well with other data sets that a

program uses. Fursin et al. [53] evaluated this assumption MiBench benchmark for 20 datasets per benchmark. They found that although the variability increases for many optimizations, it is found to be small for the best optimization configuration and a compromised optimization configuration across data sets can be found.

Chen et al. [20] evaluated iterative compilation for much larger data sets by creating a huge, publicly available data set for 32 programs. They showed that despite of the diversity of the data set, iterative compilation can be used to find a robust strategy for all programs. They further showed that there exists at least one combination of compiler optimizations that achieves 86% or more of the best possible speedup across all data sets.

Fursil et al. [54] focused on speeding up the evaluations of a large number of optimizations in iterative compilation using static multi-versioning of the most time-consuming code sections, and a low-overhead run-time phase detection scheme. This technique can speed up iterative search by several orders of magnitude and can be beneficial during the training data generation stage of our models.

This approach requires little or no knowledge of the platform and can adapt programs to any given architecture. This approach is currently used in library generators and adaptive tools [137, 97, 124, 117, 1, 44]. 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 excessive compilation and execution time needed in order to optimize each program. This prevents a wider adoption of iterative compilation for general purpose compilers.

This problem can be more effectively addressed using 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. Machine learning has been actively promoted as a possible solution to cope with ever rising complexity of computer systems including dramatically increasing number of available program optimizations such as compiler flags for more than a decade.

Calder et al. [CGJ1997] used neural networks and decision trees for static branch prediction at compile time. They trained the neural network with the input program features associated with each branch in the training set such as control flow and op-code information. This approach outperformed static heuristics [6, 14] by a considerable margin.

In another work [129], an approach referred to as Meta optimization was introduced to fine-tune compiler heuristics using genetic algorithm. As the experimental use-cases, three optimizations were selected, namely hyper-block formation, register allocation and data pre-fetching. They achieved significant improvements for hyper-block formation and data pre-fetching. However, the heuristic for register allocation did not turn out to be more effective as they were able to only achieve an average 2% improvement over the

manually tune heuristic.

Monsifrot at el. [106] attempted to generate the optimization heuristic for a simple optimization called loop unrolling. Though simple, it is difficult to devise a prediction rule for this optimization. They constructed a classifier based on a decision tree to predict loop unrolling in a program. This was a preliminary approach to show that decision trees can be successfully used to learn the target-specific heuristics for loop unrolling with a reasonable accuracy.

In another work [16], Cavazos et al. showed that supervised learning can be effectively used for predicting instruction scheduling. Though instruction scheduling drastically reduces program's execution time, it does have adverse effects on certain blocks in the program. Hence, it is important to predict which part of the program may be used for instruction scheduling. Using training features associated with the instruction scheduling, Cavazos et al. built a supervised learning based classifier to predict with acceptable accuracy whether scheduling a certain block in the program will increase its overall execution performance.

Stephensonet al. [128] addressed the loop unrolling optimization in the context of predicting unrolling factor (the degree of loop unrolling). Using the training features associated with loop unrolling, they built a supervised learning based classifier to predict loop unrolling factor with reasonable precision.

Agakov et al. [2] worked on speeding up the iterative optimization technique which is affected by higher number of evaluations in large search spaces. This technique builds a model of the program features and search space by offline training and directs the optimization algorithm to focus only on those areas of search space which have potentially larger contribution, resulting in a substantial speed up of iterative compilation. In a similar work [77], Ipak et al. used machine learning to build predictive design-space models describing the relationship among design parameters. Their generated models are capable to produce reasonably accurate performance estimates for different points in the space and enable efficient discovery of tradeoffs among parameters in different regions.

In a more recent work [17], Cavazos et al. used machine learning to build a predictive model to correlate the performance counters of a program with the good optimization options. Using their predictive models, they achieved a 10% average speedup over the highest optimization setting the PathScale compiler on SPEC benchmarks.

Li et al. [94] addressed the problem of generating optimized sorting algorithms using genetic algorithms and machine learning based classifiers. Their proposed approach is able to select the best sorting algorithm as a function of the characteristics of the input data. They showed that their technique performs significantly better than the many conventional sorting implementations.

As an incremental approach, Dubach et al. [39] addressed the problem of finding right optimizations for a program using machine learning by finding a mapping from

the program features to a probability distribution over good optimization phases. An important aspect of their approach was that it was able to adapt to architectural changes without re-training the algorithm. When a new program needs to be compiled, the best predicted executable is immediately generated, without iterative compilation and without the need for a training phase specific for the target architecture.

Tournavitis et al. [132] proposed a profile-driven compiler-based auto parallelization technique using machine learning. They used profiling data to extract actual control and data dependences and integrated profile-driven parallelism detection and machine-learning based mapping in a single framework. They also discussed the limitations of existing auto-parallelization techniques and demonstrated that their proposed technique can significantly outperform the existing techniques. However, they did not address the important problem of scalability.

Qilin et al. [95] addressed the problem of mapping computations to processing elements automatically in a heterogeneous multiprocessors system. In this context, they proposed an adaptive mapping technique based on offline training, referred to as Qilin, and demonstrated its efficacy for mapping computations to processing elements on a CPU+GPU machine. Their proposed technique can dynamically determine an effective partitioning of work across heterogeneous resources, but targets only data-parallel operations [17].

Since machine learning based compiler optimization is generally exposed to the problem of dealing with theoretically infinite number of program features, identification and selection of the best features from an infinitely large search space offers a dedicated challenge. Leather et al. [93] addressed this problem by developing an optimized search space described by a grammar and searched with genetic programming and predictive modeling to find the best static features. However, the static features discovered are those that can be summarized into a fixed-length feature vector. Also, their technique only outperforms static source code features (such as, SRC) by only a couple of percent on average.

Park et al. [114] used a graph-based characterization technique to predict the best optimizations for a program. Their proposed techniques build a program's control flow graph which is then applied to support vector machine to prediction models with the shortest path graph kernel. The authors showed that this method of characterizing programs is competitive with previous characterization techniques.

Stock et al. [130] addressed the problem of automatic vectorization of compute-intensive programs using machine learning. They predict the performance of SIMD code using different machine learning models trained offline on the features extracted from the generated assembly codes. The predictions were used at compile-time to discriminate between numerous possible vectorized variants generated from the input code. They evaluated their machine learning models on different computations and showed good improvements over Intel ICC's auto-vectorized code.

Moore et al. [107] propose a technique to dynamically determine the application affinity (thread-to-core mapping) in multi-core machines using machine learning. Referred to as AutoFinity, their proposed technique first gathers the training data for a range of affinities and program behavior from the training programs. It then utilizes machine learning methods to construct an action table which provides policies for thread-to-core mappings. The generated policy is used at runtime to select a program's affinity. The policy can handle programs that were not part of the training and/or thread counts that have not been considered by training.

In a recent work, Park et al. [115] presented a machine learning technique to select the best polyhedral optimizations for compute-intensive programs using limited iterative search. They first used static cost models to reduce the set of candidate optimizations and then predict the performance of each optimization with machine learning models. They achieved significant performance improvement for various benchmarks on multi-core platforms.

Kejariwal et al. [15] focused on inlining heuristic constraints for program optimization. They built machine learning models to learn the correlation between inlining vectors and program completion time. Combined with other global optimizations, the machine learning based model selects an inlining vector that minimizes the completion time of a program. Their evaluations on GNU GCC compiler and optimized 22 combinations (program, input) from SPEC CINT2006 showed promising results.

Existing studies usually focus on a few positive outcomes (predictions) to improve execution time, power consumption or other characteristics using some off-the-shelf black-box classification and predictive modeling techniques such as SVM, neural networks or KNN [9, 70, 92], several optimizations and a few benchmarks combined with several ad-hoc program or architecture features. Though undoubtedly interesting, such limited studies can only demonstrate some potential of using machine learning for predictions but do not include deep and systematic analysis of the selection of a learning algorithm and related features for large and realistic training sets which are the major research challenges in the field of machine learning for decades, and far from being solved [9].

Having drawn inspiration from the potential benefits of machine learning, we introduced a novel compiler technology, called Milepost GCC, 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 embedded systems. Rather than developing a specialized compiler by hand for each configuration, our machine learning based platform aims to produce optimizing compilers automatically.

Our rigorous experimentation and analysis showed that machine learning based program auto-tuning does outperform iterative compilation, but at the same time it

can not be singled out as the best approach to guarantee optimal solution for all cases. In order to find the strong feature-optimization correlation and to capture the precise run-time dynamics for predicting the best transformations, machine learning requires a huge training data set which should cover adequate number of possible scenarios and observations. Obtaining such a huge data set for training with all unforeseen examples is extremely difficult and hence limits the prediction accuracy. Therefore, we established that machine learning alone can not serve this purpose as desired.

1.2 Motivation

The potential of machine learning for auto-tuning was now well-tested by us and we already had a machine learning based compiler at hand which is the first of its kind. However, along with the problem of insufficient training data set, we also realized that the "black box" nature of machine learning algorithms together with the lack of common experimental methodology and culture of sharing large, diverse and reproducible experimental sets makes it too tedious or sometimes even impossible to validate results of existing approaches and use them to improve compilers, applications and architectures. All these issues started to raise many concerns about practicality and scalability of our machine learning based approach for compilation and architecture in realistic production scenarios.

Considering all the aforementioned problems, we established that machine learning can not be used effectively as a standalone prediction system for compiler auto-tuning without a collaborative approach of collecting the relevant experimental results from the community and disseminating the artifacts in return. This led us to introduce the innovative idea of using a combination of offline and online learning for this purpose. We decided to collect the optimization results and codes from the community and replace the related optimizations from our repository with the better ones after validation. This further allows us to get new observations, thus leading to better feature-optimization correlation and model adjustment to improve prediction accuracy through online learning.

The research carried our in this thesis is motivated by the development of a novel, scalable and extensible optimization methodology and public framework that attempts to address all above-mentioned challenges in a cooperative and coherent way while gradually unifying and validating existing ad-hoc techniques and tools. Instead of relying on a few positive and often non-reproducible experimental outcomes, we propose to formalize and expose the whole optimization scenario including multiple optimization choices and characteristics to the community or a workgroup in a modular and portable way as a buildbot. By this way, we can easily distribute various optimization scenarios among many participants and continuously explore available optimization choices for all shared code and data set samples from the community in realistic environments

while focusing on unexpected behavior and mispredictions. All behavior anomalies can be continuously collected and exposed in a centralized repository to find most optimal predictive models and correlating algorithm, program, architecture, data sets and other features for a given scenario either automatically or through crowdsourcing as it is currently successfully used in other sciences including biology and artificial intelligence.

1.3 Cooperative research and experimentation

Many of the challenges and pitfalls pertaining to compiler performance tuning are caused by the lack of a common experimental methodology, lack of interdisciplinary background, and lack of unified mechanisms for knowledge building and exchange apart from numerous similar publications, where reproducibility and statistical meaningfulness of results as well as sharing of data and tools is often not even considered in contrast with other sciences including physics, biology and artificial intelligence. In fact, it is often impossible due to a lack of common and unified repositories, tools and data sets. At the same time, there is a vicious circle, since initiatives to develop common tools and repositories to unify, systematize, share knowledge (data sets, tools, benchmarks, statistics, models) and make it widely available to the research and teaching community are practically not funded or rewarded academically where a number of publications often matter more than the reproducibility and statistical quality of the research results. As a consequence, students, scientists and engineers are forced to resort to some intuitive, non-systematic, non-rigorous and error-prone techniques combined with unnecessary repetition of multiple experiments using ad-hoc tools, benchmarks and data sets. Furthermore, we witness slowed down innovation, dramatic increase in development costs and time-to-market for the new embedded and HPC systems, enormous waste of expensive computing resources and energy, and diminishing attractiveness of computer engineering often seen as “hacking” rather than systematic science.

Our basic idea is to bring interdisciplinary community together to collaboratively explore various research and experimental scenarios while explaining unexpected behavior and mispredictions. However, unlike some other sciences where similar approach has already been successfully used for years, it is not yet widely used in design and optimization of computer systems due to at least two major problems: variability in behavior of computer systems such as execution time and very complex and continuously evolving experimental setups with multiple hard-wired ad-hoc and ever changing tools and architectures combined with some tuning and analysis scripts while often sharing results in non unified CSV, TXT and XLS files with some limited meta-description or at most in MySQL and similar databases, as conceptually shown in Figure 5.2a. Usually, by the end of tedious development and experimentation, new versions of

compilers, libraries, operating systems and architectures are already available making results potentially outdated while problems possibly solved or considerably evolved.

1.4 Collective Mind: cooperative experimentation

We propose to use our recent Collective Mind framework and Hadoop-based repository of knowledge (cM for short) [59, 100] to extract and share the open-source software pieces together with various possible inputs and metadata at c-mind.org/repo. This metadata is gradually extended by the community via popular, human readable and easily extensible JSON format [83], currently describing how to build and run shared pieces together with all dependencies on the specific hardware and software including compilers, operating systems and run-time libraries. All these shared software pieces are then continuously and randomly optimized and executed with different data sets using distributed cM buildbot for Linux and Windows-based devices [100] or cM node for Android devices [101] across shared computational resources provided by volunteers. Such resources range from smartphones, tablets, laptops and desktops to data centers, supercomputers and cloud services gradually covering all existing hardware configurations and environments. Furthermore, the community can use lightweight cM wrappers around identified software pieces within a real and possibly proprietary applications to continuously monitor their behavior and interactions within the software project. Similar to nature and biological species, such approach treats all exposed and shared software pieces as *computational species* while continuously tracking and learning their behavior versus different optimizations across numerous hardware configurations, realistic software environments and run-time conditions. cM infrastructure then continuously records only the winning solutions (optimizations for a given data set and hardware) that minimize all or only monitored costs (execution time, power consumption, code size, failures, memory and storage footprint, and optimization time) of a given software piece on a Pareto frontier [90] in our public cM repository.

Software engineers can now assemble their projects from the cM plugins with continuously optimized computational species. Such software projects can continuously and collaboratively achieve better performance while reducing all costs across all hardware thus making software engineering performance- and cost-aware. Furthermore, software developers are now able to practically help compiler writers and hardware designers improve their technology as conceptually shown in Figure 1.1b. Indeed, our approach helps create the first to our knowledge public, realistic, large, diverse, distributed, evolving and continuously optimized benchmark with related optimization knowledge while gradually covering all possible software and hardware.

At the same time, we can also apply an extensible, top down methodology originating from physics when learning behavior of complex systems. The compiler community first learns and optimizes coarse grain behavior of large shared software pieces in-

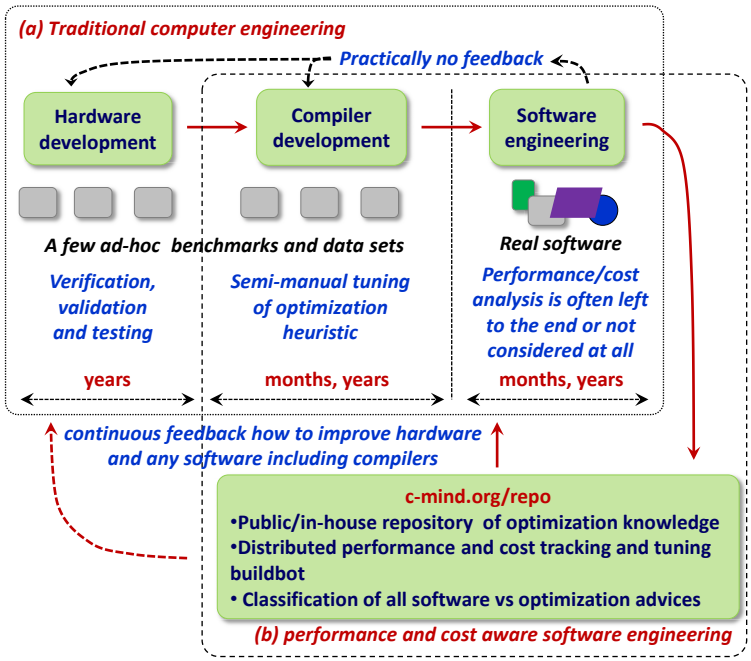


Figure 1.1 – *(a) Traditional computer engineering versus (b) Our new collaborative performance and cost-aware software/hardware co-design as a web service.*

cluding whole applications, library functions, kernels and most time consuming loops versus global compiler optimization flags or other coarse-grain optimizations. After enough knowledge is collected, the community can gradually move to finer grain levels including just a few source lines or binary instructions versus all internal and individual compiler optimization decisions via our Interactive Compilation Interface. This plugin-based interface is already available in mainline GCC [55], and we plan to add it to LLVM in the future [59].

More importantly, our approach helps considerably improve existing methodology on optimization and run-time adaptation prediction using machine learning. Current methodology (used in most of the papers referenced in Section 1.1 and including ours) usually focuses on *showing that it is possible to predict* one or several optimizations to improve execution time, power consumption or some other characteristics using some off-the-shelf machine learning techniques such as SVM, (deep) neural networks or KNN [9, 70, 92] combined with a few ad-hoc program or architecture features. In contrast, our growing, large and diverse benchmark allows the community for the first time to apply methodology from sciences such as biology, medicine and AI based on big data predictive analytics [68]. For this purpose, cM infrastructure continuously classifies all winning species in terms of distinct optimizations and exposes them to the community in a unified and reproducible way through the public repository. This, in turn, allows our colleagues with interdisciplinary background to help the software engineering community *find the best predictive models* for these optimization classes together *with relevant features from software species, hardware, data set and environment*

state either manually or automatically. Such features (including extraction tool) and predictive models are continuously added to the species using cM wrappers and their meta-data thus practically enabling self-tuning software automatically adaptable to any hardware and environment.

Importantly, cM continues tracking unexpected behavior (abnormal variation of characteristics of species such as execution time, or mispredictions from current classification) in a reproducible way in order to allow the community improve predictive models and find missing features that can explain such behavior. Also, in contrast with using more and more complex and computationally intensive machine learning techniques to predict optimizations such as deep neural networks [9, 70, 92], we decided to provide a new manual option useful for compiler and hardware designers. This option allows the community to combine existing predictive techniques as a cheap way to quickly analyze large amount of data, with manually crafted human-readable, simple, compact and fast rules-based models (decision trees) that can explain and predict optimizations for a given computational species. Thus, we are collaboratively building a giant optimization advice web service that links together all the shared software species, optimizations and hardware configurations while resembling Wikipedia, IBM Watson advice engine [45], Google knowledge graph [102] and a brain.

We understand that the success of our approach will depend on the active involvement from the community. Therefore, we tried to make our approach as simple and transparent to use as possible. For example, our light-weight cM version for Android mobile systems [101] is a "one-button approach" allowing anyone to share their computational resources and tune shared computational species. At the same time, extraction of software pieces from large applications is still semi-manual and may incur some costs. Therefore we are gradually working on automating this process using plugin-based capabilities in GCC and LLVM. Furthermore, together with participating companies and volunteers, we already extracted, described and partially [1] shared 285 computational species together with around 500 input samples [2] from major benchmarks and software projects. We then validated our approach in STMicroelectronics during 3 months to help our colleagues tune their production GCC compiler and improve real customer software. During that time, we continuously optimized execution time, code size, compilation time and power consumption of all shared computational species using at least 5000 random combinations of compiler optimization flags on spare private cloud servers and mobile phones. We also managed to derive 79 distinct optimization classes covering all shared species (small real applications or hotspot kernels extracted from large applications with their run-time data set either manually as we did in [54], or using Codelet Finder from CAPS Enterprise as we did in the MILEPOST project [55],

[1] We cannot share extracted pieces from proprietary software but we still use them internally.

[2] We currently have more than 15000 input samples collected in our past projects for our shared computational species [53, 103, 20]. However since they require more than 17GB of storage, at the moment we decided to share only representative ones, i.e. which require distinct compiler optimization.

or using semi-manual extraction of OpenCL/CUDA kernels combined with OpenME plugin interface to extract run-time state [59]) that we correlated with program semantic and dynamic features using SVM [to be presented in Section 2.5.3] and other predictive analytics techniques. With the help of domain specialists (compiler engineers), we then analyzed predictive models for end-user software, found meaningless correlations, manually isolated problems [3], prepared and shared counter-example code sample, found missing program and input features to fix wrong classifications, and developed adaptive, self-tuning and statically compiled code. Finally, we managed to substitute ad-hoc benchmark used at the architecture verification department of our industrial partners with the minimal and realistic one based on derived optimization classes that helped to dramatically reduce development and testing time.

These positive outcomes demonstrate how our approach can help eventually involve the software engineering community into development and improvement of compilers and hardware. We also show how continuously growing collective knowledge repository accessible via unified web service can become an integral part of the practical software and hardware co-design of self-tuning computer systems while decreasing all development costs and time-to-market for new products. More importantly, the side effect of our approach to share code and data in a reproducible way help support recent international initiatives on reproducible research and sustainable software engineering [104].

1.5 Real-life motivating example

A couple of decades ago, my scientific advisor, Dr. Grigori Fursin, started developing and analyzing various artificial neural networks as part of a possible non-traditional and brain-inspired computer [48, 49, 50]. Such networks can mimic brain functions and are often used for machine learning and data mining [9]. For example, Figure 1.2 shows one of the oldest and well-known one-layer, fully interconnected, recurrent (with feedback connections) Hopfield neural network [72]. It is a popular choice for function modeling, pattern recognition and image filtering tasks including noise reduction. Implemented as a software, this neural network has a fairly simple and regular code where each neuron receives a weighted sum of all inputs of an image as well as outputs of all other neurons. This sum is then processed using some neuron activation function including sigmoid or linear ones to calculate the output value. The small and simple C kernel presented in Figure 1.2 is one of many possible implementations of a threshold filter we used as a part of a linear activation function, i.e. switching neuron output from 0 to 1 when its input meets a given threshold. Very simplistically, the quality of a neural network is usually determined by its processing speed as well as capacity (maximum amount of patterns or information that can be stored in such networks) and recognition

[3] In spite of many papers that present some simple automatic optimization predictions, our practical and industrial experience with large data sets shows that it is currently not possible to fully automate this process. Therefore, manual analysis is still often required similar to other natural sciences.

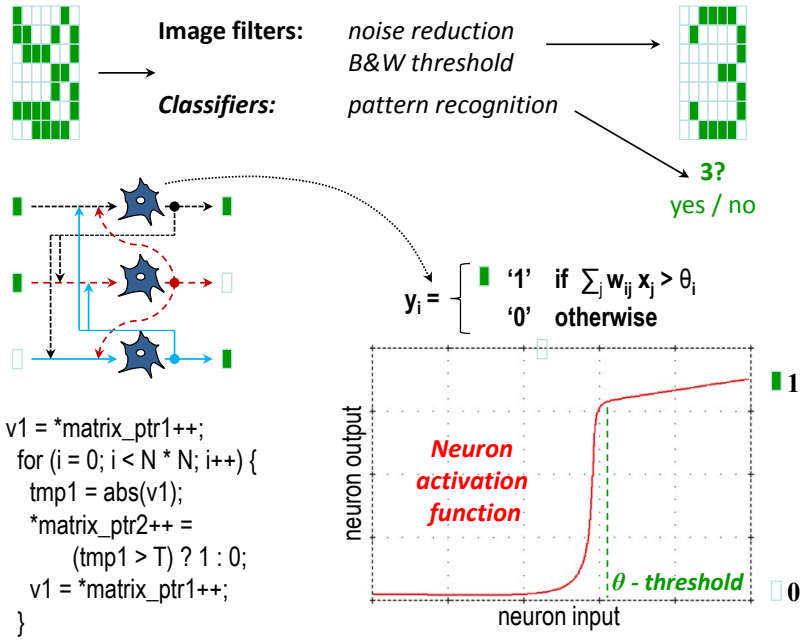


Figure 1.2 – *Conceptual example of pattern recognition, image filtering and character noise reduction using Hopfield fully interconnected and recurrent neural network. Simple C kernel is a part of a neuron activation function processing thresholds for all neurons.*

accuracy (correct predictions versus failures). It heavily depends on the total number of neurons, connections and layers [76], and is primarily limited by the speed and resources of the available hardware including specialized accelerators. Hence, neural network software/hardware co-design process always involves careful balancing of performance versus all associated costs including storage size, memory footprint, energy consumption, development time and hardware price depending on usage scenarios and required time to market. Indeed, Grigori's research on improving neural networks requires many iterative runs of a slightly evolving modeling software with varying parameters to maximize prediction accuracy. In this case, the main concern is about minimizing compilation and execution time of each execution across available hardware. However, when the best found network is found and deployed in a large data center or cloud service (for example, for big data analysis), end users would like to minimize all additional costs including energy and storage consumption across all provided computer systems. Finally, when deploying neural networks in small, autonomic and possibly mass-produced devices such as surveillance cameras and mobile phones or future robots and Internet of Things objects, more strict requirements are placed on software and hardware size, memory footprint, real time processing, and the cost of the whole system.

Twenty years ago, the software engineering of neural networks was relatively straightforward. Users did not have a choice but to simply select the latest hardware with the accompanying and highly tuned compiler to achieve nearly peak performance for their software including for the code shown in Figure 1.2. Therefore, in order to innovate and

process more neurons and their configurations, users usually had to wait for more than a year until arrival of a new hardware. This hardware would likely double performance of our software and provide more memory and permanent storage but often at a cost of higher power consumption and thus dramatically rising electricity bill.

In contrast, we now have an impressive choice of hardware of all flavors which our software can be executed on. Each year, there are numerous variations of processors appearing on the market with different features (properties) including frequency, number of cores, cache size, ISA extensions, specialized hardware accelerators (such as GPU and even revived semiconductor neural networks), power consumption and price. Furthermore, we can now have easy access to large-scale parallel resources from home via popular virtualized cloud services from Amazon, Google, Microsoft and others. Therefore, the number of experiments we can now run is mainly limited by the price we can afford to pay for computing services. At the same time, we also enjoy continuous community-driven improvements of operating systems together with numerous free or proprietary libraries and software development tools including popular optimizing compilers such as GCC and LLVM. One may expect that with so many advances in the computer technology, practically any recent compiler would generate the fastest and most energy efficient code for such an old, simple, small and frequently used software piece shown in Figure 1.2 across existing hardware. Nevertheless, since users pay for experiments, together with Dr. Grigori Fursin, we eventually decided to validate their performance/cost efficiency.

For the sake of accountability and reproducibility, we started gradually collecting at c-mind.org/nnet-tuning-motivation various information about several computer systems we used including their price, cost, available operating systems, compilers and optimizations. Figure 1.3a shows a tiny subset of this multidimensional space of design and optimization choices. At the same time, whenever running real experiments, we also started recording their execution time and all associated costs including compilation time, code size, energy usage, software/hardware price and utility bill. It is worth mentioning that by performance tuning, we mean reducing execution time. However, on modern out-of-order processors with complex memory hierarchy, the dependency between performance and total execution time may be non-linear. Thus, depending on user requirements, these characteristics have to be tuned separately.

We further decided to perform a simple and well-known optimization compiler flag autotuning [1, 55] with at least 100 iterations to see whether there is still room for improvement over the fastest default compiler optimization level (-O3). Figure 1.3b shows one of many possible 2D projections of the multidimensional space of characteristics (which we consider as costs of running our experiments or tasks). We then gradually track the winning solutions that maximize performance and at the same time minimize all costs using our experience in physics and electronics, namely by applying Pareto frontier filter [90].

P1) **Intel** Core i5-2540M, 2.60GHz, 2 cores
P2) **Qualcomm** MSM7625A FFA, **ARM** Cortex A5, 1 GHz, 1 core
P3) **Allwinner** A20 (sun7i), ARM Cortex A7, 1.6GHz, Mali400 GPU, 2 core
P4) **NVidia** Quadro NVS 135M, 400MHz, 16 cores

W1) **32 bit** processor mode
W2) **64 bit** processor mode

X1) **GCC 4.1.1**, opt.flags~190, release date=2006
X2) **GCC 4.4.1**, opt.flags~270, release date=2009
X3) **GCC 4.4.4**, opt.flags~270, release date=2010
X4) **GCC 4.6.3**, opt.flags~320, release date=2012
X5) **GCC 4.7.2**, opt.flags~340, release date=2012
X6) **GCC 4.8.3**, opt.flags~350, release date=2014
X7) **GCC 4.9.1**, opt.flags~357, release date=2014
X8) **LLVM 3.1**, release date=2012
X9) **LLVM 3.4.2**, release date=2014
X10) **Open64 5.0**, release date=2011
X11) **PathScale 2.3.1**, release date=2006
X12) **NVidia CUDA Toolkit 5.0**, release date=2012
X13) **Intel Composer XE 2011**, cost = ~800euro
X14) **Microsoft Visual Studio 2013**

T1) 7.2E10
T2) 9.6E9
T3) 2.4E9
T4) 1.0E9

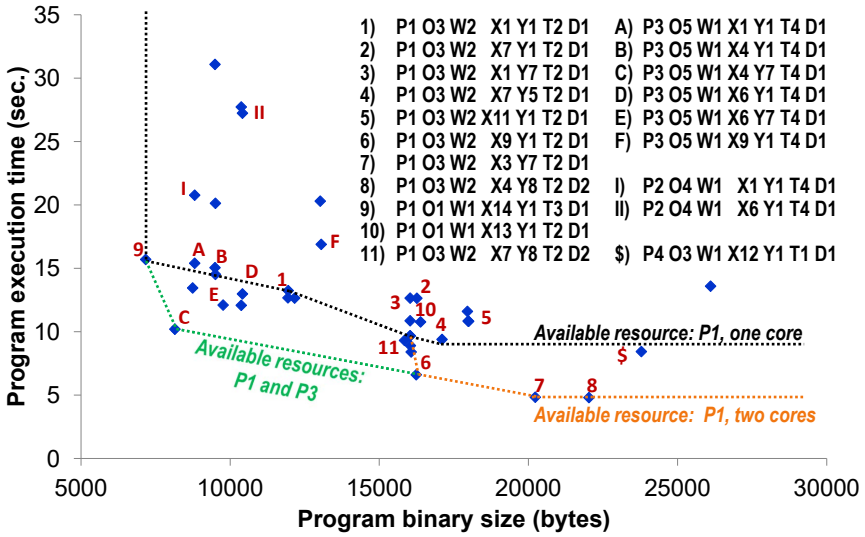
D1) grayscale **image 1**, size=1536x1536
D2) grayscale **image 2**, size=1536x1536

O1) **Windows 7 Pro SP1**, cost~170 euros
O2) O1 with **MinGW32**
O3) **OpenSuse 12.1**, Kernel 3.1.10
O4) **Android 4.1.2**, Kernel 3.4.0
O5) **Android 4.2.2**, Kernel 3.3.0

S1) **Dell Laptop** Latitude E6320, Mem=8Gb, 52W, 1200 euro
S2) **Samsung Mobile** GT-S6312, Mem=0.8Gb, 5W, 200 euros
S3) **Polaroid Tablet** MID0927, Mem=1Gb, 13W, 100 euros
S4) **Semiconductor neural network**,1.5years development

Y1) **Performance** (usually -O3)
Y2) **Size** (usually -Os)
Y3) -O3 -fmodulo-sched -funroll-all-loops
Y4) -O3 -funroll-all-loops
Y5) -O3 -fprefecth-loop-arrays
Y6) -O3 -fno-if-conversion
Y7) **Auto-tuning** with more than 6 flags (-fif-conversion)
Y8) **Auto-tuning** with more than 6 flags (-fno-if-conversion)

(a)



(b)

Figure 1.3 – *(a) A small subset of various hardware, software, development tools and optimizations used in our research on neural networks in the past 20 years (P - processors, W - processor mode, X - compiler, O - operating system, S - system, T - total number of processed pixels or neurons, D - software data set, Y - compiler optimization used) (b) 2D projection of the multidimensional space of characteristics together with winning solutions on the Pareto frontier (all data and interactive graphs are available at c-mind.org/nnet-tuning-motivation).*

We quickly realized that in contrast to the traditional wisdom, the latest technology is not necessarily the fastest or most energy efficient and further optimization is always required. For example, when moving from GCC 4.1.1 (released in 2006) to GCC 4.9.1 (released in 2014) , we observed a modest 4% improvement [4] in single core

[4]Similar to physics, we execute optimized code many times, check distribution of characteristics for normality [43], and report expected value if variation is less than 3%.

execution time of our neural network and 2% degradation in a code size on Intel E6320 based system (released in 2008). However, 8 years old GCC 4.1.1 can achieve 27% improvement in execution time after auto-tuning (which comes at cost of 100 recompilations and executions as well as increasing binary size by 34%)! Interestingly, 8 years old PathScale 2.3.1 produces faster code than the latest version of GCC 4.9.1 and LLVM 3.4.2! Furthermore, when using internal parallelization, LLVM 3.4.2 beats GCC 4.9.1 by about 23% but has a sub-linear scaling versus number of threads. In contrast, 2 years old GCC 4.6.3 achieves the best result and linear scaling versus number of threads when using both parallelization and auto-tuning.

When running the same code on cheap, commodity mobile phones with ARM architecture, the execution time increased dramatically by around 5 times! However, the power consumption dropped by about 10 times! When trying to use specialized hardware (GPUs or our semiconductor neural networks), we could increase execution time by about tens to hundreds of times, but at a considerable development cost and time to market. Furthermore, with time, we discovered that the same best found optimization for one class of images can considerably degrade performance on another class of images. We also encountered problems with cache contentions on multi-core systems, sub-linear scaling on many core systems, unexpected frequency scaling, non-deterministic I/O for large images, and many other problems that had to be addressed by new optimizations. These issues can not be easily solved by static compilers due to a fundamental problem of a lack of run-time information at compile time. Therefore, we even tried to move to dynamic and possibly adaptive languages including Java and Python but were not yet able to achieve similar performance while spending even more energy and storage during just-in-time compilation.

Sadly and similar to many other scientists and software engineers, we now have to waste considerable amount of our time on a tedious and ad-hoc navigation through the current technological chaos to find some good hardware, software and optimization solutions that can speed up our programs and reduce costs instead of innovating as conceptually summarized in Figure 1.4.

Worse, software engineers are often not even aware of all available design and optimization choices they have, to improve performance of their software and reduce development and usage costs. Furthermore, costs that has to be minimized depend on usage scenarios: in mobile systems running out of battery, one may want to fix a power budget and then balance execution time and algorithm accuracy; in embedded devices, code size and consumed energy may be more important than execution time; JIT may require careful balancing of compilation and optimization times versus potential performance gains, while users of data centers and supercomputers may care primarily about both execution time and the price of computation. Therefore, we strongly believe that current performance- and cost-blind software engineering has to be changed to improve productivity and boost innovation in science and technology.

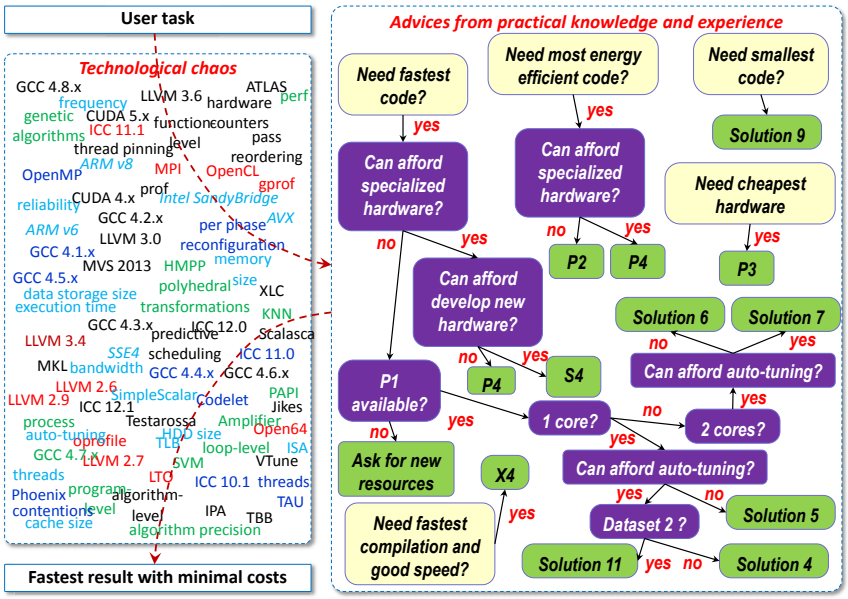


Figure 1.4 – *Example of gradually and manually crafted advices as decision trees to deliver best performance and cost for our neural network depending on available resources, usage scenarios (requirements) and data sets.*

1.6 Research objectives

The research objectives of this thesis are as follows.

1. Tuning compiler optimizations using machine learning which has the potential of automatically adapting the internal optimization heuristic at function-level granularity to improve execute time, code size and compilation time of a new program on a given architecture.

2. Making machine learning based compilation a realistic technology for general-purpose production compilers, extending the scope of currently available machine learning based approaches to support higher number of architectures and compiler flags, aggressive optimizations, and larger number of transformations.

3. To offer computer engineering community practical ways to automatically improve software performance while reducing power consumption and other usage costs across rapidly evolving computer systems.

4. To collaboratively solve the software optimization problems while improving productivity of software developers.

5. Offering scientific community to share their most frequently used software pieces together with various possible inputs and features and optimizing them to allow the interdisciplinary community to collaboratively correlate the best found optimizations with gradually exposed features from the software, hardware, data sets

and environment.

The most closely related work [19] discusses continuously tuning the GCC compiler flags using a data center, however, like most other techniques it uses black box auto-tuning with only several programs while focusing on a few speedups and without any released tools. Finally, authors in [93] suggest to automatically derive combinations of features from a compiler using grammars but only for one optimization (unrolling), has no released infrastructure, and does not include analysis of the scalability of the approach in presence of ever growing number of features, optimizations and programs. Furthermore, the related features are often not even available in a system.

Therefore, to the best of our knowledge, we provide the first simple and practical methodology and open-source framework to unify, formalize and connect together available ad-hoc techniques and tools for auto-tuning and machine learning by using recent advances in agile methodologies, web and crowdsourcing technology, and schema-free repositories.

1.7 Thesis contribution

The scientific contribution in this thesis can be summarized as follows.

1. We introduce a machine learning based compiler (Milepost GCC) which is based on 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. Milepost consists of an interactive compilation interface and plugins to extract program features and exchange optimization data with an open public repository (cTuning.org)[32]. It can automatically adapt 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.

2. On top of Milepost, we develop a collaborative framework, called Collective Mind (cM), which serves as a distributed platform for cooperative formalization and validation of program optimization (auto-tuning) and machine learning to make it understandable, practical, reproducible and scalable.

3. We further envisage a novel experimental methodology in the Collective Mind to continuously optimize and classify multiple code and data set samples shared by the community while exposing, analyzing and solving unexpected behavior either automatically or through crowdsourcing.

4. We implement the Collective Mind with two experimental setups to validate iterative compilation and machine learning in two major companies and on several

mobile embedded devices. The results discussed in the thesis show that the Collective Mind effectively serves as a collaborative platform for sharing codes, data set samples from major benchmarks, and optimizing real applications in terms of execution time using at least 5000 combinations of GCC compiler optimization flags currently deriving 79 distinct and pruned optimization classes.

5. As an elaborated proof of the concept, we also present a case study in an industrial setup, where we exposed our framework's mispredictions on a production code to domain specialists who "deconstructed" and isolated the problem, prepared and shared counter-example benchmark, and learned correct algorithm, program and data set features to fix wrong classification.

6. Our white box approach helps to deliver minimal representative benchmark to an architecture verification and testing department of our industrial partner. The community now has an extensible tool set to continue analyzing all exposed problems and find new features to improve and optimize predictive models.

1.8 Thesis organization

The rest of the thesis is organized as follows.

Chapter 2 gives the theoretical background of compiler optimization and its several techniques. We also discuss the theoretical aspects of the key machine learning concepts used in our experimental framework. At the end, we give an overview of crowdsourcing which is an integral part of our collaborative framework.

Chapter 3 provides details of our experimental setup used for various evaluations in the thesis. We then discuss compiler auto-tuning using iterative compilation and show that iterative compilation does effectively tune optimization heuristics of a compiler, but results in excessive search costs and execution time that motivate the use of machine learning to learn optimizations across programs based on their features.

Chapter 4 provides an overview and detailed architecture of our machine learning based compiler, Milepost GCC. We briefly discuss its prediction models along with its evaluation and comparison with iterative compilation.

Chapter 5 introduces the framework of our collaborative framework, the Collective Mind. We also discuss the essentials of cooperative research and experimentation as well as its major challenges. We then provide a comprehensive overview of our public and open-source Collective Mind infrastructure and repository. At the end of the chapters, we discuss several possible usage scenarios.

Chapter 6 provides a detailed discussion about crowdsourcing, feature learning and model improvement with our collaborative framework. We present several public research scenarios and experimental pipelines as well as a detailed discussion on learning data set features to enable adaptive software.

Chapter 7 summarizes the overall thesis and describes the potential future work.

2

Background

2.1 Introduction

Compiler optimization is the core concept of this thesis. Hence, this chapter provides an in-depth mechanism of optimizing compiler, its various transformations, and the popular compiler optimization technique, the iterative compilation. Subsequently, we describe the issues pertaining to iterative compilation and present machine learning as a potential solution to cope with these issues. Since our proposed optimization framework is based on machine learning, we provide theoretical background of several machine learning algorithms used in our proposed framework. At the end, we describe the concept of crowdsourcing which forms an integral part of our proposed optimization methodology.

2.2 Compiler basics

A *compiler* is a program that translates from one input language, the source language (e.g., C), to another output language, the target language (e.g., machine code for Pentium processor series) [64]. Generally, a compiler consists of two parts: (i) *analysis* part that decomposes a source code and applies a grammatical structure on the constituent parts to create an intermediate form, and (ii) *synthesis* part that generates the target code from the intermediate form [3].

Besides these two high-level parts, a compiler generally operates in multiple low-level phases shown in Figure 2.1 and described as follows [3].

1. *Lexical analyzer* reads character stream of the source program and groups the characters into meaningful sequences called *lexemes*. A token is formed for each

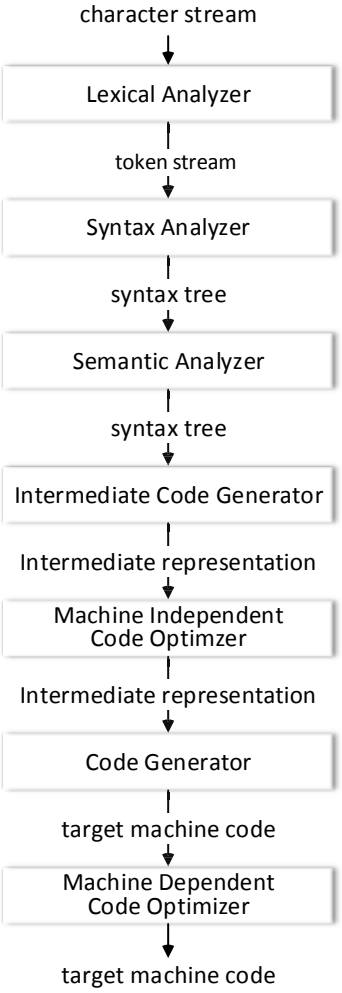


Figure 2.1 – *Phases of compiler operation*

lexeme (consisting of token class and an optional attribute) and passed to the subsequent phase.

2. *Syntax analyzer* generates a syntax tree from the tokens generated by the lexical analyzer that represents the grammatical structure of the token stream.

3. *Semantic analyzer* checks the semantic consistency with the language definition from the information contained in the syntax tree. The relevant information is saved either in the syntax tree or in a *symbol table*.

4. *Intermediate code generator* constructs one or more explicit low-level or machine-like intermediate representation of the source program which is reproducible and easy to translate in to the target machine.

5. *Code optimizer* attempts to improve the intermediate code for length reduction, speed and power efficiency. Code optimization can be machine- dependent or independent depending on whether the code is being optimized for a specific machine or independent of the platform.

6. *Code generator* maps the optimized intermediate code to the target language and assigns registers to hold variables.

The core concept of the work proposed in this thesis is code optimization which varies from compiler to compiler. The most aggressive form of code optimization coins the concept of optimizing compiler which spends significant amount of time in the optimization phase.

2.3 Optimizing compiler

With the advent of high-performance embedded devices having sophisticated computing components such as multiple Digital Signal Processors (DSPs), Graphics Processing Units (GPUs) and other co-processors, the aggressive compiler and software optimization has become indispensable. Though, code optimization is an integral phase of a compiler, in order to meet the high computational demands and resource constraints of modern computing devices, a rigorous code optimization is required. An optimizing compiler performs an aggressive optimization of programs for reducing their run-time, minimizing occupied memory, and decreasing power consumption [3]. Optimizing compiler uses a sequence of algorithms, called optimizing transformations, on a given source program to improve it in terms of execution time, code size, memory footprint and power consumption.

2.3.1 Optimizing transformations

The programs written for older platforms do not reflect the design features of new hardware designs and are ported to the new platforms without major changes due to economical reasons. Hence, these programs suffer performance degradation on new platforms and require to be improved to adapt to the new computing platform. This is achieved by optimizing transformations which is a method of re-arranging a program's operations in order to improve its performance on the new platform without changing the meaning of the program [51]. Some optimizing transformations used in the research of this thesis are discussed in the following sections.

2.3.1.1 If-Conversion

If-conversion transforms all the control dependencies in a program into data dependencies. This is achieved by removing a branch instruction around an instruction with a predicate on the instruction. The instructions having *true* predicate execute as though they are not predicated. On the other hand, the instructions having *false* predicate execute as NOP instructions [21]. Following example shows the process of if-conversion on a block of code. The code before if-conversion contains a branch which is replaced by the if-conversion with predicated execution. Instead of testing a branch condition

and executing only a single control-dependent path, both paths are executed and the results are controlled by the predicates. The effect of if-conversion transformation can be seen in Figure 2.2.

```
1  /* Before If-conversion */
2  if (cond) Branch B1
3  v2 = MEM[x];
4  v1 = v2 + 1;
5  v0 = MEM[v1];
6  B1: v5 = v3 + v4;
```

```
1  /* After If-conversion */
2  c1, c2 = cond;
3  v2 = MEM[x] <c2>;
4  v1 = v2 + 1 <c2>;
5  v0 = MEM[v1] <c2>
6  B1: v5 = v3 + v4;
```

2.3.1.2 Loop unrolling

Loop unrolling attempts to minimize a loop's overhead incurred due to branch instructions by reducing the number of instructions that control the loop and replicating the body of the loop. The instructions forming the loop's body must be independent in order to avoid data dependency. Loop unrolling results in reduction of the total number of instructions executed by the CPU when the loop is executed. It is mainly because the number of branch instructions and the loop control overhead (increment in the control variable, testing condition to terminate the loop) is minimized. Loop unrolling significantly improves a program efficiency by achieving increased instruction-level parallelism and effective utilization of spatial locality of data items in memory [34]. Following simple example demonstrates loop unrolling. After unrolling, the following code block is executed 20 times instead of 100.

```
1  /* Before loop unrolling */
2  for (i = 0; i < 100; i = i + 1)
3  {
4      x[i] = sqrt(x[i]);
5  }
```

```
1  /* After loop unrolling */
2  for (i = 0; i < 100; i = i + 5)
3  {
4      x[i] = sqrt(x[i]);
5      x[i + 1] = sqrt(x[i]);
6      x[i + 2] = sqrt(x[i]);
7      x[i + 3] = sqrt(x[i]);
8      x[i + 4] = sqrt(x[i]);
```

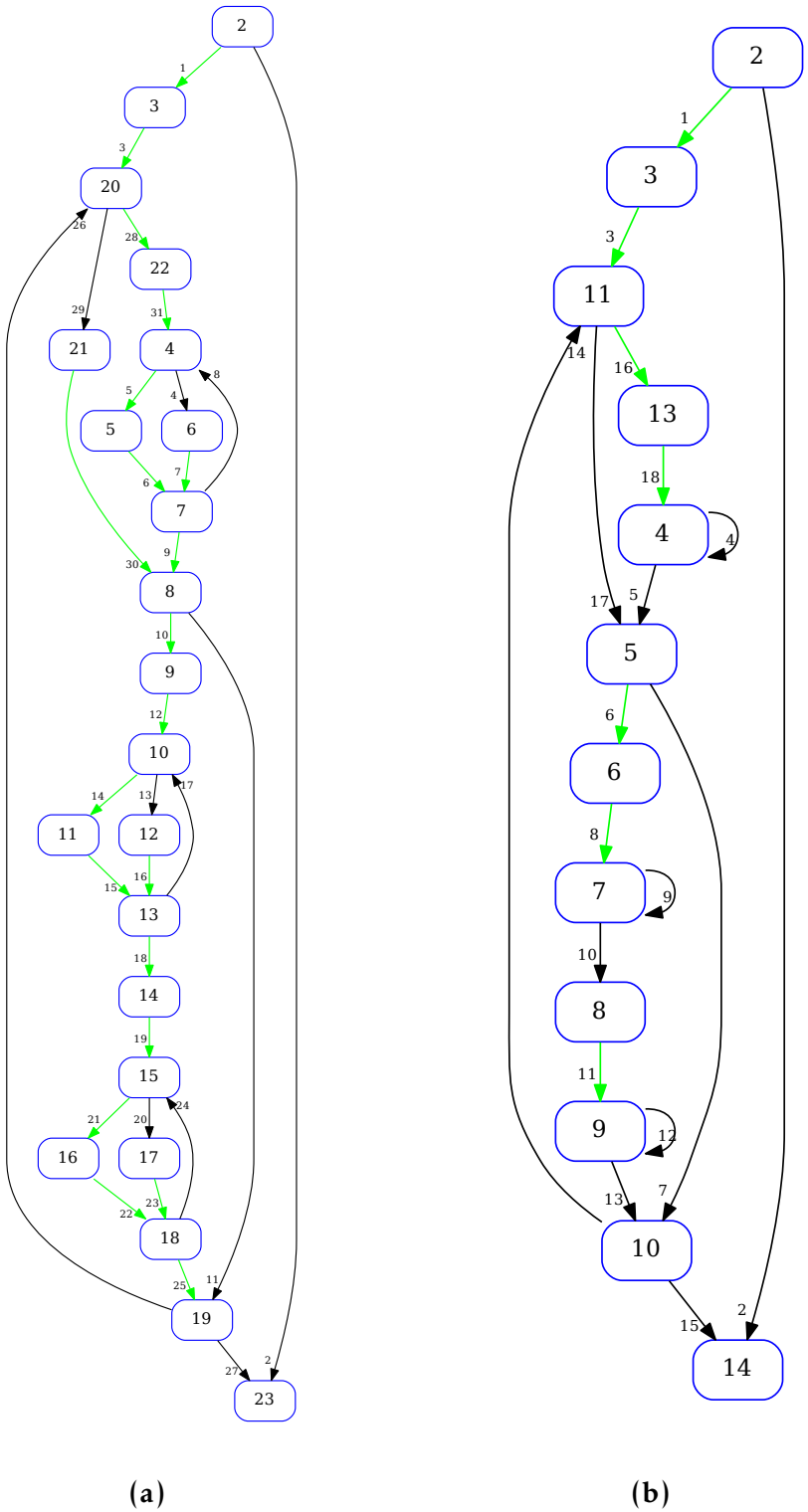


Figure 2.2 – *Control flow graph of mc.codelet-9.1 constructed using a) gcc-4.6.3 -O3 b) gcc-4.6.3 -O3 -fno-ifconversion*

```
9 }
```

2.3.1.3 Guessing branch probability

This transformation determines the probability of occurrence of conditional branches in a program code. Based on the likelihood of the branches, the instructions are reordered accordingly. The branches can be predicted using profile-based techniques which involves producing a profile by several runs of a program and reordering the code after analyzing the profile. Hence, profile-based branch prediction are cumbersome and time-consuming [6]. The compiler generally uses heuristics to guess the branch probabilities and predict the code arrangement during compile time. Nevertheless, this may result in different object code for the same program during different compilations [10].

2.3.1.4 Function inlining

Function inlining is an optimization technique that aims to reduce the overhead involved in calling a function and returning from it. This technique transforms all the discrete functions of a program into a single function which is embedded directly in the code structure where it is used to eliminate the function calling overhead such as saving the state of the function, argument passing and retrieving function state from the stack [36, 112]. Function inlining significantly improves a program's performance in terms of execution speed, memory utilization and energy consumption [86].

Consider the following code fragment.

```
1 float addition(float a, float b)
2 {
3     return a + b;
4 }
5
6 float subtract(float a, float b)
7 {
8     return addition(a, -b);
9 }
```

The function *addition*() can be expanded in function *subtract*() as follows:

```
1 float subtract(float a, float b)
2 {
3     return a + - b;
4 }
```

This expansion can be further optimized as:

```
1 float subtract(float a, float b)
2 {
3     return a - b;
```

```
4 }
```

Selecting an optimal combination of transformations presents a dedicated challenge in compiler optimization. A transformation may result in overall performance improvement in terms of execution time. On the other hand, it may drastically increase the execution time of a program. In the same way, a combination of various transformations selected under a compiler flag (for e.g. -O3) may or may not provide execution speedup. We can not simply turn on all the transformations as a rule of thumb to achieve a maximum speedup. Due to hundreds of possible transformations under a compiler flag, determination of right combination of transformations is not trivial. Figure 2.3 shows the speedup achieved by enabling and disabling a specific transformation for a benchmark. It is evident that while enabling a transformation for a specific benchmarks does result in execution speed improvement, it increases the execution time for other benchmark.

Transformation	Codelet	Enabled	Disabled	Speedup
if-conversion	video_x264_mc.codelet	5.94s	4.65s	1.28
	Powerstone_auto2	5.36s	7.49s	0.85
unroll-loops	UTDSP_compress_arrays_SWP_4.1	5.94s	4.65s	1.28
	SNU_RT_fibcall_1.1	5.36s	7.49s	0.85
guess-branch-probability	Powerstone_auto2	5.94s	4.65s	1.28
	Video_x264_pixel	5.36s	7.49s	0.85

Figure 2.3 – *Speedup achieved by enabling and disabling various transformations. Above listed transformations are enabled by default at -O3.*

2.4 Iterative compilation (auto-tuning)

Iterative optimization is a popular approach to adapting programs to new architectures automatically using feedback-directed compilation [56]. This optimization approach searches for the best optimization scheme for a target machine from a given set of optimizations. This is achieved by compiling a program with a selected set of transformations and their parameters and evaluating its impact on the target machine. The evaluation is performed with a cost model which serves as the objective function of the optimization process and determines the effectiveness of a compiled code on the target machine. The process of iterative compilation continues until it converges to a minimum cost which represents the best optimized code [99].

Figure 2.4 depicts the process of feedback-directed iterative optimization. The next combination of the optimization based on the evaluation feedback is selected either randomly by generating a random number, or by assigning pre-computed probabilities to each optimizations. The probabilities of the optimizations change with respect to the evaluation feedback and the good optimizations are selected with higher probabilities [139]. Although feedback-directed iterative compilation is a natural way to

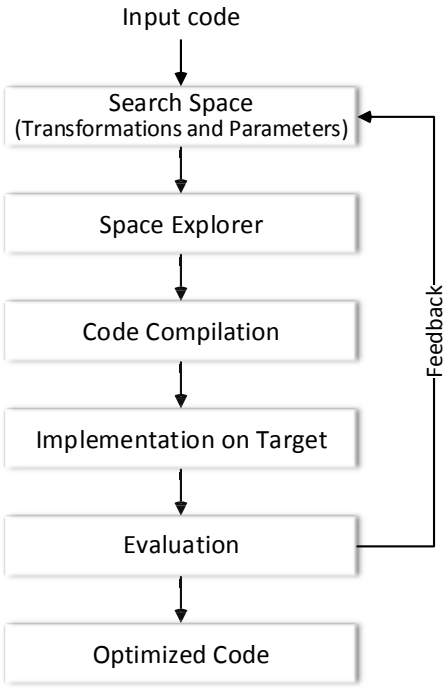


Figure 2.4 – *Feedback-directed iterative compilation*

select the best optimization scheme for a target machine, it does have the following disadvantages.

- In order to converge to the best optimization for a target machine, the search space of iterative compilation must contain all the possible combinations of optimizations. The huge search space results in a long compile and execution time to optimize a given program which makes it infeasible for iterative compilation to achieve a wider adoption by the modern industrial compilers [139, 56].
- It is difficult to build analytical models that can predict how different combinations of optimizations will interact with each other [116].
- Iterative compilation often depends on the selection of the data set. With the change of input data set, the performance of the target application may suffer significant degradation [99].

In contrast to feedback-directed iterative compilation, machine learning can successfully address the above mentioned issues by learning optimization strategies using prior knowledge of other programs' behavior. Following section provides a theoretical background of some machine learning techniques pertinent to our proposed machine learning based compiler optimization framework.

2.5 Machine learning for tuning compiler optimization

Although iterative compilation is theoretically able to find the optimal compilation settings needed to ensure portable performance on a specific architecture, it is costly in terms of optimization time. Moreover, the search space is too large to explore all possible optimizations. In order to address the issues pertaining to iterative compilation, machine learning can be exploited to reuse the knowledge across iterative compilation runs, gaining benefits of iterative compilation while reducing the number of executions needed. Unlike iterative compilation that operates on a given static model, machine learning operates on building an exclusive prediction model from the given program features and corresponding optimizations in a dataset to make data-driven decisions. Machine learning can be used in two possible ways for compiler optimization: supervised learning and unsupervised learning.

- Supervised learning requires an offline dataset comprising program features with accurately labelled classes of optimization. This approach requires collection of various program features and experimentally determining their appropriate classes (optimizations). A drawback of this approach is that it is able to deal with only those observations which are included in the training dataset and largely mispredicts the new instances. A possible solution to this problem is to incorporate new observations (program features) and re-train the learning framework on the fly to adjust the prediction model. Examples of supervised learning algorithms include Support Vector Machine (SVM), decision trees and K-Nearest Neighbor (KNN), to name a few.

- In contrast to supervised learning, unsupervised learning builds a prediction model by directly interacting with the learning environment without requiring any given dataset. In this learning framework, an optimization is randomly applied to a given program code and its outcome is analyzed to adjust the future decisions. In this way, the learning proceeds in a direction in which better actions (optimizations) are selected as more observations are analyzed. This learning mimics human learning process which starts out with random actions and rectifies its decisions as more experience is acquired. However, despite of its appealing features, the time required to build an optimal prediction model for compiler optimization using unsupervised learning is yet to be explored. Examples of unsupervised learning include reinforcement learning, artificial neural networks and deep learning, to name a few.

While collection and labelling of dataset offers a dedicated challenge in supervised learning and turns out to be cumbersome, dealing with unforeseen observations is another issue. It is nearly impossible to incorporate all the observations and scenarios in the learning dataset. Likewise, starting out with no information (dataset) at hand and

proceeding to optimal optimization decisions with reasonable convergence time is yet another challenge in unsupervised learning. Hence, it is evident that none of the above mentioned learning paradigms can be significantly effective if used independently.
The work carried out in this thesis is motivated by augmenting supervised learning with a collaborative framework to incorporate users' experience in the learning model and constructing a dynamic prediction model for compiler optimization. Starting out with a labelled dataset of program features, we train the learning algorithm with supervised learning. In order to refine the learning accuracy and prediction model, new observations are obtained from users' experience through crowdsourcing (see Section 2.6). While the supervised learning directs the prediction model to the right path, the feedback obtained from users' experience and incorporation of new information to the learning model further improves the prediction accuracy over time.

Before presenting a detailed description of our machine learning based compiler optimization technique, it is pertinent to discuss the theoretical background of some machine learning techniques that can be potentially used for compiler optimization.

2.5.1 Decision trees

A decision tree is a predictive model and a supervised learning technique in the form of a graph where each node (called tree's leaf) represents a test on an attribute (e.g., whether to unroll a loop or not) and its possible outcomes. The classification task starts out with the tree's root with the initial test on its attributes and follows the appropriate branches. Tree traversal continues until the classification process encounters a leaf and the given observation/example is mapped to the leaf's class. Given the adequate number of attributes, a decision tree maps a training set example to its appropriate class with a high accuracy. However, an important aspect of training a decision tree is to find the relationship between a class and its attributes such that the decision tree should be able to correctly classify the examples not only from the training set, but also the unseen examples. For a given problem, the number of different decision trees that correctly classify a given example is usually very high. However, as a rule of thumb, the decision tree with the simplest structure is known to better capture the structure inherent in the problem [119]. Having said that, inadequate number of training examples may cause the decision trees to overfit the training data [105]. In general, decision trees may suffer from fragmentation, repetition and replication [80].

Figure 2.5 depicts the decision tree of *-falign-functions* optimization. The four static program features (see Table 4.1) on which the optimization is performed are represented by *ft11*, *ft22*, *ft32* and *tf46* and defined as follows: (i) *ft11* represents the number of basic blocks with number of instructions less then 15, (ii) *ft22* is the number of binary integer operations in the program, (iii) *ft32* is the number of basic block where total number of arguments for all phi-nodes is in the interval [1, 5], and (iv) *ft46* is the number of occurrences of integer constant zero. The decision tree in Figure 2.5 shows that based

on the values of the features, how many of the given examples will go through *-falign-functions* optimization. In the figure, $-1(i)$ represents the number of programs which will not be optimized based on the tested condition at each node. After traversing the tree, only 6 programs out of 21 are found to be eligible for *-falign-functions* optimization.

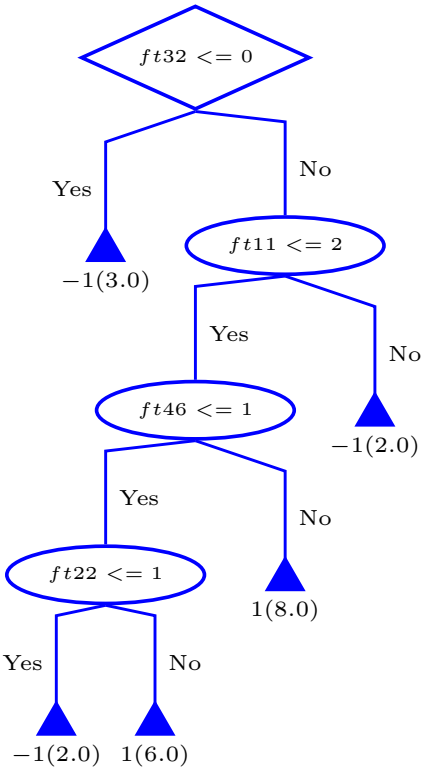


Figure 2.5 – *Decision tree of -falign-functions optimization*

2.5.2 K-nearest neighbor (KNN)

K-nearest neighbor is a classification and regression technique that maps a given observation to k training samples which are closest in distance to the given example. The distance of the given example and the k training samples can be Euclidean distance (in case of continuous variables) or Hamming distance (for discrete variables). The value of k selected for classification depends on the problem and is calculated by heuristics.

The KNN problem can be mathematically described by Equation 2.1,

$$\hat{C}(x) = \frac{1}{k} \sum f_i \in N_k(x) c_i \tag{2.1}$$

where $f = (f_1, f_2, ..., f_n)$ is the input feature vector, $\hat{C}$ is the predicted class, and N_k is the neighborhood of f defined by the k nearest neighbors f_i in the training set and c_i is the class label of the i^{th} instance of the training set.

Figure 2.6 depicts the mapping of a given object represented by a star to its appropriate class with respect to the chosen neighborhood radius k. For $k = 3$, the object is mapped to class A because there are more instances of class A in the vicinity of the object surrounded by the radius k than class B. On the other hand, for $k = 6$, the object is mapped to class B.

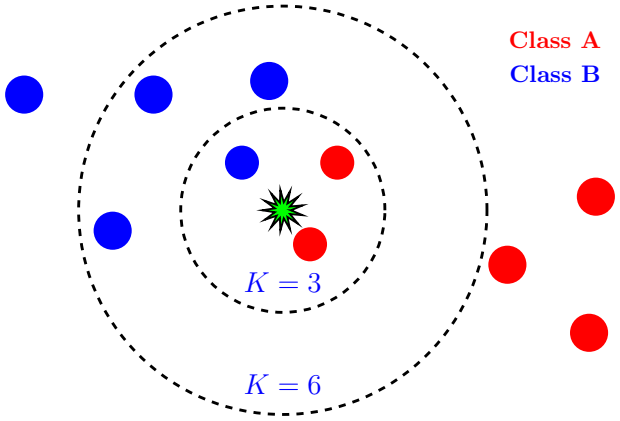


Figure 2.6 – *Classification with KNN algorithm*

It is evident that the classification accuracy of KNN algorithm is not only subjected to the neighborhood radius k, but also largely depends on the nature of feature space. Noisy data can drastically reduce the classification accuracy. Hence, the training data requires to be scaled and pruned appropriately before the training [139].

2.5.3 Support vector machine (SVM)

Support vector machine is a supervised learning approach to learn classification and regression problems. SVM uses the principles from statistical theory to estimate a function from a set of training examples, each containing a feature vector and the corresponding label (class). In order to find the mapping between a given observation and its class, SVM selects one function, from a given set of functions, which minimizes a certain *risk* that the estimated function is different from the actual function [123].

To learn the best compiling settings, the feature vectors may comprise various program characteristics such as trip count of loops, number of operations in a loop body, the programming language, nesting levels of loops, number of instructions in a method, number of branches, performance counters, microarchitecture- dependent or independent characteristics, reactions to transformations, etc [122, 56]. The corresponding classes in the training data may comprise sets of transformations, compiler flags, etc that can be applied to the given program features.

SVM attempts to separate the classes by a function induced from the training data set and to generalize a classifier for new observations. The goal of SVM optimization is to separate the classes such that the distance between the nearest data points from all the classes (called margin) is maximum. Without losing generality, in case of a two-class

problem, we can perceive a boundary separating the two classes and representing a classifier. Among various possible classifiers, one that maximally separates the classes by maximizing margin is called optimal separating hyperplane, as shown in Figure 2.7. The data points from two classes, represented by vector x, are separated by two hyperplanes. The maximum margin between the two hyperplanes is given by $\frac{2}{\|w\|}$, where w is the normal vector to the hyperplanes. The objective of the SVM is to maximize the margin by minimizing $\|w\|$ with the constraint that there are no data points between the two hyperplanes.

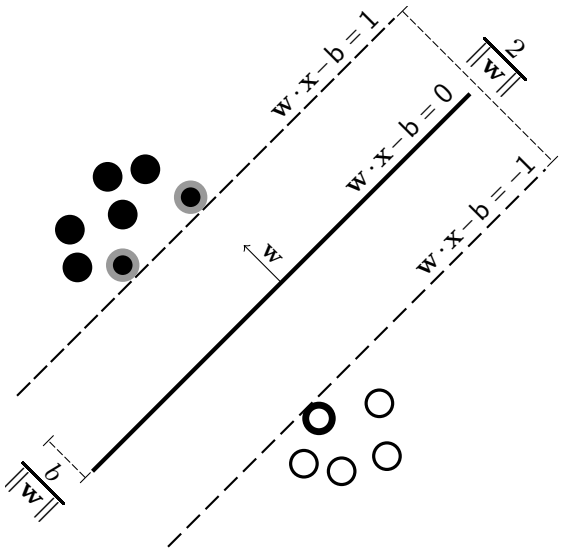


Figure 2.7 – *Data points separated by two maximum-margin hyperplanes in ax two-class SVM problem*

2.6 Experiment crowdsouring

Crowdsourcing forms an integral part of the collaborative compiler auto-tuning approach proposed in this thesis. It is an approach to combine human knowledge and expertise with computing in order to cope with the availability of comprehensively labeled data sets and expressive evaluation strategies. This is particularly helpful for collecting training data for machine learning from the experiences of human experts. By this way, we can continuously improve the prediction accuracy of machine learning algorithms and the relevant useful information obtained from the continuously evolving prediction models can be crowdsourced to the community. The basic motivation for the volunteers taking part in this collaborative effort lies in challenging their skills to solve a problem for a greater cause and serve the community.

Crowdsourcing is being used in conjunction with machine learning in a wide range of domains where the discovery of missing features is of primary concern. Zou et al. [140] proposed a formal framework for modeling feature discovery with a data set using crowd queries. They successfully extracted salient feature names along with their labels on the

data set. In another work [120], the authors used a combination of machine learning and crowdsourcing for autonomous driving. Crowd contribution is utilized for collecting complex 3D labels and tagging diverse scenarios for the evaluation of learning systems. Wu et al. [138] used machine learning with crowdsourcing for better understanding customer reviews. They use different machine learning algorithms to process reviews from an offline data. The reviews with different prediction results from the algorithms are passed to the human volunteers and their opinions are aggregated for the final analysis. Other domains where machine learning has been used in combination with crowdsourcing include collaborative audio enhancement [87], predicting the quality of new contributors to the social networks [87], disaster relief systems [61], paraphrase acquisition [13], classification of galaxies (Galaxy Zoo project) [84], online games [85] and entrepreneurship [8], to name a few.

Having inspired from the benefits of crowdsourcing in machine learning, we utilize crowdsourcing in our proposed collaborative framework for compiler auto-tuning, the Collective Mind (cM), to distribute analysis and multi-objective off-line and on-line auto-tuning of computer systems among many participants while utilizing any available smart phone, tablet, laptop, cluster or data center. This is immensely effective in continuously observing, classifying and modeling their realistic behavior. With this technique, we can easily distribute various optimization scenarios among many participants and continuously explore available optimization choices for all shared code and data set samples from the community in realistic environments while focusing on unexpected behavior and mispredictions. All behavior anomalies are continuously collected and exposed in a centralized repository to find most optimal predictive models and correlating algorithm, program, architecture, data set and other features for a given scenario either automatically or through crowdsourcing as it is currently successfully used in other sciences including biology and artificial intelligence. The crowdsourcing approach used in the Collective Mind is discussed in detail in Chapter 5.

2.7 Summary

In this chapter, we discussed the basics of compiler architecture and argued why compiler optimization is indispensable. We also discussed a special type of compiler, the optimizing compiler, which is capable of performing several types of optimizing transformations. We further discussed iterative compilation which is the most popular type of compiler optimization but suffers from long compiling and execution time. We then discussed some machine learning techniques which can be potentially used for tuning compiler optimization. At the end, we described the theoretical background of crowdsourcing technique which is a part of our proposed collaborative framework for compiler optimization.

3

Compiler auto-tuning

3.1 Introduction

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. The increasing complexity of compiler optimization over time is evident from the rapid increase in compiler optimization flags and their corresponding parameters as shown in Figure 3.1 for GCC. We demonstrate that iterative compilation can effectively solve this problem, however often with excessive compiler optimization space search costs.

The chapter is organized as follows. The next section describes the experimental setup used for compiler auto-tuning. We then describe how iterative compilation can deliver multi-objective optimization.

3.2 Experimental setup

The tools, benchmarks, architectures and environment used in the demonstration of iterative compilation are briefly described in this section. The same experimental setup is also used in the development and evaluation of Milepost GCC described in Chapter 4.

3.2.1 Compiler

We considered several compilers for our research and development including Open64 [26], LLVM/Clang [25, 23], ROSE [28], Phoenix [27], and GCC [24]. GCC was selected as it is a mature and popular open-source optimizing compiler that supports 6+ front ends for popular programming languages, has a large community, is competitive with the best commercial compilers, and features a large number of program transformation

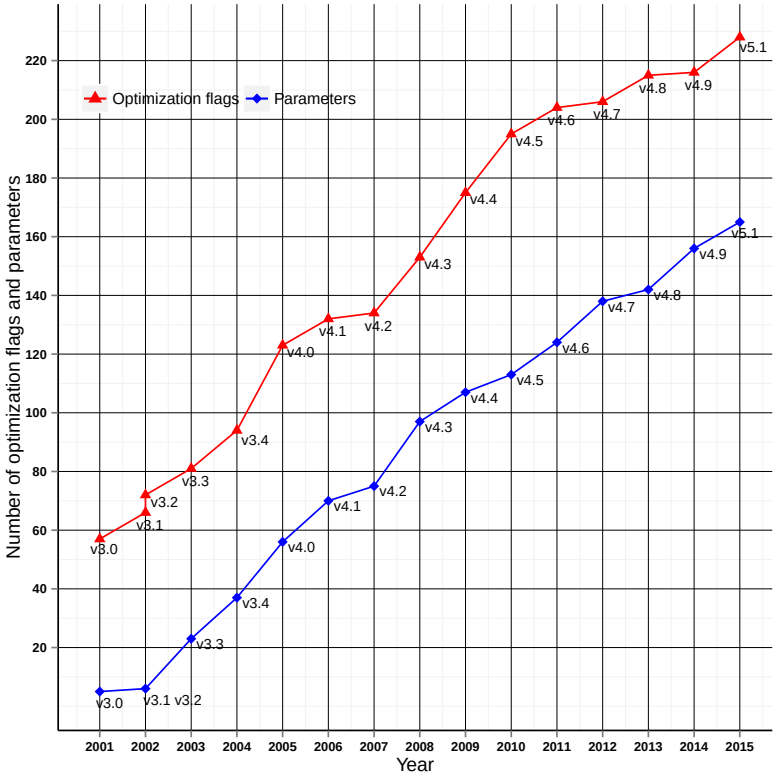


Figure 3.1 – *Evolution of optimization flags and their parameters in GCC*

techniques including advanced optimizations such as the polyhedral transformation framework (GRAPHITE) [134]. 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 GCC 4.4.4 as the base for our machine-learning enabled self-tuning compiler.

3.2.2 Optimizations

There are approximately 225 flags available for tuning in the most recent version of GCC (i.e. v5.1.0), most 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.

3.2.3 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 [63])
- *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.

3.2.4 Benchmarks and experiments

We use both embedded and server processors. Hence, we selected MiBench/cBench [65, 53, 52] 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 [111] 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 future, we plan to use more statistically rigorous approaches [131, 62]. 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.

automotive_bitcount	automotive_susan_c	automotive_susan_e
bit_shifter (32.3%)	susan_corners (98%)	susan_edges (83%)
bit_count (20.9%)		susan_thin (12%)
ntbl_bitcnt (18%)		
automotive_susan_s	**automotive_qsort1**	**consumer_jpeg_c**
susan_smoothing (99.9%)	swap (33%)	encode_mcu_AC_refine (28.8%)
	compare (16.5%)	encode_mcu_AC_first (11.6%)
		jpeg_gen_optimal_table (10.5%)
consumer_jpeg_d	**consumer_tiff2bw**	**consumer_tiff2rgba**
jpeg_idct_islow (37.2%)	LZWDecode (70.3%)	LZWDecode (74%)
ycc_rgb_convert (21.9%)	compresscontig (13.3%)	horAcc8 (10.5%)
decode_mcu (15.5%)		
consumer_tiffdither	**consumer_tiffmedian**	**office_stringsearch1**
LZWDecode (21%)	create_colorcell (51%)	strsearch (85%)
find1span (18.5%)	get_histogram (11%)	
fsdither (17.9%)		
find0span (14.1%)		
Fax3Encode2DRow (12.6%)		
network_dijkstra	**network_patricia**	**network_blowfish_d**
dijkstra (45.3%)	bit (26.3%)	BF_encrypt (66.4%)
enqueue (12.8%)	pat_insert (11.6%)	BF_cfb64_encrypt (33%)
	pat_search (8.9%)	
network_blowfish_e	**security_rijndael_d**	**security_rijndael_e**
BF_encrypt (68%)	decrypt (62.4%)	encrypt (57.8%)
BF_cfb64_encrypt (31.3%)	decfile (11.9%)	encfile (14%)
telecom_adpcm_c	**telecom_adpcm_d**	**telecom_CRC32**
adpcm_coder (99.9%)	adpcm_decoder (99.9%)	crc32file (24.6%)
telecom_gsm		
Calculation_of_the_LTP_parameters (50.6%)		
Short_term_analysis_filtering (15.2%)		
Autocorrelation (9.4%)		

3.2.5 Collective optimization database

All experimental results were recorded in the public Collective Optimization Database [33, 52, 60] at cTuning.org, allowing independent analysis of our results.

3.3 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 optimization flags 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 3.2 shows the best execution time speedup achieved for each benchmark over the highest GCC optimization level (-O3) after 1000 iterations across 3 selected architectures.

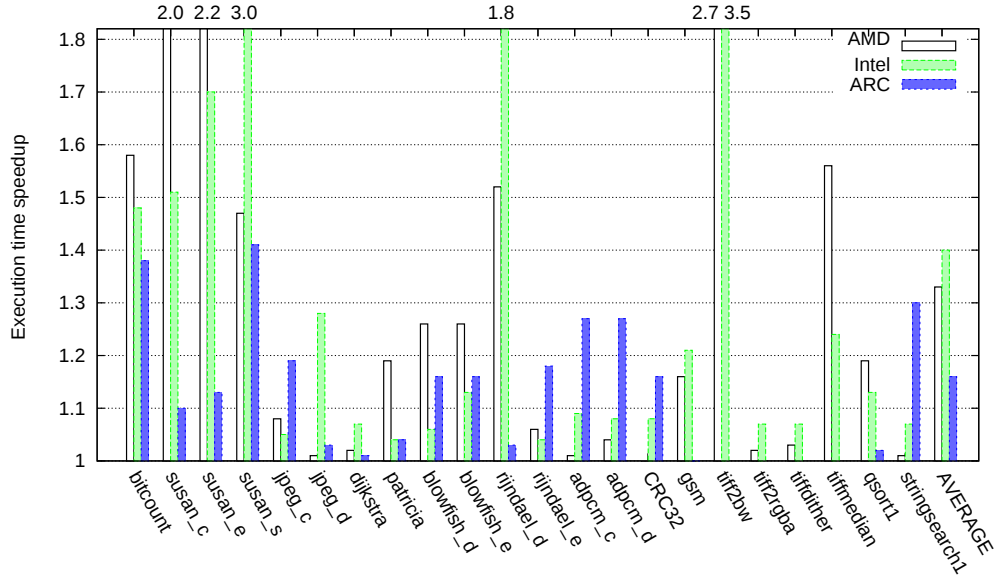


Figure 3.2 – *Maximum execution time speedups over the highest GCC optimization level (-O3) using iterative compilation with uniform random distribution after 1000 iterations on 3 selected architectures.*

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 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.

Figure 3.3 show high variation of execution time speedups, code size improvements and compilation time speedups during iterative compilation across all benchmarks on

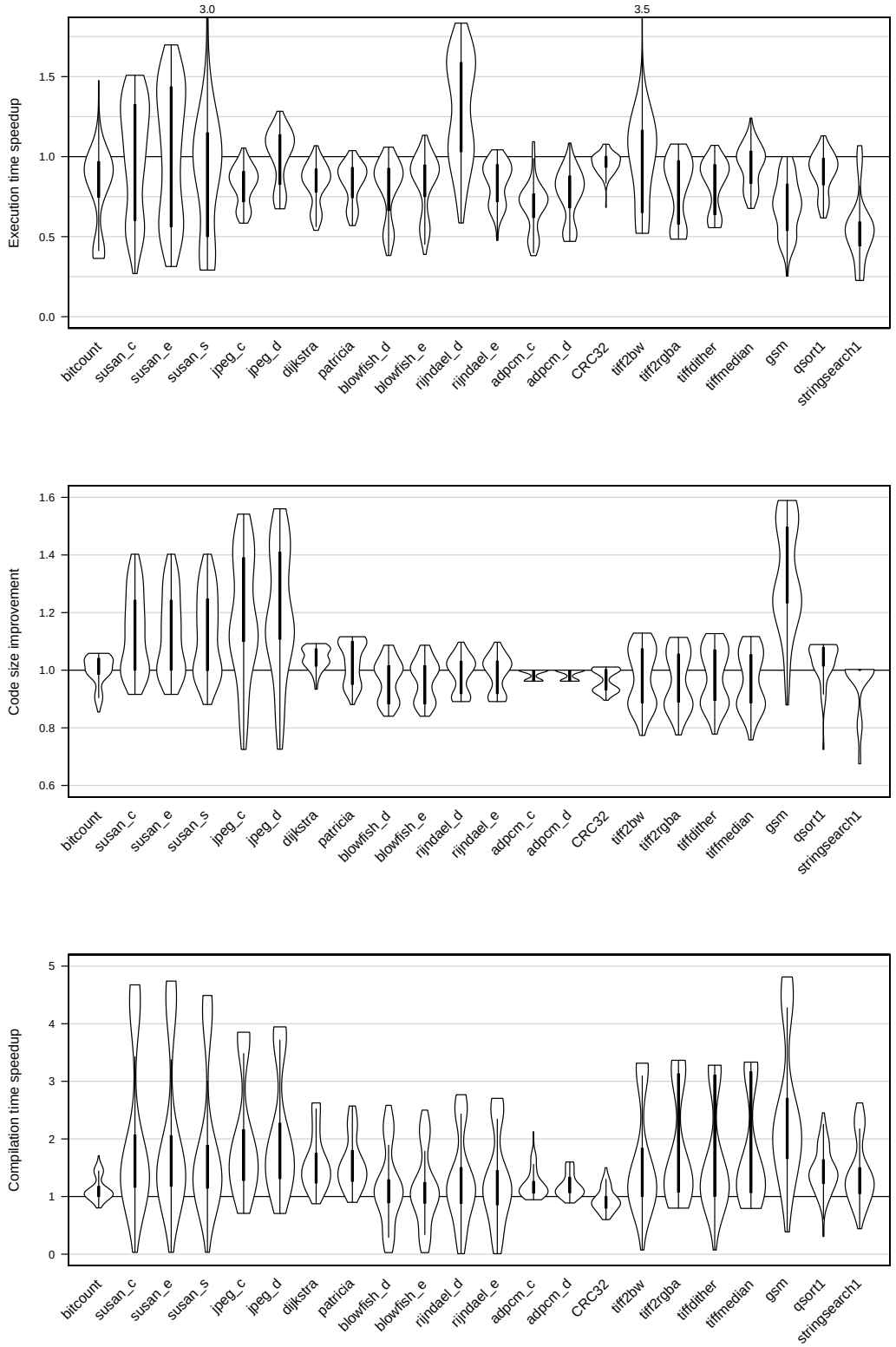


Figure 3.3 – *Distribution of execution time speedups, code size improvements and compilation time speedups on* Intel *platform during iterative compilation (1000 iterations).*

Intel platform as violin graphs [1].

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 3.4, 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.

Figure 3.5 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 execution time, code size and compilation time at the same 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 chapter, 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 [69, 74].

The pruned combinations of flags corresponding to Figure 3.5 which improves execution time (speedup > 1.0), code size and compilation time across all cBench programs and the specified platform architectures are presented in Table 3.1. 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. Some combinations can reduce compilation time by 70% which can be critical when compiling large-

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

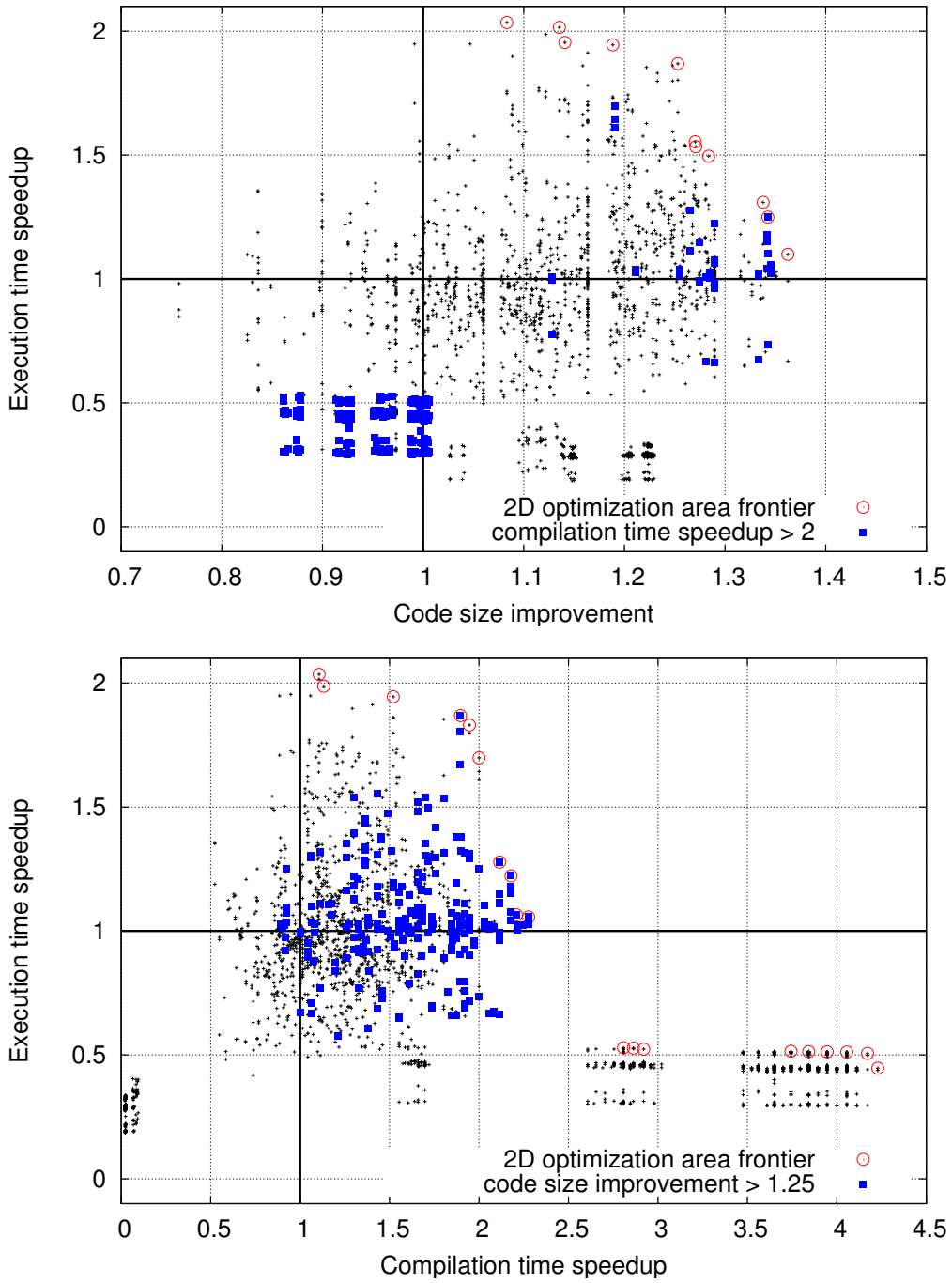


Figure 3.4 – *Distribution of execution time speedups, code size improvements and compilation time speedups for benchmarks* susan_c *on* AMD *platform during iterative compilation. Depending on optimization scenarios, good optimization cases are depicted with circles on 2D optimization area frontier and with squares where third metric is more than some threshold (compilation time speedup > 2 or code size improvement > 1.2).*

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's best optimization heuristic (-O3) is

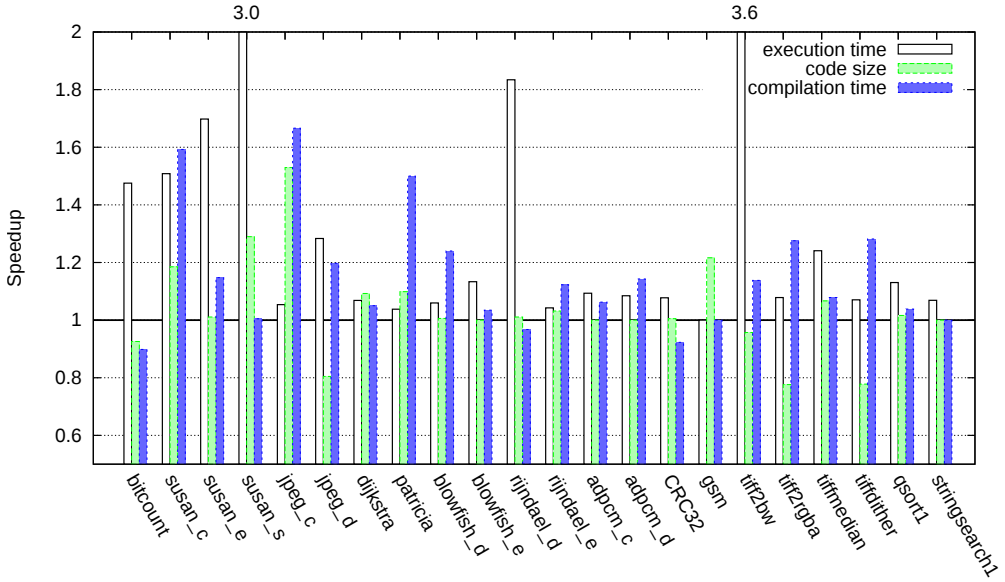


Figure 3.5 – *Code size improvements and compilation time speedups for optimization cases with execution time speedups within 95% of the maximum available on* Intel *platform (as found by iterative compilation).*

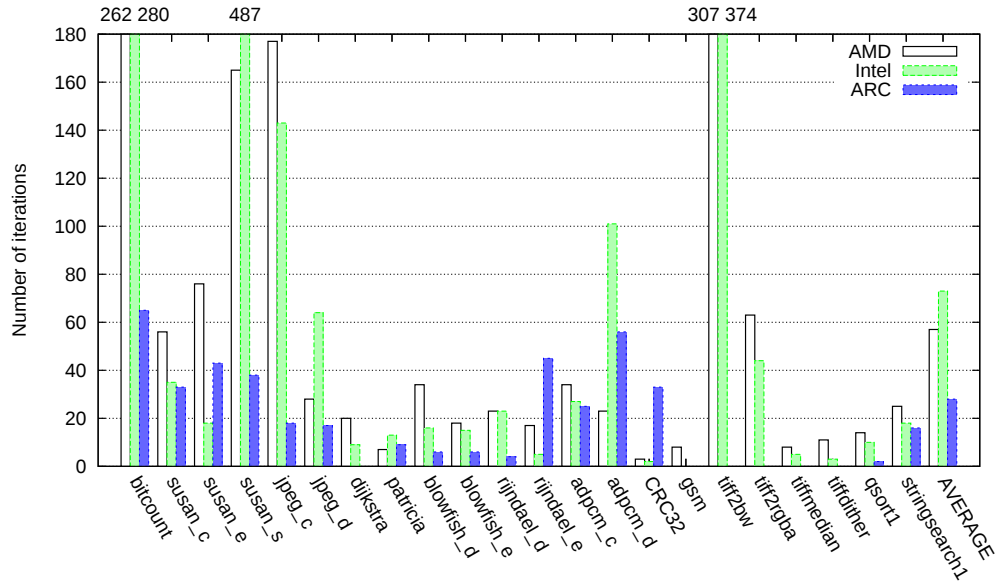


Figure 3.6 – *Number of iterations needed to obtain 95% of the available speedup using iterative compilation with uniform random distribution.*

far from optimal. All combinations of flags found per program and architecture during this research are available on-line in the Collective Optimization Database [33] to allow end-users to optimize their programs or enable further collaborative research.

Finally, Figure 3.6 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

-O1 -fcse-follow-jumps -fno-tree-ter -ftree-vectorize
-O1 -fno-cprop-registers -fno-move-loop-invariants -fno-tree-copy-prop -fno-dce -frename-registers -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-tree-copyrename -fno-inline-functions-called-once -fno-schedule-insns2 -fno-tree-ccp -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-vrp -fno-tree-loop-optimize -fno-tree-ter
-O2 -fno-caller-saves -fno-guess-branch-probability -fno-ira-share-spill-slots -fno-web -fno-tree-reassoc -funroll-all-loops
-O2 -fno-caller-saves -fno-reorder-blocks -fno-strict-overflow -funroll-all-loops -fno-ivopts
-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 -fno-cprop-registers -fno-if-conversion -fno-peephole2 -funroll-all-loops -falign-loops
-O3 -falign-loops -fno-delete-null-pointer-checks -fno-gcse-lm -fira-coalesce -fno-web -fsched2-use-superblocks -fno-tree-vectorize -funsafe-loop-optimizations -floop-interchange -fno-tree-pre -funroll-all-loops
-O3 -fno-gcse -floop-strip-mine -fno-move-loop-invariants -fno-predictive-commoning -ftracer
-O3 -fno-inline-functions-called-once -frename-registers -fno-tree-copyrename -fno-regmove
-O3 -fno-inline-functions -fno-move-loop-invariants

Table 3.1 – *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.*

compiler where multiple combinations of optimizations can both considerably increase or decrease performance, change code size and compilation time.

Our experimental results suggest that iterative compilation can effectively generalize and automate the program optimization process but can be too time consuming. The total execution time for the first 70 iterations for `telecom_adpcm_d` is 323m and for `network_dijkstra`, it is 16 mins. Hence, it is important to speed up iterative compilation process.

3.4 Summary

In this chapter, we empirically demonstrated that iterative compilation can effectively perform tuning of compiler optimizations, but the large number of evaluations required for each program makes it impractical with respect to execution time, compilation time and power consumption. Inspite of providing acceptable optimization results, the optimization space in iterative compilation is too large to be effectively explored in reasonable time. This motivates the use of machine learning techniques to mitigate the need for per-program iterative compilation and learn optimizations across programs based on their features. In the next chapter, we present the Milepost framework which speeds up program optimization through machine learning.

4

MILEPOST GCC: Speeding up iterative compilation with machine learning

4.1 Introduction

Iterative compilation can considerably outperform existing compilers but at the cost of excessive recompilation and program execution during optimization search space exploration as shown in the previous chapter. Multiple techniques have been proposed to speed up this process. For example, ACOVEA tool [1] utilizes genetic algorithms; hill-climbing search [51] and run-time function-level per-phase optimization evaluation [54] have been used, as well as the use of Pareto distribution [69, 74] 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.

In this chapter we demonstrate how machine learning can be effectively used for tuning compiler optimization heuristics. We describe Milepost GCC [55], our open-source machine learning-based compiler which consists of an Interactive Compilation Interface (ICI) and plugins to extract program features and exchange optimization data with a public repository (cTuning.org). 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.

The Milepost project takes an orthogonal approach based on the observation that similar programs may exhibit similar behavior and require similar optimizations so it is possible to correlate program features and optimizations, thereby predicting good transformations for unseen programs based on previous optimization experience [106, 16, 128, 2, 73, 17, 60]. 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 [17], microarchitecture-independent characteristics [73], reactions to transformations [60] or semantically non-equivalent program modifications [47]).

The contribution in the context of this thesis includes the maintainance of Milepost GCC, and its evaluation by a number of empirical experiments on GRID5000, GCC ICI extension, migration to a new GCC compiler, statistical analysis and flag pruning. 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. The experimental setup used in the evaluation of Milepost GCC platform is described in Section 3.2.

4.1.1 Milepost adaptive optimization framework

The Milepost framework shown in Figure 4.1 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 [33, 52, 60]. The Milepost framework proceeds in two distinct phases, in accordance with typical machine learning practice: *training* and *deployment*.

4.1.1.1 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 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 [18, 52], 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.

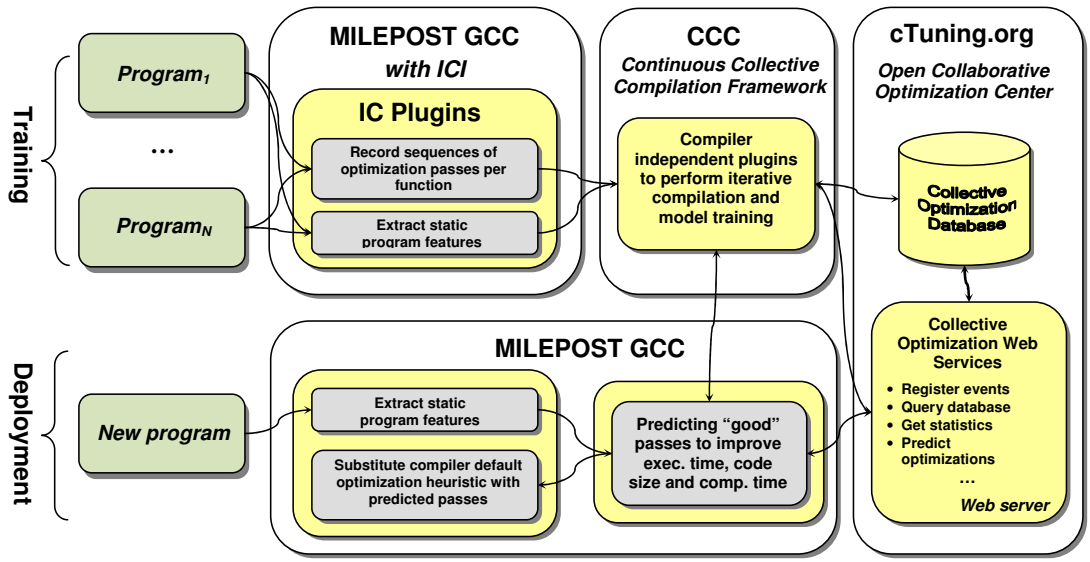


Figure 4.1 – *Open framework to automatically tune programs and improve default optimization heuristics using predictive machine learning techniques, Milepost GCC with Interactive Compilation Interface (ICI) and program features extractor, CCC Framework to train ML model and predict good optimization passes, and COD optimization repository at cTuning.org.*

4.1.1.2 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.

4.1.2 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) [75] 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 too many internal 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-oriented self-tuning compiler to provide a common, stable, and 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. The internal structure of ICI is shown in Figure 4.2.

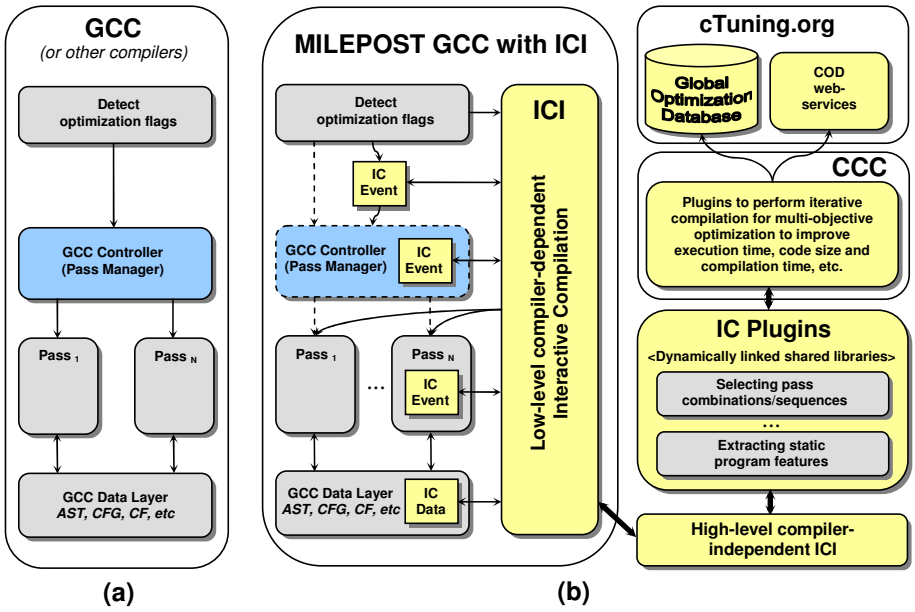


Figure 4.2 – *GCC Interactive Compilation Interface: a) original GCC, b) Milepost GCC with ICI and plugins*

We separate ICI into two parts: low-level compiler-dependent and high-level compiler independent. The main reason of this separation is 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. They even allow construction of different pass orderings for each and every function in the program being compiled. This mechanism simplifies the inclusion 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.

4.1.3 Static program features

Milepost GCC's 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. The entities are a direct mapping of similar entities defined by the language reference, or generated during compilation. Examples of such 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 [136, 135].

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 4.1) which later serves to detect similarities between programs, build machine learning models and predict the best combinations of passes for new programs. More details about aggregation of semantical program properties for machine learning based optimization are provided in [108].

4.2 Predicting optimization passes with machine learning

The Milepost approach to learning optimizations across programs is based on the observation that similar programs may exhibit similar behavior for a similar set of optimizations [2, 60], 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 4.3. 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 the construction of a *probabilistic model* and 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 be improved more by using the transductive method depending on the size of the training set, the number of samples of the program space, as well as program and architecture attributes.

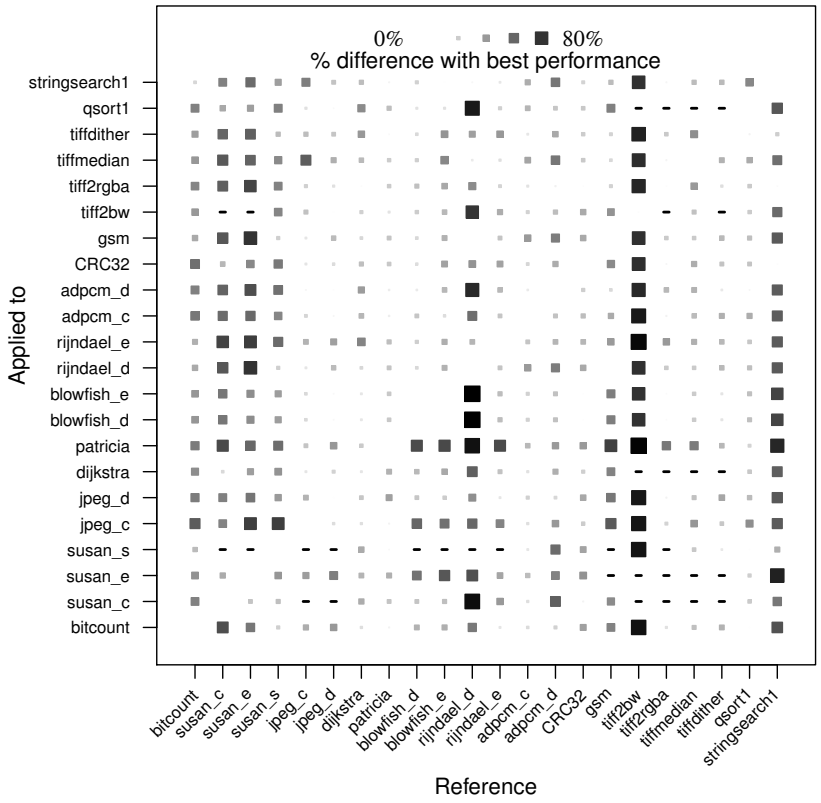


Figure 4.3 – *% difference between speedup achievable after iterative compilation for "*applied to*" program and speedup obtained when applying best optimization from "*reference*" program to "*applied to*" program on* AMD. *"-" means that best optimization was not found for this program.*

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 3.3. 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

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 4.1 – *List of static program features currently available in Milepost GCC V2.1*

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.

4.2.1 Probabilistic machine learning model

Our probabilistic machine learning method is similar to that of [2] 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) [12]. However, unlike prior work [2, 12] 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.

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:

$$\mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmax}}\, q(\mathbf{x}|\mathbf{t},\boldsymbol{\theta}). \tag{4.1}$$

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 [91] 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 Independent and Identically Distributed (IID) model where each of the elements of the combination are considered independently. In IID model, all the good solutions have the same probability distribution as the others and all are mutually independent.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. This is a reasonable and realistic model to provide simplicity.

$$P(\mathbf{x}|T^j) = \prod_{\ell=1}^{L} P(x_\ell|T^j), \tag{4.2}$$

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 4.4 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.

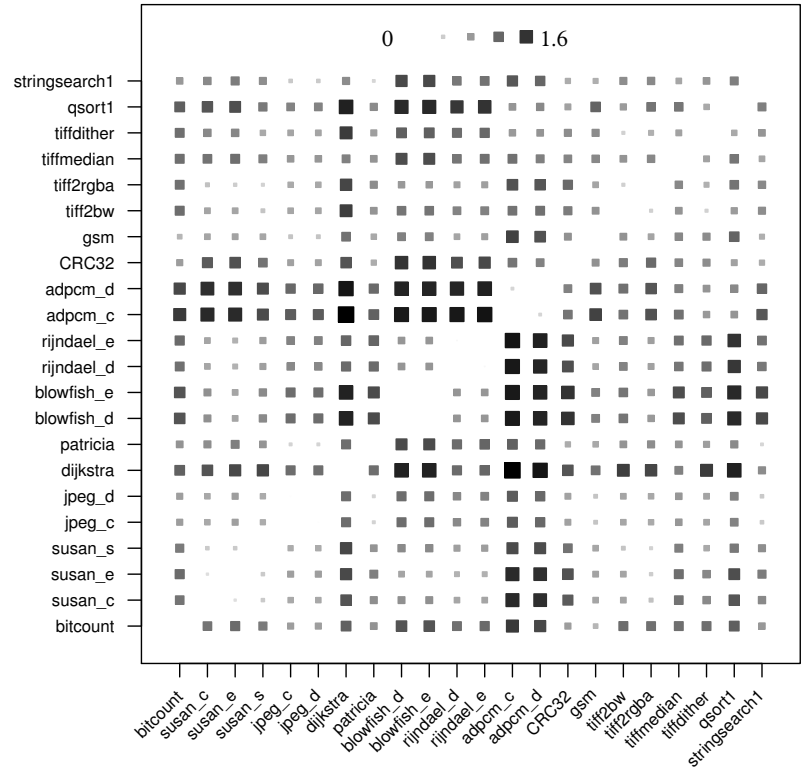


Figure 4.4 – *Euclidean distance for all programs based on static program features normalized by feature 24 (number of instructions in a method).*

Figure 4.5 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 the best combination of optimizations from the closest program. Interestingly, the results suggest that simply selecting the best combination of optimizations from a similar program may not perform

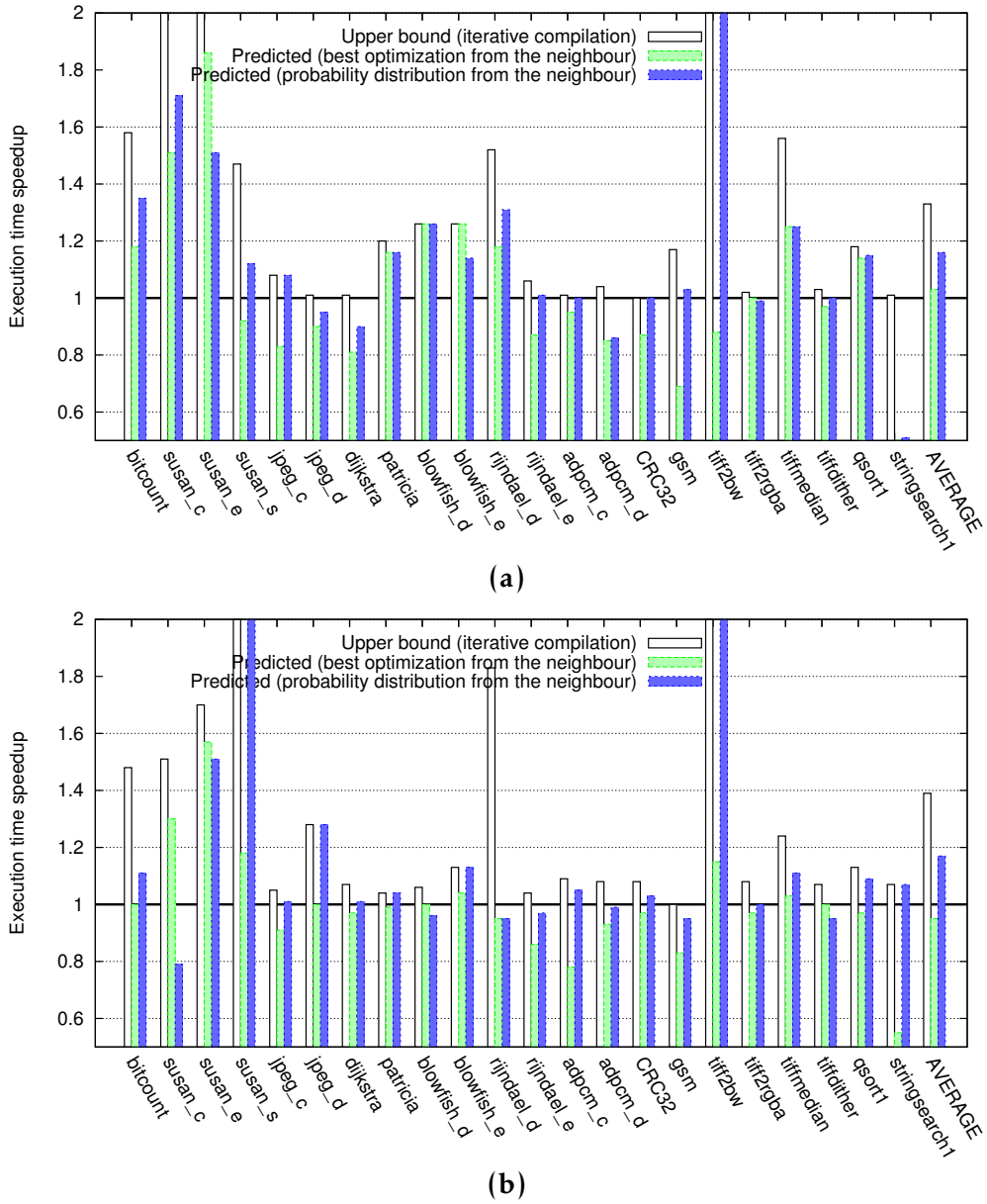


Figure 4.5 – *Speedups achieved when using iterative compilation on (a) AMD and (b) Intel with random search strategy (1000 iterations; 50% probability to select each optimization;), when selecting best optimization from the nearest program and when predicting optimization using probabilistic ML model based on program features.*

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.

4.2.2 Transductive machine learning model

We describe a new transductive approach where optimization combinations themselves are used, as features for the learning algorithm, together with 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 [40] 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 consists 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 [42] 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.

Figure 4.6 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 4.7 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 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.

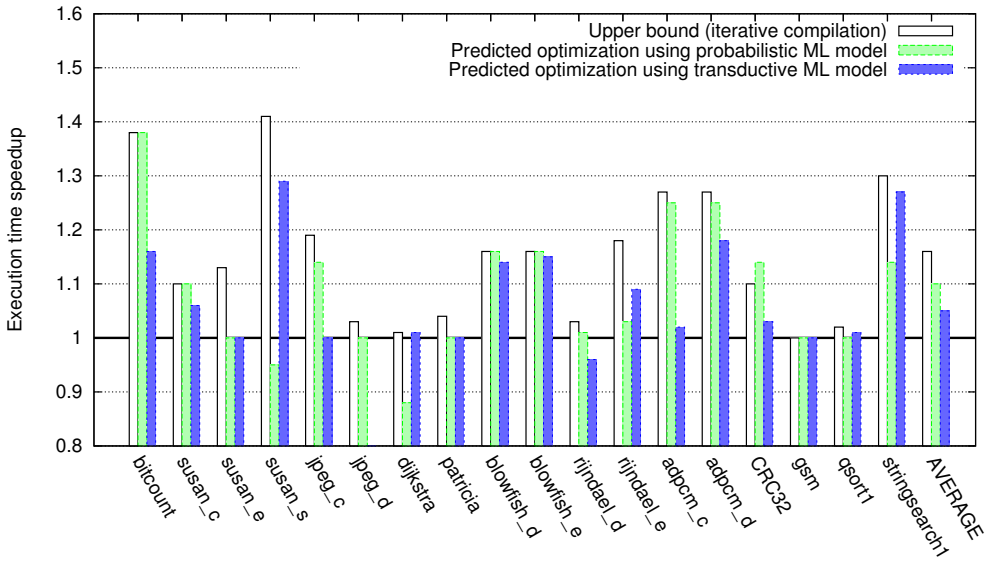


Figure 4.6 – *Speedups achieved when using iterative compilation on ARC with random search strategy (1000 iterations; 50% probability to select each optimization;) and when predicting best optimizations using probabilistic ML model and transductive ML model based on program features*

4.3 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 4.2.1. 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 3.3, in order to find the upper bounds for execution time, code size and compilation time.

For simplicity, we decided to use a probabilistic machine learning model from Section 4.2.1. 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 nearest neighbours. 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

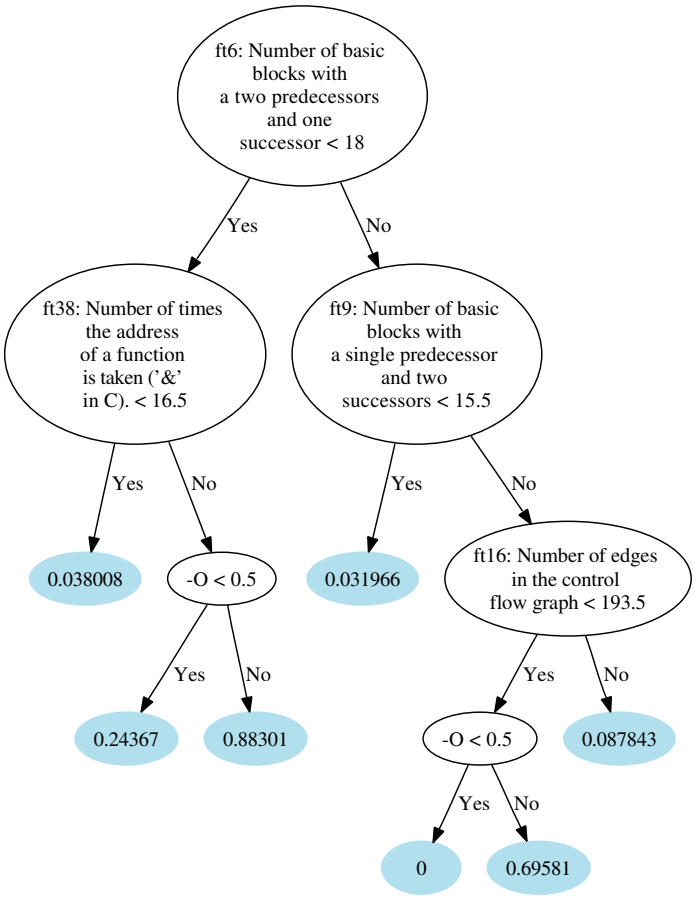


Figure 4.7 – *Top levels of decision trees learnt for ARC.*

time and (c) improve compilation time while not degrading execution time. Figure 4.8 shows the achieved execution time speedups, code size improvements and compilation time speedups over -O3 optimization level when applying selected optimizations 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 [33] to help users and researchers validate and reproduce such results. These optimization cases are referenced by the following cTuning RUN_ID reference numbers: 24857532370695782,

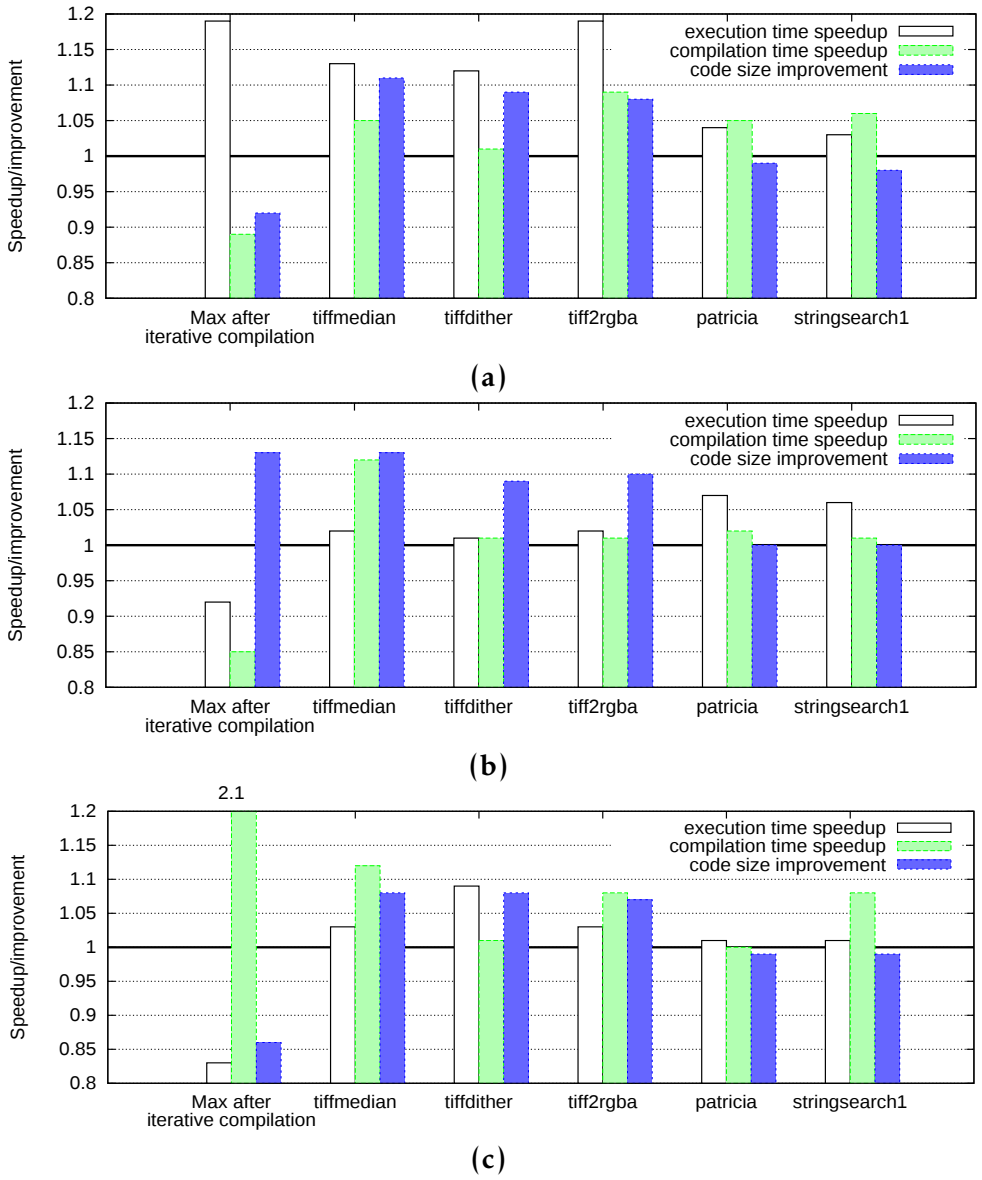


Figure 4.8 – *Execution time speedups (a), code size improvements (b) and compilation time speedup (c) for BerkeleyDB on Intel when applying optimizations from 5 closest programs from MiBench/cBench (based on Euclidean distance using static program features of 3 hottest functions) using several optimization scenarios.*

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.

4.4 Summary

In this chapter, we showed that our machine learning based compiler, 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. Having said that, machine learning requires a huge dataset of good solutions with diverse features, i.e., having a few best optimization solutions across many benchmarks to map features to correct optimizations and program characterization. Since all the possible observations and scenarios can not be foreseen and presented during training, machine learning based auto-tuning does not deliver desired prediction accuracy for all optimization cases. Moreover, the training is too long which includes data collection and data cleansing and there are no representative benchmarks, datasets or shared models to be improved by the community. In order to enhance machine learning dataset and include new observations, we need a collaborative effort to collect the experimental data from the community and crowdsource the relevant optimizations in return. In the next chapter, we describe our collaborative framework for compiler optimization, the Collective Mind, which is based on our machine learning based compiler Milepost GCC. Collective Mind framework further enhances the capabilities of our compiler and addresses the aforementioned issues by incorporating new information into its repository, learning new observations, disseminating the best optimizations to the community and allowing the community to improve the shared modules, benchmarks and datasets.

5

Crowdsourcing compiler auto-tuning practical with Collective Mind

5.1 Introduction

In this chapter, we describe the framework of our practical, collaborative and publicly available solution to cope with the problems discussed in Chapter 1 using a collaborative knowledge management system called Collective Mind (cM) [98, 58]. The cM comprises a repository and infrastructure with unified web interfaces and online advise system. This collaborative framework preserves and shares many artifacts including hundreds of codelets, numerical applications, data sets, models, universal experimental analysis and auto-tuning pipelines, self-tuning machine learning based meta compiler, and unified statistical analysis and machine learning plugins in a public repository to initiate systematic, reproducible and collaborative research and development in which experiments and techniques are validated, ranked and improved by the community.

The chapter is organized as follows. We start with the formalization of the eventual needs of end-users and system developers or providers. Afterwards, we describe the Collective Mind infrastructure and repository in detail.

5.2 Collective Mind approach

End-users generally need to perform some tasks (playing games on a console, watching videos on mobile or tablet, surfing Web, modeling a new critical vaccine on a supercomputer or predicting a new crash of financial markets using cloud services) either as fast as possible or with some real-time constraints while minimizing or amortizing all associated costs including power consumption, soft and hard errors, and device or

service price. Therefore, end-users or adaptive software require a function that can suggest most optimal design or optimization choices **c** based on properties of their tasks and data sets **p**, set of requirements **r**, as well as current state of the computing system **s** under consideration:

$$\mathbf{c} = F(\mathbf{p}, \mathbf{r}, \mathbf{s})$$

This function is associated with another function representing behavior of a user task running on a given system depending on properties and choices:

$$\mathbf{b} = B(\mathbf{p}, \mathbf{c}, \mathbf{s})$$

This function is of particular importance for hardware and software designers that need to continuously provide and improve choices (solutions) for a broad range of user tasks, data sets and requirements while trying to improve own return on investment (ROI) and reduce time to market. In order to find optimal choices, it should be minimized in presence of possible end-user requirements (constraints). However, the fundamental problem is that nowadays this function is highly non-linear with such a multi-dimensional discrete and continuous parameter space which is not anymore possible to model analytically or evaluate empirically using exhaustive search [137, 51]. For example, **b** is a behavior vector that can now include execution time, power consumption, compilation time, code size, device cost, and any other important characteristic; **p** is a vector of properties of a task and a system that can include static program features [106, 129, 2, 55], data set properties, hardware counters [17, 81], system configuration, and run-time environment parameters among many others; **c** represents available design and optimization choices including algorithm selection, compiler and its optimizations, number of threads, scheduling, processor ISA, cache sizes, memory and interconnect bandwidth, frequency, etc; and finally **s** represents the state of the system during parallel execution of other programs, system or core frequency, cache contentions and so on.

5.2.1 Interdisciplinary collaborative methodology

Current multiple research projects mainly show that it is possible to use some off-the-shelf on-line or off-line adaptive exploration (sampling) algorithms combined with some existing models to approximate above function and predict behavior, design and optimization choices for 70-90% cases but in a very limited experimental setup. In contrast, our ambitious long-term goal is to understand how to continuously build, enhance, systematize and optimize hybrid models that can *explain and predict all possible behaviors and choices* while selecting minimal set of representative properties, benchmarks and data sets for predictive modeling [96]. We reuse our interdisciplinary knowledge in physics, quantum electronics and machine learning to build a new methodology that

can effectively deal with rising complexity of computer systems through gradual and continuous top-down problem decomposition, analysis and learning. We also develop a modular infrastructure and repository that allows to easily interconnect various available tools and techniques to distribute adaptive probabilistic exploration, analysis and optimization of computer systems among many users [125, 22] while exposing unexpected or unexplained behavior to the community with interdisciplinary backgrounds particularly in machine learning and data mining through unified web interfaces for collaborative solving and systematization.

5.3 Collective Mind infrastructure and repository

Eventually, we started searching for a possible solution that could liberate software developers from the tedious and not necessarily relevant job of continuous optimization and accounting while gradually making existing software performance- and cost-aware. At first, we tried to create a simple database of optimizations and connect it to some existing benchmarking and auto-tuning tools to keep track of all optimizations [52, 55]. However, when trying to implement it within production environments of our industrial partners, we faced several severe problems including difficulty to expose all design and optimization choices from continuously evolving software, and difficulty to reproduce performance numbers collected from different machines. This eventually pushed us to develop a full-fledged repository of knowledge with unified web services (Collective Mind or cM for short) similar to ones that helped successfully systematize research and experimentation in biology, genomics and other natural sciences. Such repository should be able to keep the whole auto-tuning setups with all dependencies including optimized software, data sets and auto-tuning tools. This, in turn, should allow us to distribute the whole auto-tuning setups among many users to crowdsource software optimization (or any other experimentation) in a reproducible way while considerably reducing usage costs. Briefly [1], cM helps to decompose software into standalone pieces interconnected through cM wrappers. Such light-weight wrappers currently support major languages including C, C++, Fortran, Python, PHP and Java, and allow the community to gradually expose various design and optimization choices **c**, features **f**, dependencies on other software and hardware, monitored characteristics (costs) **b** and environment state **s** in a unified way through extensible JSON format [83]. Figure 5.1 depicts the high-level view of Collective Mind framework and repository.

The software pieces can be extracted and then shared together with their wrappers and data set samples in the Hadoop-enabled [126] cM repository. For example, with the help of our colleagues and supporters, we already gradually and semi-automatically extracted and shared 285 software pieces (codelets) together with several thousand

[1] Though we provide minimal information about Collective Mind framework in this chapter, it should be enough to understand proposed concepts. However, in case of further interest, more details can be found in [59, 100]

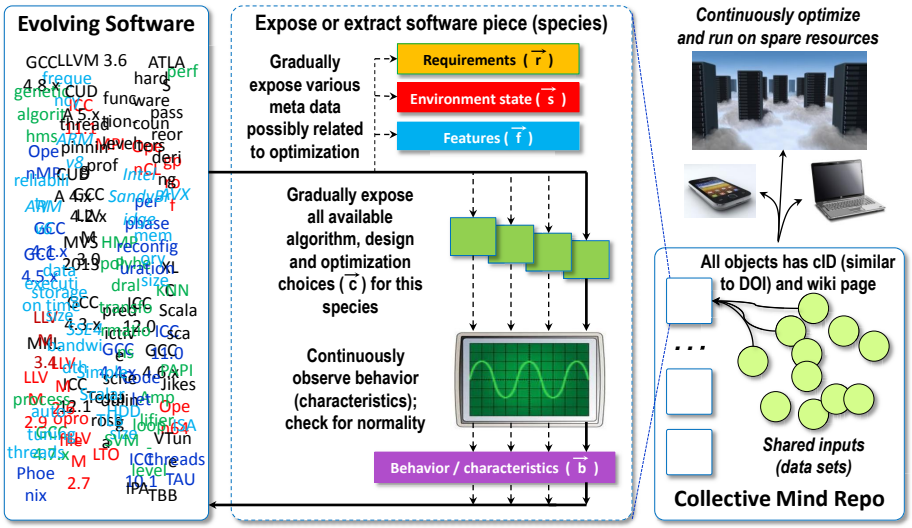


Figure 5.1 – *Collective Mind Framework and Repository (cM) help to decompose any complex software into pieces with light-weight wrappers that expose design and optimization choices, measured characteristics, features and environment state in a unified and mathematical way(using vectors). It was developed to unify and systematize software autotuning, make it practical and reproducible, and distribute it among numerous computing resources such as mobile phones and data centers shared by volunteers [59, 100].*

data set pairs from several real software projects as well as 8 popular benchmark suits including NAS, MiBench, SPEC2000, SPEC2006, Powerstone, UTDSP and SNU-RT. Recently Pablo et al. [35] proposed an open source framework Codelet Extractor and REplayer(CERE). CERE finds and extracts the hotspots of an application as codelets. This can liberate software engineers from developing their own ad-hoc and complex tuning setups in favor of implementing common auto-tuning pipelines consisting of shared software pieces, data sets, tools and optimization space exploration modules. Such pipelines can then be easily shared and distributed across a large number of diverse computer systems either using open source *cM buildbot* or a small *cM node* that can deploy experiments on Android-based devices [101]. cM will then continuously "crawl" for better optimizations for all shared software pieces, data sets and compilers, while recording experiments in a reproducible way in the public cM repository at c-mind.org/repo.

At a coarse-grain level, modules serve as wrappers around existing command line tools such as compilers, source-to-source transformers, code launchers, profilers, among many others. Such modules are written in python for portability and productivity reasons, and can be launched from command line in a unified way using Collective Mind front-end *cm* as following:

cm ⟨ *module name or UID* ⟩ ⟨ *command* ⟩ ⟨ *unified meta information* ⟩ – ⟨ *original cmd* ⟩

These modules enable transparent monitoring of information flow, exposure of vari-

ous characteristics and properties in a unified way (meta information), and exploration or prediction of design and optimization choices, while helping researchers to abstract their experimental setups from constant changes in the system. Internally, modules can call each other using just one unified *cM access function* which uses a schema-free easily extensible nested dictionary that can be directly serialized to JSON as both input and output as following:

```
r = cm_kernel.access({'cm_run_module_uoa':<module name or UID>,
                      'cm_action':<command>,
                      parameters})
```

where command in each module is directly associated with some function. Since JSON can also be easily transmitted through Web using standard http post mechanisms, we implemented a simple cM web server that can be used for P2P communication or centralized repository during crowdsourcing and possibly multi-agent based on-line learning and tuning.

Each module has an associated storage that can preserve any collections of files (whole benchmark, data set, tool, trace, model, etc) and their meta-description in a JSON file. Thus, each module can also be used for any data abstraction and includes various common commands standard to any repository such as *load*, *save*, *list*, *search*, *etc*. We use our own simple directory-based format as following:

```
.cmr/<Module name or UID>/<Data entry UID>
```

where .cmr is an acronym for Collective Mind Repository. In contrast to using SQL-based database in the first cTuning version that was fast but very complex for data sharing or extensions of structure and relations, a new open format allows users to be database and technology-independent with the possibility to add, update, delete and share entries or repositories in whole using standard OS functions and tools like SVN, GIT or Mercury, or easily convert them to any other format or database if necessary. Furthermore, cM can transparently use open source JSON-based indexing tools such as ElasticSearch [126] to enable fast and powerful queries over schema-free meta information. Now, any research artifact will not be lost and can now be referenced and directly found using the so called cID (Collective ID) of the format: ⟨ *module name or UID* ⟩:⟨ *data entry or UID* ⟩.

Such infrastructure allows researchers and engineers to connect existing or new modules into experimental pipelines like "research LEGO" with exposed characteristics, properties, constraints and states to quickly and collaboratively prototype and crowdsource their ideas or production scenarios such as traditional adaptive exploration of large experimental spaces, multi-objective program and architecture optimization or continuous on-line learning and run-time adaptation while easily utilizing all available benchmarks, data sets, tools and models provided by the community. Additionally, single and unified access function enables transparent reproducibility and validation of

any experiment by preserving input and output dictionaries for a given experimental pipeline module. Furthermore, we decided to keep all modules inside repository thus substituting various ad-hoc scripts and tools. With an additional cM feature to install various packages and their dependencies automatically (compilers, libraries, profilers, etc) from the repository or keep all produced binaries in the repository, researchers now have an opportunity to preserve and share the whole experimental setup in a private or public repository possibly with a publication.

We started collaborative and gradual decomposition of large, coarse-grain components into simpler sub-modules including decomposition of programs into kernels or codelets [141] to keep complexity under control and possibly use multi-agent based or brain inspired modeling and adaptation of the behavior of the whole computer system locally or during P2P crowdsourcing. Such decomposition also allows community to first learn and optimize coarse-grain behavior, and later add more fine-grain effects depending on user requirements, time constraints and expected return on investment (ROI) similar to existing analysis methodologies in physics, electronics or finances.

5.3.1 Data and parameter description and classification

In traditional software engineering, all software components and their APIs are usually defined at the beginning of the project to avoid modifications later. However, in our case, due to ever evolving tools, APIs and data formats, we decided to use agile methodology together with type-free inputs and outputs for all functions focusing on quick and simple prototyping of research ideas. Only when modules and their inputs and outputs become mature or validated, then (meta)data and interfaces are defined, systematized and classified. However, they can still be extended and reclassified at any time later. For example, any key in an input or output dictionary of a given function and a given module can be described as “choice”, “(statistical) characteristic”, “property” and “state”, besides a few internal types including “module UID”, “data UID” or “class UID” to provide direct or semantic class-based connections between data and modules. Parameters can be discrete or continuous with a given range to enable automatic exploration. Thus, we can easily describe compiler optimizations; data set properties such as image or matrix size, architecture properties such as cache size or frequency, represent execution time, power consumption, code size, hardware counters; categorize benchmarks and codelets in terms of reaction to optimizations or as CPU or memory bound, and so on.

Our proposed framework provides a practical and evolutionary approach based on the aforementioned formalization of objectives of various research projects where the community gradually provides simple wrappers for the tools used including compilers, source-to-source transformers, code launchers, profilers to transparently monitor all information flow in experimental setups as shown in Figure 5.2b.

At the same time, researchers gradually expose various characteristics of behav-

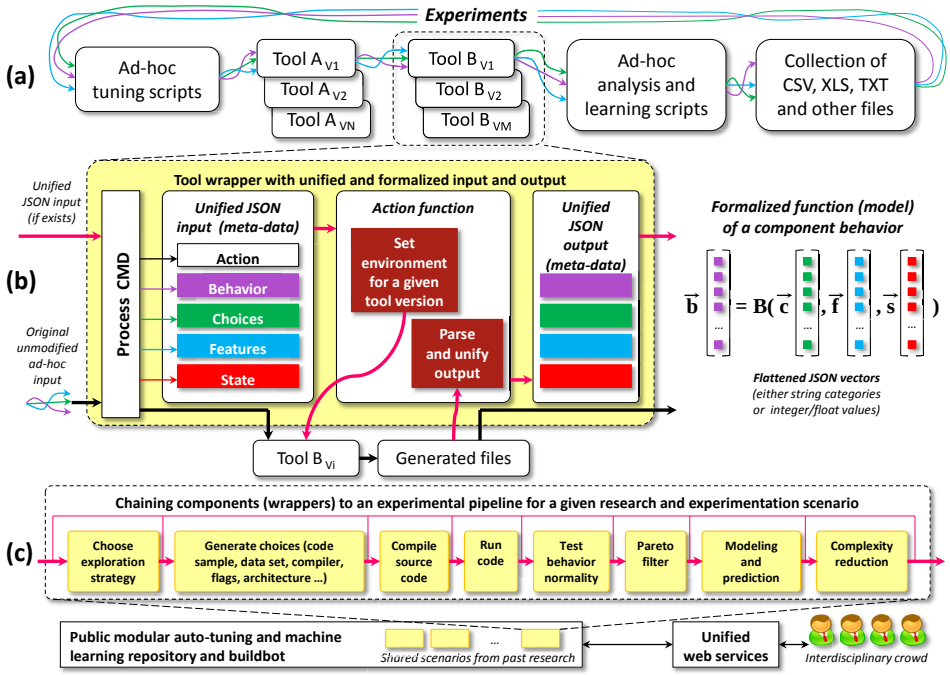


Figure 5.2 – *(a) Conceptually depicted current ad-hoc experimentation; (b) wrappers developed by the community around existing tools to gradually expose behavior (characteristics), choices, features and system state using unified JSON input and output format; (c) wrappers and modules chained together as LEGO to implement various experimentation scenarios within a public buildbot that can be collaboratively explored and improved by the community.*

ior **b**, choices **c**, system state **s** and features **f** (meta information) from this flow *only when needed to implement a given research scenario* using popular and human readable, language-independent and easily extensible JSON data format [83] based on combinations of string keys, values, lists and dictionaries as in the following example:

```
{"characteristics":{
    "execution_times": ["10.3","10.1","13.3"],
    "code_size": "131938", ...},
  "choices":{
    "os":"linux", "os_version":"2.6.32-5-amd64",
    "compiler":"gcc", "compiler_version":"4.6.3",
    "compiler_flags":"-O3 -fno-if-conversion",
    "platform":{"
      "processor":"intel_xeon_e5520", "l2":"8192",
      "memory":"24" ...}, ...},
  "features":{
    "semantic_features": {"number_of_bb": "24", ...},
    "hardware_counters": {"cpi": "1.4" ...}, ... }
  "state":{
    "frequency":"2.27", ...}
}
```

From past experience in building community-based frameworks, we noticed that researchers are not always good programmers and naturally care more about quick prototyping of their research ideas rather than drowning in complex specifications for experiments that may be even thrown away in the end. Therefore, in contrast to other frameworks, we decided to *get rid of pre-defined data specifications and rigid SQL-based databases which are difficult or even impossible to extend in rapidly evolving projects* in favor of agile methodology [4] which is gaining more popularity recently and noSQL databases to let community derive the most simple, appropriate and backward compatible specification just enough for their needs and only when research scenario and modules are validated and can be shared with a wide community. JSON perfectly fits such approach and is now backed up by many companies, supported by most of the recent languages, web technologies and schema-free repositories [126], and can be easily used for web services and P2P communication during experimentation.

Therefore, each wrapper has an associated file to describe the information flow (input and output) using our own *flat JSON format* to be able to reference any key in the complex JSON hierarchy using just one string. Such flattened key always starts with # followed by *#key* if it is a dictionary key or *@position_in_a_list* if it is a value in a list. For example, flattened key for the second execution time "10.1" in the above dictionary example is *"##characteristics#execution_time@1"*. By now, we prepared the following description of the information flow enough to validate many existing auto-tuning and machine learning techniques.

```
"flattened_json_key":{
  "type": "text" | "dict" | "list" | "integer" | "float" | "category" | "uid",
  "characteristic": "yes" | "no",
  "feature": "yes" | "no",
  "state": "yes" | "no",
  "has_choice": "yes" | "no",
  "choices": ["list of strings if categorical choice"],
  "explore_start": "start number if numerical range",
  "explore_stop": "stop number if numerical range",
  "explore_step": "step if numerical range",
  "can_be_omitted": "yes" | "no",
}
```

This specification is currently under constant extension. Finally, we introduce modules that perform mathematical and other actions on unified JSON inputs and outputs (similar to filters in electronics) or simply chain wrappers and other modules into experimental pipelines within a public buildbot to quickly prototype research ideas using existing components or gradually convert existing ad-hoc experimental setups to a unified format as shown in Figure 5.2c. Wrappers and modules are written in Python

for productivity and portability reasons (though technically any language can be used), and can easily call each other using one unified API function with input and output JSON, thus substituting and unifying all ad-hoc experimentation scripts, or can be invoked from the command line by just prefixing original tool with a buildbot front-end as following:

buildbot_fe ⟨*wrapper/module name or UID*⟩ ⟨*action_function*⟩ @*unified_input.json* –– ⟨*original CMD*⟩

Each wrapper or module has an assigned unique ID and an associated directory storage of format *.repository/*⟨*wrapper/module name or UID*⟩*/*⟨*data entry UID*⟩ to preserve any related research artifact with an associated meta-description such as features or classification in a JSON file thus effectively abstracting data access. For example, module *source.code* can preserve all code samples, module *dataset* will keep all data sets, wrapper *compiler* will keep description of various compilers and their tuning parameters, module *model* will keep various shared predictive models with different parameters, module *experiment.result* will keep auto-tuning results and so on. Meta description is transparently indexed using open-source JSON-based ElasticSearch framework [126] allowing fast and complex search queries.

5.3.2 OpenME interface for fine-grain analysis, tuning and adaptation

Most of the current compilers, applications and run-time systems are not prepared for easy and straightforward fine-grain analysis and tuning due to associated software engineering complexity, sometimes proprietary internals, possible compile or run-time overheads, and still occasional disbeliefs in effective run-time adaptation. Some extremes included either fixing, hardwiring and hiding all optimization heuristics from end-users or oppositely exposing all possible optimizations, scheduling parameters, hardware counters, etc. Some other available mechanisms to control fine-grain compiler optimization through pragmas can also be very misleading since it is not always easy or possible to validate whether optimization was actually performed or not. Instead of developing yet more source-to-source tools or binary translators and analyzers, we developed a simple event-based plugin framework called Interactive Compilation Interface (ICI) to "open up" previously hardwired tools for external analysis and tuning. ICI was written in plain C originally for Open64 and later for GCC, requires minimal instrumentation of a compiler and helps to expose or modify only a subset of program properties or compiler optimization decisions through external dynamic plugins based on researcher needs and usage scenario. This interface can easily evolve with the compiler itself, has been successfully used in the MILEPOST project to build machine-learning self-tuning compiler [55], and is now available in mainline GCC. Based on this experience, we developed a new version of this interface (OpenME) [125] that is used to "open up" any available tool such as GCC, LLVM, Open64, architecture simulator, etc., in a unified way as shown in Figure 5.3a, or any application, for example, to train predictive scheduler on heterogeneous many-core architectures [81] as shown in

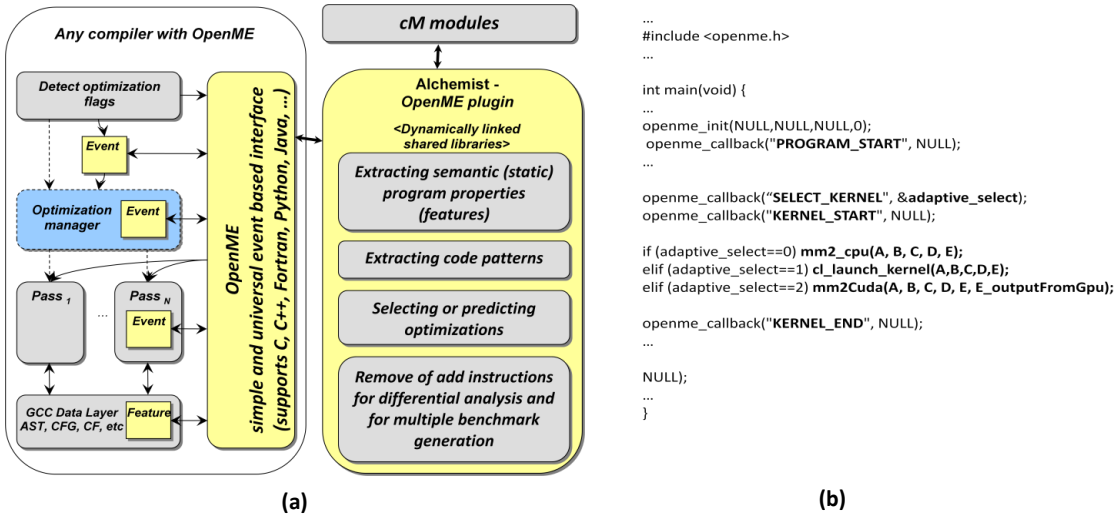


Figure 5.3 – *Event and plugin-based OpenME interface to "open up" rigid tools (a) and applications (b) for external fine-grain analysis, tuning and adaptation, and connect them to cM*

Figure 5.3b.

5.4 Co-existence of multiple versions of tools and libraries

Yet another challenge that makes experimentation and life of computer researchers and engineers very exciting is continuously changing tools and libraries. Presented approach with tool wrappers and an artifact repository helps to elegantly solve this problem. We naturally consider packages and libraries as research artifacts (or choices) too and therefore moved them to a repository with an associated unified module to be able to install any given package on a given user machine on demand while automatically resolving all dependencies. A special OS-dependent script is always created during installation to set up binary, includes and library paths and all other necessary environment variables inside a wrapper just before tool execution. We already prepared packages and installation scripts compatible with our buildbot for most of the versions of popular compilers, tools and libraries, including GCC, LLVM, ICC, Open64/PathScale compilers, PGI compilers, ROSE infrastructure, Oracle JDK, VTune, visual studio compilers, NVidia GPU toolkit, perf, gprof, GMP, MPFR, MPC, PPL, LAPACK and others to relieve community from this burden. Interestingly, we can use the same repository as an installation target thus providing an opportunity to researchers to preserve and share their whole experimental setups in private or public repositories possibly with a publication while referencing any research artifact directly using the format similar to DOI: ⟨*wrapper/module name or UID*⟩:⟨*data entry UID*⟩.

5.5 Summary

This chapter provided an in-depth overview of our collaborative framework and repository, the Collective Mind, for compiler auto-tuning and showed how we can collaboratively share benchmarks, datasets and models with the community. The next chapter explains how do we cope with the unexpected behavior and unforeseen scenarios by collaboratively discovering missing features, thus improving model prediction accuracy.

6

Crowdsourcing feature learning and model improvement

6.1 Introduction

The cM framework is subjected to continuous evolution and improvement. This necessitates discovering missing features for enhancing its repository and improving model prediction accuracy. This is only possible when we offer the community to share their experimental results, benchmarks and code for continuously incorporating new observations into the knowledge-base for better model prediction.

In this chapter, we formalize the current research on auto-tuning and machine learning allowing to implement various research scenarios as shared experimental pipelines. For the proof of the concept, we describe two experimental scenarios to validate compiler auto-tuning and machine learning combined with continuous and incremental complexity reduction. We also present a case study demonstrating our methodology in practice to expose missing features, improve compiler optimizations and make a real image processing application adaptive at run-time.

6.2 Public research scenarios and experimental pipelines

Optimization formalization allows researchers to implement most of the current auto-tuning techniques as a mathematical problem in terms of multiple characteristics (behavior), choices and features while easily reusing and chaining together well-known interdisciplinary techniques as buildbot plugins including normality test to analyze variation of experimental results and detect behavior anomalies [43], Pareto frontier filter to leave only optimal solutions during multi-objective optimization [90, 74] and

complexity reduction and differential analysis techniques [121, 82] to continuously isolate behavior anomalies, compact experimental data on the fly, leave only influential optimization dimensions (choices), related features and most accurate models. Furthermore, common optimization framework and cooperative methodology allows community to share multiple code and sample data sets and collaboratively explore large optimization spaces using our public buildbot while making use of machine learning statistically meaningful as conceptually summarized in Figure 6.1.

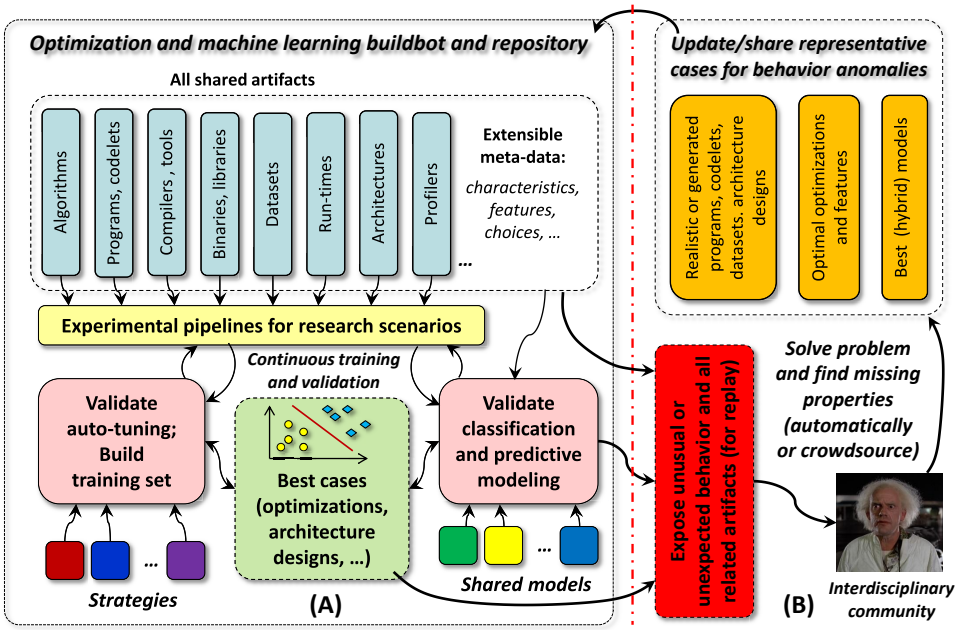


Figure 6.1 – *Summary of the presented cooperative approach and practical buildbot to collaboratively and semi-automatically learn and improve behavior of computer systems using complete public experimental pipelines including code and dataset samples, tools, models, features and all other associated artifacts shared, analyzed and improved by the interdisciplinary community.*

However, our approach also requires radical change in mentality of researchers when defining experiments that can be collaboratively explored through spare computational resources including mobile phones or cloud services. Rather than focusing on a few positive speedups from auto-tuning or prediction from machine learning that are relatively straightforward and can now be continuously shared in the public repository to directly improve end-user's applications, compilers, and run-time systems, researchers will need to prepare such experimental pipelines that can *continuously "crawl" for unusual or unexpected behavior of computer systems and models when spare resources become available*:

```
while (true)
    lsr = get_list_of_available_spare_resources()
    if len(lsr) > 0:
        sr=random(lsr)
        lep=get_list_of_shared_experimental_pipelines(get_features(sr))
        if len(lep) > 0:
            ep=run_pipeline(sr, random(lep), timeout(lsr))
            save_and_prune_expected_results(ep,sr)
            expose_unusual_behavior(ep,sr)
```

If a researcher has difficulties explaining results, mathematical formalization of a problem also allows exposing it to an interdisciplinary community that can help analyze and understand domain-specific problems (anomalies) while manually finding related features in the whole software and hardware stack to improve predictions which is currently practically impossible to generalize and automate until deep learning becomes practical and powerful enough [70, 92]. In the next sections, we will demonstrate how to use our approach to validate several well-known and far from being solved problems including automatic compiler flag tuning and prediction. Based on our practical experience and feedback from our industrial partners, it now takes just a few days rather than months to implement such scenarios as Python-based buildbot modules and wrappers (plugins), thus considerably increasing productivity and return on investment when prototyping research ideas.

6.2.1 Validating compiler auto-tuning (iterative compilation)

As the first practical usage of the presented approach and framework, our industrial partners desperately required practical compiler flag auto-tuning that has been well-known for decades, far from being solved and is getting tougher with years. However, in contrast to existing ad-hoc setups, we can now design an experimental pipeline as such to automatically and recursively query its all connected tool wrappers for available choices and monitor characteristics in a provided computer resource such as code and data set samples, compilers and their optimizations, execution time, power consumption, and hardware counters' profilers, and so on. These choices and behavior characteristics are aggregated in a JSON dictionary as **json_c** and **json_b** respectively. Such dictionaries can quickly become complex, for example, to accommodate other tuning techniques particularly on function, loop and instruction levels. Therefore, we use our flat JSON format introduced in Section 5.3.1, to flatten above dictionaries into vectors **c** and **b** together with their descriptions **c_desc** and **b_desc** that are automatically obtained from all associated tool wrappers.

The first relatively straightforward usage scenario allows end-users to *crowdsource program optimization*. In such scenario, a user just needs to provide some basic meta information about compilation and execution command lines for a given program, and use our buildbot web front-end or command line to mark characteristics to monitor and choices to explore including compilers, data sets, flags, or anything else available in the system, select preferable shared search strategy plugin that can be random, probabilistic, genetic, among many others, and chain available filters to process empirical data on the fly if needed. Importantly, unification of experimental results in a vector form simplifies and enables usage of multiple publicly available visualization, data mining and analytics web services for example from Google or available in various packages for Python, R, Weka, MATLAB, SciLab, and other popular tools. As example, we ran experimental pipeline to continuously optimize real image corner detection program

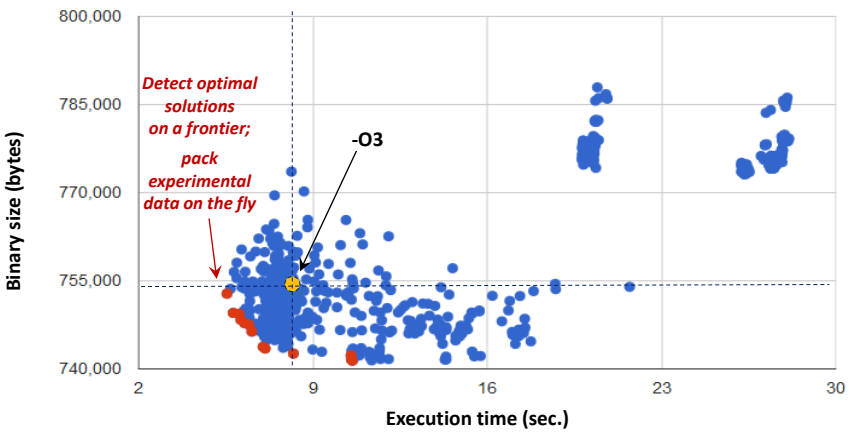


Figure 6.2 – *Variation in execution time vs code size when crowdsourcing optimization of an image corner detection application with a fixed dataset on Samsung Galaxy Series mobile phone with ARMv6 830MHz processor when randomly selecting compiler flags for Sourcery GCC 4.7.2. Yellow point represents -O3 and red circles show Pareto frontier.* This data will be available for validation at the conference.

using our colleagues' Android-based mobiles (mainly Samsung Galaxy Series), Sourcery GCC v4.7.2 with randomly generated combination of compiler flags of format *-O3 -f(no-)optimization_flag –parameter param=random_number_from_range*, and chained Paretto frontier filter for three characteristics (execution time, code size and compilation time) required by our partners. Figure 6.2 shows 2D visualization of the multi-dimensional optimization and characteristic space using Google Web Services. Before exploring multiple optimization choices on an available resource, note that we validate existing results using default choice configuration vector **c_def** such as *-O3* for compilers (shown by a yellow point on a figure) or even several randomly selected points from an explored space. If the difference on any characteristic dimension is more than some threshold (currently set as 2%), we skip such computer resource and provide opportunity to record this case as *suspicious* including all inputs and outputs for further validation and analysis by the community as described later in Section 6.2.2. Now, a user can easily select optimal cases shared by the community depending on the further application usage, i.e. the fastest variant (or probably with some balance in code size) to be used in a smart phone or cloud service, or smallest variant if it is used in some tiny devices with very limited resources, for example to support recent "Internet of Things" initiative.

6.2.2 Validating machine learning (classification and predictive modeling)

Optimization formalization and unification in our framework opens up another interesting possibility to crowdsource a global problem solving in compilation and architecture while avoiding explosion in the amount of experimental data. For example, we would like to understand if machine learning can be really efficient in predicting compiler optimizations. Current experimental scenarios attempt to address this problem by

-O3 -fif-conversion ***-fno-ALL***
-O3 –param max-inline-insns-auto=88 -finline-functions ***-fno-ALL***
-O3 -fregmove -ftree-vrp ***-fno-ALL***
-O3 -fomit-frame-pointer -fpeel-loops -ftree-fre ***-fno-ALL***
-O3 -falign-functions -fomit-frame-pointer -ftree-ch ***-fno-ALL***
-O3 -ftree-dominator-opts -ftree-loop-optimize -funswitch-loops ***-fno-ALL***
-O3 -ftree-ccp -ftree-forwprop -ftree-fre -ftree-loop-optimize ***-fno-ALL***
-O3 -finline-functions -fivopts -fprefetch-loop-arrays -ftree-loop-optimize -ftree-vrp ***-fno-ALL***
-O3 -fdce -fgcse -fomit-frame-pointer -freorder-blocks-and-partition -ftree-reassoc -funroll-all-loops ***-fno-ALL***
-O3 -fivopts -fprefetch-loop-arrays -fsched-last-insn-heuristic -fschedule-insns2 -ftree-loop-optimize -ftree-reassoc -ftree-ter ***-fno-ALL***
-O3 -fforward-propagate -fguess-branch-probability -fivopts -fmove-loop-invariants -freorder-blocks -ftree-ccp -ftree-ch -ftree-dominator-opts -ftree-loop-optimize -ftree-reassoc -ftree-ter -ftree-vrp -funroll-all-loops -funswitch-loops -fweb ***-fno-ALL***

Table 6.1 – *Some of the top performing combinations of optimization flags in GCC 4.6.3 out of 79 found optimization clusters found across Intel E5520 architecture using our buildbot on a local data center and several ARM-based mobile phones. Meta flag* ***-fno-ALL*** *means that all other optimization flags have been switched off when applying complexity reduction plugin and leaving only most influential flags.*

selecting a few benchmarks, tune each of them on a given platform for a few months, collecting a large amount of training data and then show that it is possible to build a model with some ad-hoc static/semantic or dynamic features to predict optimizations, usually from the same training set using cross-validation. Though technically correct, such approach is focusing only on "positive outcomes", prone to the same "big data" problem as described before and usually results in very limited studies covering a small part of computer systems that do not help to understand whether a model will predict well in industrial setup with many more benchmarks and features available. Instead, we would like to create and continuously update a pool of top performing optimizations for any given compiler that are different than -O3 and continuously cluster all available benchmarks in terms of those optimizations. The idea is that the benchmarks in the same optimization cluster naturally also share some features that can be used for prediction. At the same time, we would like to focus not only on high speedups (positive results) but also on slowdowns (negative results that are currently overlooked by the community) to be able to hint compiler designers that there is a possible problem with the internal optimization heuristic as it is simply not possible to add these optimization flags to -O3 to improve all the benchmarks.

We reused and extended experimental pipeline from the previous section to address above problems using spare computer resources and shared code and dataset samples while solving a problem of small training sets and more importantly focusing on both positive and negative results ("unexpected behavior"). To demonstrate our approach, we used developed buildbot to continuously optimize 285 shared code and dataset com-

binations from 8 popular benchmarks including NAS, MiBench, SPEC2000, SPEC2006, Powerstone, UTDSP and SNU-RT in terms of execution time on a local cloud service with 100 nodes, Intel E5520 processor (2.27GHz frequency, 8Mb last level cache) and GCC 4.6.3, using either the pool of top performing optimization combinations or at least 5000 random combinations of flags during 5 months. Whenever a new top performing combination of optimizations was found outside the pool, we applied it to all shared programs to perform online clustering while removing all redundant combinations that produce speedup similar to the new combination across all benchmarks. So far, our buildbot has found 79 distinct combinations of optimizations (optimization clusters) that cover all shared code and data set samples. Table 6.1 present some of the top performing pruned combinations of flags.

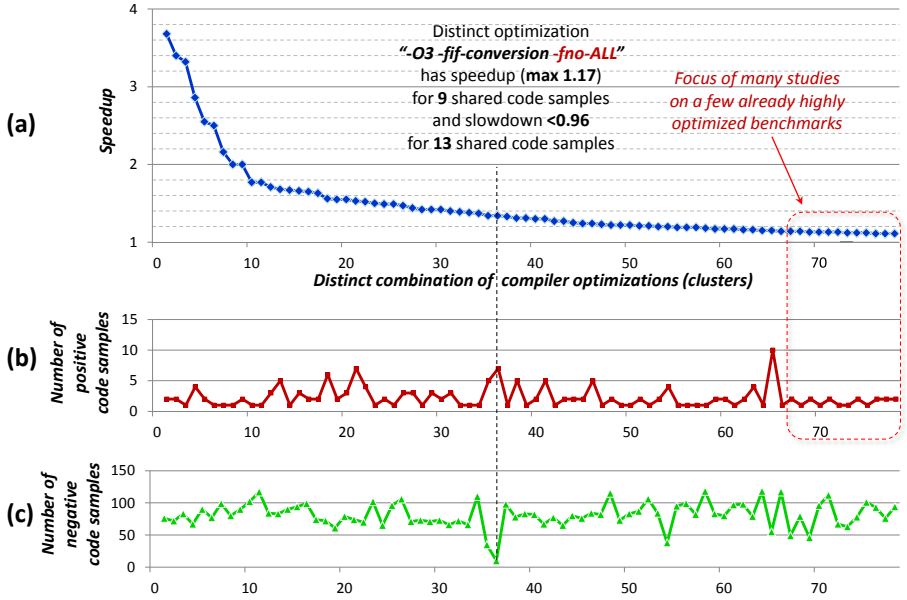


Figure 6.3 – *(a) 79 distinct combinations of optimizations (optimization clusters) covering all 285 shared code and dataset samples on Intel E5520, GCC 4.6.3 and at least 5000 random combinations of flags together with maximum speedup achieved within each optimization cluster; (b) number of benchmarks with speedup at least more than 1.1 for a given cluster; (c) number of benchmarks with speedup less than 0.96 (slowdown) for a given cluster.*

Figure 6.3 shows maximum speedups achieved for each optimization cluster across all benchmarks together with the number of benchmarks which achieve the highest speedup using this optimization (or at least more than 1.1) and the number of benchmarks with speedups less than 0.96 (slowdown) for the same optimization. For example, distinct combination of optimizations *-O3 -fif-conversion -fno-ALL* achieved maximum speedup on 7 benchmarks (including 1.17 speedup on at least one of these benchmarks) and slowdowns for 13 benchmarks. Note that unlike previous works, such clustering of continuously pruned combinations of optimization flags together with reproducible experimental setup cab already help compiler developers from our industrial partners to isolate and possibly solve code size, compilation time and performance regressions or other problems in production compilers, thus considerably enhancing existing buggy

Number of code and dataset samples	Prediction accuracy using optimized SVM
12 (from prior work) [55]	87%
285 from current work	56%

Table 6.2 – *Prediction accuracy when using optimized SVM with full cross-validation for 12 and 285 code and dataset samples from prior and current works respectively combined with all available semantic features (from MILEPOST GCC) and dynamic features (from hardware counters).*

buildbots. Furthermore, it helps to automatically systematize and prune large collections of benchmarks and data sets, leaving only representative ones for a given research problem (such as leaving only one code and related data set sample per optimization cluster). However, more importantly, it makes use of machine learning more understandable since all benchmarks in red clusters with maximum speedups are distinct - we just need to build a predictive model to associate a previously unseen program with one unique cluster.

At this stage, most of the existing works would attempt to build a predictive model using some off-the-shelf machine learning technique such as SVM or KNN and a few ad-hoc features. We also decided to validate such approach using SVM model from R package with full cross-validation for all 285 benchmarks used in our study and only 12 from the previous work on MILEPOST GCC [55]. Our feature vector **f** was automatically generated using 56 semantic features available in MILEPOST GCC (extracted for each benchmark at *-O1* optimization level after *pre* pass) combined with 30 hardware counters *("cycles", "instructions", "cache-references", "cache-misses", "L1-dcache-loads", "L1-dcache-load-misses", "L1-dcache-prefetches", "L1-dcache-prefetch-misses", "LLC-prefetches", "LLC-prefetch-misses", "dTLB-stores", "dTLB-store-misses", "branches", "branch-misses", "bus-cycles", "L1-dcache-stores", "L1-dcache-store-misses", "L1-icache-loads", "L1-icache-load-misses", "LLC-loads", "LLC-load-misses", "LLC-stores", "LLC-store-misses", "dTLB-loads", "dTLB-load-misses", "iTLB-loads", "iTLB-load-misses", "branch-loads", "branch-load-misses")* obtained using standard performance monitoring tool *perf* available in most Linux distributions by default.

Table 6.2 summarizes results of our modeling. When using just a few benchmarks, prediction accuracy is quite high and supports findings from other papers including [55]. However, interestingly, when adding considerably more benchmarks, prediction accuracy drops dramatically and starts exhibiting close to random behavior (50%). In order to understand such behavior, we decided to take a closer look at one of the optimization clusters and "deconstruct" it. We noticed that optimization combination *-O3 -fif-conversion -fno-ALL* is one of the simplest ones in our pool while having 7 benchmarks with positive speedup and 10 with negative ones. Unification of feature vectors in our framework allows to apply standard complexity reduction to incrementally remove all features one by one while rebuilding model and maintaining an adequate level of prediction accuracy. Naturally, it can also be done using statistical techniques such as ANOVA or PCA [17], but since we would like to isolate possible problem, we need

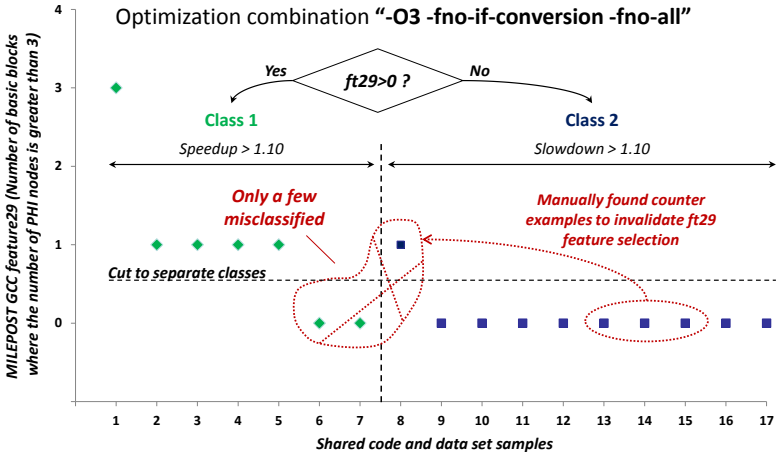


Figure 6.4 – *Automatic detection of the relevant feature(s) to predict optimization cluster "-O3 -fno-if-conversion -fno-ALL" using complexity reduction. However, we manually converted several code samples to provide counter examples that invalidated this feature and showed that using small training sets in many current studies can be totally misleading.*

precise analysis. Our pruning left only one semantic feature from MILEPOST GCC (ft29) that counts the number of basic blocks where the number of phi-nodes is greater than 3. Visualization at Figure 6.4 helps us to derive a decision that count of ft29 greater than 0 can effectively separate the two classes with only 3 mispredictions out of 17.

In an industrial setup, we also need to understand whether this feature makes sense and how to use this information to improve a compiler. Therefore, we exposed all these experimental data to our industrial colleagues and compiler developers who confirmed experimental results but could not explain this feature. Considering that confirming relevance of a feature may not be straightforward, we decided to try to find a counter example instead to invalidate this result. We selected a simple *blocksort function* from *bzip2* that has 0 phi-nodes and tried to manually add phi-nodes by transforming source code as following (added lines are highlighted):

```
...
volatile int sum, value = 3;
int sumA = 0;
int sumB = 0;
int sumC = 0;
for (j = ftab[ss⟨⟨8] & (~((1⟨⟨ 21)));j⟨copyStart[ss];j++)
    k = ptr[j] - 1;
     sumA += value;
     sumB += value;
     sumC += value;
...
```

This manual transformation added 3 PHI nodes to the code, resulting in a change of *ft29* threshold value from 0 to 1 while speedup remained the same. We performed similar transformation in a few other benchmarks that did not influence the original speedup while changing *ft29* from 0 to any number thus invalidating original decision

separating 2 classes and showing that our model is misleading. At the same time, we shared all counter examples in a buildbot repository thus providing code samples with unusual and reproducible optimization behavior similar to buggy buildbot where samples causing compiler crashes are continuously collected and analyzed.

From this example, it is evident that our community has often been using machine learning for compiler optimization in a wrong way: the fundamental problem is that many popular off-the-shelf statistical models were originally developed for pattern recognition and can work well only with a large amount of training data and features available such as thousands or even millions of public images. Our training set even with numerous features and hundreds of benchmarks is simply too small to build statistically meaningful model. At the same time, relatively high prediction accuracy on very small training sets can now be explained by finding some meaningless hyperplanes in a sparse feature space while failing to find any relevant correlation. This finding supports our idea to move away from "black box" machine learning approaches at least at this stage while focusing our effort to add much more benchmarks and use knowledge of domain specialists to collaboratively search and explain relevant features.

6.3 Learning dataset features to enable adaptive software

Though we demonstrated how our approach and methodology can help automate classification of shared software species to improve optimization predictions, it still did not solve another fundamental problem of static compilation - lack of run-time information. On the other hand, since cM continuously records unexpected behavior, it helped to automatically detect that one of the real customer's software species (image B&W threshold filter from a surveillance camera application similar to one shown in Figure 1.2) requires two distinct optimizations with around 20% improvement in execution time on Intel Core i5-2540M across all shared images (data set samples) as shown in Figure 6.5.

In order to understand such behavior, we can now reuse the same clustering methodology to classify available data sets and expose those features that can explain such behavior and separate optimization classes. Compiler designers again helped us analyze this software species and gradually identified a suspicious "sub-species", causing an unusual behavior: *(temp1 ⟩ T) ? 255 : 0*. One optimization class included "if conversion" transformation, which added several predicated statements that may degrade performance if additional branches are rarely taken due to a few additional useless cycles to check branch condition. At this stage, compiler designers concluded that it is a well-known run-time dependency which is difficult or even impossible to solve in static compilers. Nevertheless, one of the volunteers noticed that some images shown in Figure 6.5 were captured during the day and some during the night. This helped us find new, simple and relevant feature related to both data set and the environment state

Dataset Class	Optimization class	
	Class 1	Class 2
Shared data set sample$_1$ 	**+21.5%** ± 1.5% improvement in execution time	**-8.2%** ± 1.5% degradation in execution time
Shared data set sample$_2$ 	**-11.9%** ± 1.5% degradation in execution time	**+17.3%** ± 1.5% I mprovement in execution time

Figure 6.5 – *Detecting missing dataset feature "time of the day" with the help of the community. Such feature enables adaptive software species that performs well across all inputs.*

"time of the day" that effectively separated two optimization classes.

At the same time, when analyzing multiple executions of image corner detection benchmark on a smart phone as shown in Figure 6.2, we noticed occasional 4x difference in execution times. Normally, most of the studies would simply skip such experiment. However, now we have an opportunity to record, reproduce and visualize such cases as shown in Figure 6.6.

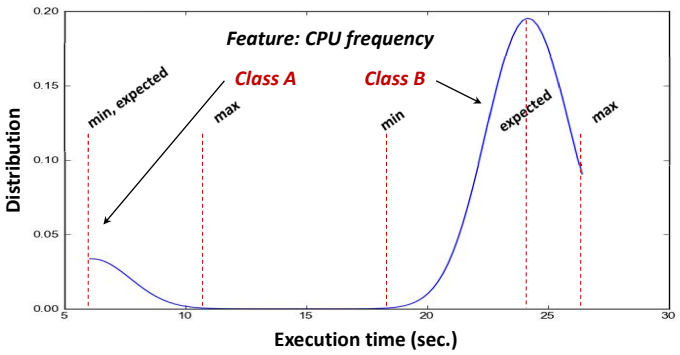


Figure 6.6 – *Unexpected behavior helped to identify and share missing feature.*

Simple analysis showed that our phone was often in the low power state at the beginning of the experiments and then gradually switched to the high-frequency state (4x difference in frequency). Though obvious, this information allowed us to add CPU frequency scaler to the pipeline and universal feature, state and choice vectors **f**, **s**, and **c** respectively together with *cpufreq* wrapper, thus using exposed "unexpected behavior" to improve public experimental pipeline and help community to avoid pitfalls in their next experiments while gradually extending collection of features in our system.

This real example demonstrates how our approach can help *collaboratively find missing and nontrivial features that may not even exist and have to be exposed* to improve

optimization prediction. Furthermore, our approach helped substitute the threshold filter in the customer's real software by a shared cM plugin consisting of two differently optimized clones of this filter and a compact decision tree. This decision tree selects an appropriate clone at run-time based on the features of a data set, hardware and environment state used. Therefore, our Collective Mind approach can also help make statically compiled software easily adaptable to different contexts as conceptually shown in Figure 6.7.

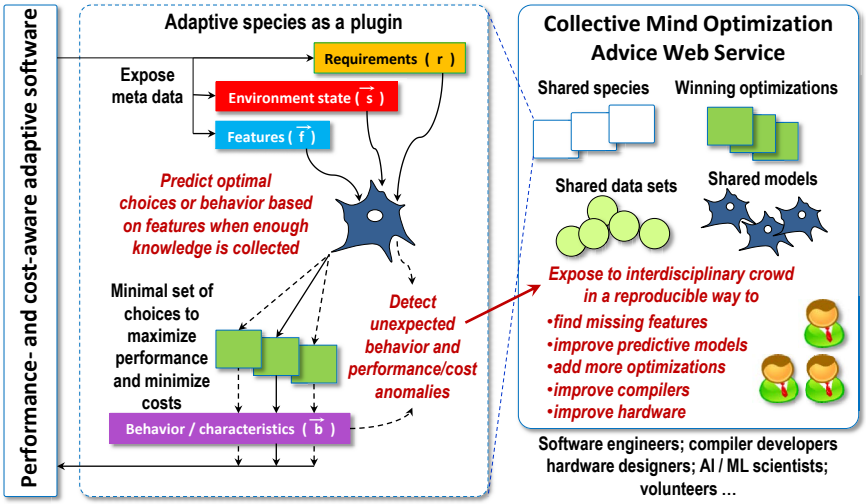


Figure 6.7 – *Concept of performance- and cost-aware self-tuning software assembled from cM plugins.*

Moreover, such software will be continuously optimized with the help of the community while maximizing its performance, minimizing development costs, improving productivity of software engineers and reducing time to market. Interestingly, cM approach can also help solve "big data problem" that we experienced in the first public cTuning framework [52, 55]. Rather than collecting and preserving all possible information from participating users, we can validate incoming data against existing models and save only unexpected behavior. We believe that presented approach can eventually enable performance- and cost-aware software engineering. We envisage that instead of struggling to integrate various ad-hoc optimization heuristics to their software projects similar to one shown in Figure 1.4, engineers will simply need to expose various features from data sets, software, hardware and environment state for their software pieces. These features will then be correlated with the top performing optimizations either automatically or with the help of the community to gradually minimize execution time, power consumption, code size, compilation time, faults, and other costs.

6.4 Summary

This chapter demonstrated how to validate, share, enhance and systematize our past research knowledge and practical experience particularly on program optimization and machine learning (largely overlooked by our community) using crowdsourcing. Presented evolutionary community-driven approach and practical, portable, plugin-based framework help to unify and connect together existing ad-hoc tools while liberating researchers and particularly students or reviewers from a tedious and sometimes impossible task of re-implementing ad-hoc experimental setups from numerous publications. It also helps researchers to quickly prototype their ideas in days rather than months by reusing and customizing shared experimental setups and data while focusing all their efforts and creativity on either solving existing problems while reusing, improving and optimizing shared predictive models, finding missing features or exposing existing contributing features, or developing truly novel approaches.

7

Conclusion and future work

The computer engineering community has been desperately trying to find some practical ways to automatically improve software performance while reducing power consumption and other usage costs across numerous and rapidly evolving computer systems for several decades [5, 38, 118, 66, 71]. In this thesis, we presented a novel and practical approach inspired by natural sciences and Wikipedia that may help collaboratively solve this problem while improving productivity of software developers. The biggest challenge in this approach is to connect together, systematize and make practical various techniques and tools from different interdisciplinary domains often overlooked by our community into a coherent, extensible and top-down optimization and classification methodology.

The backbone of our approach is a public repository of optimization knowledge at c-mind.org/repo. It allows the software engineering community to gradually share their most frequently used software pieces (computational species) together with various possible inputs and features. All shared species are then continuously and randomly optimized and executed with randomly selected inputs either as standalone pieces or within real software across numerous mobile phones, laptops and data centers provided by volunteers using our recent Collective Mind framework (cM). In contrast with a very few existing public repositories, notably SPEC and Phoronix benchmarking platforms [127, 110], cM also continuously classifies best found optimizations while exposing unexpected behavior in a reproducible way. This, in turn, allows the interdisciplinary community to collaboratively correlate found classes with gradually exposed features from the software, hardware, datasets and environment state either manually or using popular big data predictive analytics [9, 68]. Resulting predictive models are then integrated into cM plugins together with several pre-optimized (specialized) versions of a given species that maximize performance and minimize costs across as many inputs,

hardware and environment states as possible, as described in [96]

Software engineers can now assemble self-tuning applications just like "LEGO" from the shared cM plugins with continuously optimized species. Such software not only can adapt to the running hardware and context, but also continue improving its performance and minimize usage costs when more collective knowledge is available. This can help change current computer engineering methodology since software engineers do not have to wait anymore until hardware or compilers become better. Instead, the software engineering community gradually creates a large, diverse and realistic benchmark together with a public and continuously improving optimization advice system that helps improve and validate future compilers and hardware. For example, we envision that our approach will also help simplify compilers and convert them into generic libraries of code analysis, optimization and generation routines orchestrated by cM-like frameworks.

To avoid the fate of many projects that vanish shortly after publication, we agreed with our partners to share most of the related code and data at our public optimization repository to continue further community-driven developments. For example, with the help of our supporters, we already shared around 300 software species and collected around 15000 possible data sets. At the same time, we also shared various features as cM meta-data from our past research on machine learning based optimization including MILEPOST semantic code properties [55], code patterns and control flow graph extracted by our GCC/LLVM Alchemist plugin [59], image and matrix dimensions together with data set sizes from [96], OS parameters, system descriptions, hardware performance counters, CPU frequency and many others.

Public availability of such a repository and open source cM infrastructure allowed us to validate our approach in several major companies. For example, we demonstrated how our industry colleagues managed to enhance their in-house benchmarking suites to considerably improve optimization heuristics of their production GCC compiler for a number of ARM and Intel based processors while detecting several architectural errors during validation of new hardware configurations. Finally, presented approach helped to convert an important customer statically compiled image processing application into a self-tuning one that maximizes performance to reach real time constraints and minimize all other costs including energy, overall development and tuning effort, and time to market.

As a part of the future work, we plan to simplify as much as possible the experience of software engineers and volunteers wishing to participate in our project. Therefore, we are currently extending our cM framework to automate identification, extraction and sharing of the frequently used and most time consuming software pieces and their features in real programs. For this purpose, we plan to use and extend our Interactive Compilation Interface for GCC and LLVM while connecting cM framework with Eclipse IDE [41] to simplify integration of our cM wrappers and performance/cost monitor-

ing plugins with real applications, with Docker [37] and CARE [78] to automatically detect all software dependencies for sharing, and with Phoronix open benchmarking infrastructure [110] to add even more realistic software pieces to our repository. This community-driven effort continues [57].

A

Reproducing experiments

A.1 Grid5000 Framework

The ever-increasing computing requirements have necessitated the use of large-scale, highly parallel computing systems. Modern computers, no matter how much powerful, cannot meet the existing computing requirements required by the complex algorithms. This has led to the development of massive-scale, distributed computing systems in the form of grid. A grid is a collection of huge number of clusters working in parallel on a given (complex) problem which is beyond the capabilities of a single computer. Among the most notable grids all over the world, Grid5000 is a large-scale and reconfigurable infrastructure developed in France in 2003 to support experimental-driven research in parallel and distributed systems. Grid5000 has been used as a testbed in all the experimens performed in our research. It offers a highly reconfigurable, controllable and monitorable experimental platform by providing access to a large amount of resources including 1000 nodes and 8000 CPU cores, grouped in homogeneous clusters and featuring various technologies such as 10G Ethernet, Infiniband, GPUSs and Xeon PHI, etc. Additionally, Grid5000 has the following salient features [7][63].

- Adaptibility, reconfigurability and controllability.
- In-depth analysis and monitoring of large-scale distributed systems (high-performance computing, grids, peer-to-peer systems, cloud computing, and others).
- Support for the reproducibility of the experimental results pertaining to benchmarking, simulations and any other domain.
- Constant evolution and support for the major technological trends and state-of-the-art innovations related to distributed and parallel systems from hardware as

well as software perspective.

In the next sections, we describe the steps involved in carrying out experimentations on Grid5000 testbed and our exeprimental setup.

A.1.1 Experimental setup on Grid5000

The steps involved in conducting an experiment on Grid5000 is as follows.

A.1.1.1 Reservation of resources

The first step requires locating and reserving the resources for the intended experiment. The reservation of resources can be performed either (i) by manually searching the resources with their description in a web interface and then making a reservation or (ii) by specifying the experimentation requirement to the system which in turn allocates the appropriate resources.

A.1.1.2 Deployment

This step involves deploying the experimental apparatus on the resources. The deployment may be performed either by using pre-configured environments or by installing user-specified environments. An environment usually comprises a compressed file of the operating system image and a kernel file specifying which kernel to boot.
The default scheme of Grid5000 generally allocates a larger part of the disk space to the temporary file system (*/tempfs*) which is flushed out at each restart of the system. The other part of the disk space is reserved for the root file system where the image is copied. We customized the default environment by integrating our software and data comprising compilers, performance monitoring tools, benchmarks, and the collective mind framework with the image file consisting of the operating system and the kernel. The is useful for copying the whole experimental apparatus for each experimentation without requiring to copy the software and the data for each experiment. We also customized the Grid5000 allocated disk space by reserving more space for the root file system where our image is copied and reducing the space for temporary file system. For partitioning the disk space accordingly, we run the following script.

```
---
SetDeploymentEnvUntrusted:
  create_partition_table:
      substitute:
        - action: send
          file: partitions
          destination: $KADEPLOY_TMP_DIR
          name: send_partitions
        - action: exec
          name: partitioning_with_parted
```

```
          command: parted -a optimal /dev/sda --script $(cat $KADEPLOY_TMP_DIR/↩
              partitions)
# add formating step to kadeploy
  format_deploy_part:
      post-ops:
        - action: run
          name: format_with_mkfs
          file: format
SetDeploymentEnvKexec:
  create_partition_table:
      substitute:
        - action: send
          file: partitions
          destination: $KADEPLOY_TMP_DIR
          name: send_partitions
        - action: exec
          name: partitioning_with_parted
          command: parted -a optimal /dev/sda --script $(cat $KADEPLOY_TMP_DIR/↩
              partitions)
# add formating step to kadeploy
  format_deploy_part:
      post-ops:
        - action: run
          name: format_with_mkfs
          file: format
# we don't need those both step so we escape it.
  format_tmp_part:
      substitute:
        - action: exec
          name: remove_format_tmp_part_step
          command: /bin/true
  format_swap_part:
      substitute:
        - action: exec
          name: remove_format_swap_part_step
          command: /bin/true
```

```
mklabel msdos
u GB mkpart primary 0% 6%
u GB mkpart primary 6% 100%
align-check optimal 1
align-check optimal 2
```

A.1.1.3 Automating deployment

Grid5000 currenlty has over 500 users who usually require several nodes for experimentation. This results in huge delays in node availability for new users. Instead of constantly monitoring the node availability, we automate the deployment process such that our image is copied as soon as a node is available. We run the following script for automatic node deployment.

```
#!/bin/sh
```

```
2
3  # Put comments here
4
5  NODE_FILE=$OAR_FILE_NODES
6
7  if [ -z "$NODE_FILE" ]; then
8      echo "ERROR: Machines Unavailable"
9      exit
10 fi
11
12 kadeploy -e sid-x64-base-1.1-unipf -f $OAR_FILE_NODES
13
14 a=1
15 for node in $(cat $NODE_FILE | uniq); do
16         scp -o StrictHostKeyChecking=no /home/orsay/awmemon/setup/CBench/↩
               automotive_susan_e/$a.txt root@$node:/root/ccc/apps/ccc--bench-list.txt
17         scp -o StrictHostKeyChecking=no /home/orsay/awmemon/setup/CBench/↩
               automotive_susan_e/run$a root@$node:/root/ccc--run-bench
18         scp -o StrictHostKeyChecking=no /home/orsay/awmemon/ccc-run--glob-flags.sh ↩
               root@$node:/root/ccc/apps/
19         ssh -o StrictHostKeyChecking=no root@$node /root/chima
20         let "a=a+1"
21         echo $node
22 done
23
24 sleep 13h
```

A.2 Sharing artifacts for reproducibility

The Collective Mind's *ctuning* repository contains all the supported packages(compilers, libraries and tools), benchmarks, datasets, and scenarios (1.5Gb). The repository can be downloaded at https://drive.google.com/file/d/0B-wXENVfIO82UEdyYWdpSGIt-eWs/view?usp=sharing.

The *shared* repository can be downloaded at https://drive.google.com/file/d/0B-wXENVfIO82T3B4TklVakxnNXM/view?usp=sharing

The latest release of Collective Mind framework is available at http://sourceforge.net/projects/c-mind/files/latest/download

A.2.1 Compiler flags pruning

As mentioned in Section 5.3, 285+ codelets extracted from several popular benchmakrs are shared as part of Collective Mind framework. In this section, we show some of the codelets with their best found combination of flags using Collective Mind. We also show (in blue), the semi-manually pruned combination of flags.

-O3 **-falign-functions -falign-jumps** -fno-align-labels **-falign-loops** -fno-asynchronous-unwind-tables -fno-branch-count-reg **-fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves** -fno-combine-stack-adjustments **-fcommon -fcompare-elim -fconserve-stack -fcprop-registers -fcrossjumping** -fno-cse-follow-jumps -fno-cx-limited-range **-fdce -fdefer-pop** -fno-delete-null-pointer-checks **-fdevirtualize** -fno-dse -fno-early-inlining **-fexpensive-optimizations -fforward-propagate** -fno-gcse -fno-gcse-after-reload **-fgcse-las -fgcse-lm -fgcse-sm** -fno-graphite-identity **-fguess-branch-probability** -fno-if-conversion -fno-if-conversion2 -fno-inline-functions **-finline-functions-called-once -finline-small-functions** -fno-ipa-cp -fno-ipa-cp-clone **-fipa-**

matrix-reorg **-fipa-profile** **-fipa-pta** **-fipa-pure-const** **-fipa-reference** **-fipa-sra** **-fivopts** **-fjump-tables** **-fmath-errno** -fno-loop-block **-floop-flatten** **-floop-interchange** -fno-loop-parallelize-all **-floop-strip-mine** **-fmerge-constants** -fno-modulo-sched **-fmove-loop-invariants** -fno-omit-frame-pointer **-foptimize-register-move** -fno-optimize-sibling-calls **-fpeel-loops** **-fpeephole** **-fpeephole2** **-fpredictive-commoning** -fno-prefetch-loop-arrays **-fregmove** **-frename-registers** -fno-reorder-blocks -fno-reorder-blocks-and-partition **-freorder-functions** **-frerun-cse-after-loop** **-freschedule-modulo-scheduled-loops** **-fsched-critical-path-heuristic** **-fsched-dep-count-heuristic** -fno-sched-group-heuristic -fno-sched-interblock **-fsched-last-insn-heuristic** **-fsched-pressure** -fno-sched-rank-heuristic -fno-sched-spec **-fsched-spec-insn-heuristic** **-fsched-spec-load** **-fsched-spec-load-dangerous** **-fsched-stalled-insns** **-fsched-stalled-insns-dep** -fno-sched2-use-superblocks **-fschedule-insns** **-fschedule-insns2** **-fshort-enums** **-fsigned-zeros** **-fsel-sched-pipelining** -fno-sel-sched-pipelining-outer-loops **-fsel-sched-reschedule-pipelined** -fno-selective-scheduling -fno-selective-scheduling2 **-fsignaling-nans** **-fsingle-precision-constant** -fno-split-ivs-in-unroller **-fsplit-wide-types** **-fstrict-aliasing** -fno-thread-jumps **-ftrapping-math** **-ftree-bit-ccp** -fno-tree-builtin-call-dce **-ftree-ccp** **-ftree-ch** **-ftree-copy-prop** **-ftree-copyrename** -fno-tree-cselim -fno-tree-dce **-ftree-dominator-opts** **-ftree-dse** -fno-tree-forwprop -fno-tree-fre **-ftree-loop-distribute-patterns** **-ftree-loop-distribution** -fno-tree-loop-if-convert -fno-tree-loop-if-convert-stores **-ftree-loop-im** -fno-tree-loop-ivcanon **-ftree-loop-optimize** **-ftree-lrs** **-ftree-phiprop** **-ftree-pre** -fno-tree-pta -fno-tree-reassoc **-ftree-scev-cprop** **-ftree-sink** -fno-tree-slp-vectorize -fno-tree-sra **-ftree-switch-conversion** -fno-tree-ter **-ftree-vect-loop-version** -fno-tree-vectorize -fno-tree-vrp -fno-unroll-all-loops -fno-unsafe-loop-optimizations -fno-unsafe-math-optimizations **-funswitch-loops** -fno-variable-expansion-in-unroller **-fvect-cost-model** -fno-web

-O3 -fguess-branch-probability -fivopts -fmove-loop-invariants -frename-registers -fsched-critical-path-heuristic -fsched-pressure -fschedule-insns -ftree-ccp -ftree-ch -ftree-dominator-opts -ftree-loop-optimize ***-fno-ALL***

mc.codelet__9.1

-O3 **-falign-functions** -fno-align-jumps -fno-align-labels **-falign-loops** **-fasynchronous-unwind-tables** -fno-branch-count-reg -fno-branch-target-load-optimize2 **-fbtr-bb-exclusive** **-fcaller-saves** -fno-combine-stack-adjustments -fno-common **-fcompare-elim** **-fconserve-stack** -fno-cprop-registers **-fcrossjumping** -fno-cse-follow-jumps -fno-cx-limited-range **-fdce** -fno-defer-pop **-fdelete-null-pointer-checks** -fno-devirtualize -fno-dse -fno-early-inlining -fno-expensive-optimizations -fno-forward-propagate **-fgcse** **-fgcse-after-reload** **-fgcse-las** **-fgcse-lm** -fno-gcse-sm -fno-graphite-identity **-fguess-branch-probability** **-fif-conversion** **-fif-conversion2** **-finline-functions** -fno-inline-functions-called-once -fno-inline-small-functions -fno-ipa-cp -fno-ipa-cp-clone **-fipa-matrix-reorg** **-fipa-profile** **-fipa-pta** **-fipa-pure-const** -fno-ipa-reference **-fipa-sra** **-fno-ivopts** -fno-jump-tables -fno-math-errno **-floop-block** **-floop-flatten** -fno-loop-interchange -fno-loop-parallelize-all -fno-loop-strip-mine **-fmerge-constants** **-fmodulo-sched** **-fmove-loop-invariants** **-fomit-frame-pointer** **-foptimize-register-move** **-foptimize-sibling-calls** **-fpeel-loops** **-fpeephole** **-fpeephole2** **-fpredictive-commoning** **-fprefetch-loop-arrays** -fno-regmove -fno-rename-registers **-freorder-blocks** **-freorder-blocks-and-partition** **-freorder-functions** **-frerun-cse-after-loop** **-freschedule-modulo-scheduled-loops** -fno-sched-critical-path-heuristic **-fsched-dep-count-heuristic** -fno-sched-group-heuristic -fno-sched-interblock -fno-sched-last-insn-heuristic **-fsched-pressure** **-fsched-rank-heuristic** -fno-sched-spec -fno-sched-spec-insn-heuristic **-fsched-spec-load** **-fsched-spec-load-dangerous** **-fsched-stalled-insns** -fno-sched-stalled-insns-dep **-fsched2-use-superblocks** -fno-schedule-insns **-fschedule-insns2** -fno-short-enums **-fsigned-zeros** -fno-sel-sched-pipelining -fno-sel-sched-pipelining-outer-loops **-fsel-sched-reschedule-pipelined** -fno-selective-scheduling -fno-selective-scheduling2 **-fsignaling-nans** -fno-single-precision-constant **-fsplit-ivs-in-unroller** **-fsplit-wide-types** **-fstrict-aliasing** **-fthread-jumps** **-ftrapping-math** **-ftree-bit-ccp** **-ftree-builtin-call-dce** -fno-tree-ccp **-ftree-ch** **-ftree-copy-prop** -fno-tree-copyrename -fno-tree-cselim -fno-tree-dce -fno-tree-dominator-opts -fno-tree-dse **-ftree-forwprop** -fno-tree-fre -fno-tree-loop-distribute-patterns -fno-tree-loop-distribution **-ftree-loop-if-convert** -fno-tree-loop-if-convert-stores -fno-tree-loop-im -fno-tree-loop-ivcanon -fno-tree-loop-optimize -fno-tree-lrs **-ftree-phiprop** -fno-tree-pre **-ftree-pta** -fno-tree-reassoc **-ftree-scev-cprop** **-ftree-sink** -fno-tree-slp-vectorize **-ftree-sra** -fno-tree-switch-conversion **-ftree-ter** -fno-tree-vect-loop-version -fno-tree-vectorize **-ftree-vrp** **-funroll-all-loops** **-funsafe-loop-optimizations** -fno-unsafe-math-optimizations -fno-unswitch-loops -fno-variable-expansion-in-unroller -fno-vect-cost-model **-fweb**

-O3 -fcaller-saves -fdce -fguess-branch-probability -fmove-loop-invariants -fomit-frame-pointer -fsched-dep-count-heuristic -fsched2-use-superblocks -fschedule-insns2 -ftree-copy-prop -ftree-ter -ftree-vrp ***-fno-ALL***

pixel.codelet__3.1

-O3 **-falign-functions** **-falign-jumps** -fno-align-labels **-falign-loops** -fno-asynchronous-unwind-tables -fno-branch-count-reg -fno-branch-target-load-optimize2 **-fbtr-bb-exclusive** **-fcaller-saves** **-fcombine-stack-adjustments** -fno-common **-fcompare-elim** **-fconserve-stack** **-fcprop-registers** -fno-crossjumping **-fcse-follow-jumps** **-fcx-limited-range** **-fdce** **-fdefer-pop** **-fdelete-null-pointer-checks** -fno-devirtualize **-fdse** -fno-early-inlining -fno-expensive-optimizations -fno-forward-propagate -fno-gcse **-fgcse-after-reload** -fno-gcse-las **-fgcse-lm** **-fgcse-sm** **-fgraphite-identity** -fno-guess-branch-probability **-fif-conversion** **-fif-conversion2** **-finline-functions** -fno-inline-functions-called-once -fno-inline-small-functions -fno-ipa-cp **-fipa-cp-clone** **-fipa-matrix-reorg** **-fipa-profile** **-fipa-pta** **-fipa-pure-const** **-fipa-reference** -fno-ipa-sra **-fivopts** **-fjump-tables** **-fmath-errno** -fno-loop-block **-floop-flatten** **-floop-interchange** -fno-loop-parallelize-all **-floop-strip-mine** -fno-merge-constants -fno-modulo-sched -fno-move-loop-invariants -fno-omit-frame-pointer -fno-optimize-register-move -fno-optimize-sibling-calls **-fpeel-loops** -fno-peephole **-fpeephole2** **-fpredictive-commoning** **-fprefetch-loop-arrays** **-fregmove** -fno-rename-registers **-freorder-blocks** -fno-reorder-blocks-and-partition -fno-reorder-functions **-frerun-cse-after-loop** **-freschedule-modulo-scheduled-loops** -fno-sched-critical-path-heuristic **-fsched-dep-count-heuristic** **-fsched-group-heuristic** **-fsched-interblock** -fno-sched-last-insn-heuristic **-fsched-pressure** -fno-sched-rank-heuristic -fno-sched-spec -fno-sched-spec-insn-heuristic -fno-sched-spec-load **-fsched-spec-load-dangerous** **-fsched-stalled-insns** -fno-sched-stalled-insns-dep **-fsched2-use-superblocks** -fno-schedule-insns **-fschedule-**

insns2 -fno-short-enums **-fsigned-zeros** -fno-sel-sched-pipelining **-fsel-sched-pipelining-outer-loops** **-fsel-sched-reschedule-pipelined** **-fselective-scheduling** **-fselective-scheduling2** -fno-signaling-nans **-fsingle-precision-constant** -fno-split-ivs-in-unroller **-fsplit-wide-types** **-fstrict-aliasing** **-fthread-jumps** -fno-trapping-math -fno-tree-bit-ccp -fno-tree-builtin-call-dce **-ftree-ccp** -fno-tree-ch **-ftree-copy-prop** **-ftree-copyrename** **-ftree-cselim** **-ftree-dce** -fno-tree-dominator-opts -fno-tree-dse **-ftree-forwprop** -fno-tree-fre -fno-tree-loop-distribute-patterns -fno-tree-loop-distribution **-ftree-loop-if-convert** -fno-tree-loop-if-convert-stores **-ftree-loop-im** **-ftree-loop-ivcanon** **-ftree-loop-optimize** -fno-tree-lrs -fno-tree-phiprop -fno-tree-pre -fno-tree-pta **-ftree-reassoc** **-ftree-scev-cprop** -fno-tree-sink **-ftree-slp-vectorize** **-ftree-sra** **-ftree-switch-conversion** **-ftree-ter** **-ftree-vect-loop-version** -fno-tree-vectorize **-ftree-vrp** **-funroll-all-loops** -fno-unsafe-loop-optimizations **-funsafe-math-optimizations** -fno-unswitch-loops -fno-variable-expansion-in-unroller **-fvect-cost-model** -fno-web

-O3 -fcse-follow-jumps -fdce -fgraphite-identity -fregmove -freorder-blocks -ftree-copy-prop -ftree-forwprop -ftree-loop-optimize -ftree-ter -ftree-vrp ***-fno-ALL***

`dct.codelet__18.1`

Best combination of flags for all 285+ shared codelets and their corresponding pruned combination of flags is publicly available at `https://github.com/awam/opts_prune_pub`.

A.3 Crowdsourcing auto-tuning using mobile devices

There is also a continuing effort for crowdtuning using mobile phones and tablets. **Collective Mind Node**[101] is a result of such effort and is available for all Android based smartphones and tablets at `https://play.google.com/store/apps/details?id=com.collective_mind.node`. Mobile devices participating in continuous crowdtuning can be viewed at `http://ctuning.org/crowdtuning-mobiles`. Information pertaining to to mobile processors can be obtained at `http://ctuning.org/crowdtuning-processors`. Crowdtuning results for benchmarks using mobile devices is available at `http://ctuning.org/crowdtuning-results`.

Bibliography

[1] Acovea: Using natural selection to investigate software complexities. http://www.coyotegulch.com/products/acovea.

[2] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.

[3] Alfred V Aho. *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India, 2003.

[4] Scott Ambler. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[5] Krste Asanovic et.al. The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[6] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Not.*, 28(6):300–313, June 1993.

[7] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, et al. Adding virtualization capabilities to the grid'5000 testbed. In *Cloud Computing and Services Science*, pages 3–20. Springer, 2013.

[8] Paul Belleflamme, Thomas Lambert, and Armin Schwienbacher. Crowdfunding: Tapping the right crowd. *Journal of Business Venturing*, 29(5):585–609, 2014.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1st ed. 2006. corr. 2nd printing 2011 edition, October 2007.

[10] Richard Blum. *Professional assembly language*. John Wiley & Sons, 2007.

[11] F. Bodin, T. Kisuki, P.M.W. Knijnenburg, M.F.P. 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.

[12] Edwin V. Bonilla, Christopher K. I. Williams, Felix V. Agakov, John Cavazos, John Thomson, and Michael 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.

[13] Steven Burrows, Martin Potthast, and Benno Stein. Paraphrase acquisition via crowdsourcing and machine learning. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 4(3):43, 2013.

[14] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, January 1997.

[15] Rosario Cammarota, Alexandru Nicolau, Alexander V. Veidenbaum, Arun Kejariwal, Debora Donato, and Mukund Madhugiri. On the determination of inlining vectors for program optimization. In Jhala and Bosschere [79], pages 164–183.

[16] 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.

[17] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March 2007.

[18] Ccc: Continuous collective compilation framework for iterative multi-objective optimization. http://cTuning.org/ccc.

[19] Yang Chen, Shuangde Fang, Lieven Eeckhout, Olivier Temam, and Chengyong Wu. Iterative optimization for the data center. *SIGARCH Comput. Archit. News*, 40(1):49–60, March 2012.

[20] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2010.

[21] Youngsoo Choi, Allan Knies, Luke Gerke, and Tin-Fook Ngai. The impact of if-conversion and branch prediction on program execution on the intel® itanium™

processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 182–191. IEEE Computer Society, 2001.

[22] Collective Mind Live Repo: public repository of knowledge about design and optimization of computer systems. http://c-mind.org/repo.

[23] clang.llvm.org. http://clang.llvm.org.

[24] GCC: the GNU Compiler Collection. http://gcc.gnu.org.

[25] LLVM: the low level virtual machine compiler infrastructure. http://llvm.org.

[26] Open64: an open source optimizing compiler suite. http://www.open64.net.

[27] Phoenix: software optimization and analysis framework for microsoft compiler technologies. https://connect.microsoft.com/Phoenix.

[28] Rose: an open source compiler infrastructure to build source-to-source program transformation and analysis tools. http://www.rosecompiler.org/.

[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.D. Cooper, P.J. 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] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, August 2002.

[32] 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.

[33] Cod: Public collaborative repository and tools for program and architecture characterization and optimization. http://cTuning.org/cdatabase.

[34] Jack W Davidson and Sanjay Jinturkar. Aggressive loop unrolling in a retargetable, optimizing compiler. In *Compiler Construction*, pages 59–73. Springer, 1996.

[35] Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(1):6, 2015.

[36] Shammi Didla, Aaron Ault, and Saurabh Bagchi. Optimizing aes for embedded devices and wireless sensor networks. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, page 4. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[37] Docker: open source lightweight container technology that can run processes in isolation. http://www.docker.org.

[38] Jack Dongarra et.al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[39] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F.P. 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.

[40] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification*. John Wiley and Sons, Inc, New-York, USA, 2001.

[41] Eclipse: popular open source integrated development environment. http://www.eclipse.org.

[42] Ran El-Yaniv, Dmitry Pechyony, and Elad Yom-Tov. Better multiclass classification via a margin-optimized single binary problem. *Pattern Recogn. Lett.*, 29(14):1954–1959, October 2008.

[43] T. W. Epps and Lawrence B. Pulley. A test for normality based on the empirical characteristic function. *Biometrika*, 70(3):pp. 723–726, 1983.

[44] Esto: Expert system for tuning optimizations. http://www.haifa.ibm.com/projects/systems/cot/esto/index.html.

[45] David Ferrucci et.al. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79, 2010.

[46] B. Franke, M.F.P. 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.

[47] G. Fursin, M. F. P. O'Boyle, O. Temam, and G. Watts. A fast and accurate method for determining a lower bound on execution time: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):271–292, January 2004.

[48] Grigori Fursin. Restoration of symbols with noise by neural network. In *Proceedings of the national conference on physical processes in devices of electronic and laser engineering at Moscow Institute of Physics and Technology*, pages 112–127, 1995.

[49] Grigori Fursin. Measurement of characteristics of neural elements with the aid of personal computer. In *Proceedings of the national conference on physical processes in devices of electronic and laser engineering at Moscow Institute of Physics and Technology*, pages 20–28, 1997.

[50] Grigori Fursin. Modeling of processes of learning and recognition in modified neural network. In *Proceedings of the national conference on physical processes in devices of electronic and laser engineering at Moscow Institute of Physics and Technology*, pages 102–111, 1997.

[51] Grigori Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.

[52] Grigori Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In *Proceedings of the GCC Developers' Summit*, June 2009.

[53] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier 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.

[54] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver 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.

[55] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams, and Michael F. P. O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[56] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[57] Grigori Fursin, Anton Lokhmotov, and Ed Plowman. Collective knowledge: towards r&d sustainability. In *Proceedings of DATE 2016 (Design, Automation and Test in Europe)*, March 2016.

[58] Grigori Fursin, Abdul Wahid Memon, Christophe Guillon, and Anton Lokhmotov. Collective mind, part II: Towards performance-and cost-aware software engi-

neering as a natural science. *arXiv preprint arXiv:1506.06256, 18th International Workshop on Compilers for Parallel Computing (CPC'15) London, UK*, January 2015.

[59] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, D. Malony, Allen, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(4):309–329, July 2014.

[60] Grigori Fursin and Olivier Temam. Collective optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.

[61] Huiji Gao, Geoffrey Barbier, and Rebecca Goolsby. Harnessing the crowdsourcing power of social media for disaster relief. *IEEE Intelligent Systems*, 26(3):10–14, 2011.

[62] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the Twenty-Second ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2007.

[63] Grid5000: A nationwide infrastructure for large scale parallel and distributed computing research. http://www.grid5000.fr.

[64] Dick Grune, Kees Van Reeuwijk, Henri E Bal, Ceriel JH Jacobs, and Koen Langendoen. *Modern compiler design*. Springer Science & Business Media, 2012.

[65] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[66] Mary Hall, David Padua, and Keshav Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, February 2009.

[67] William Harrod. Ubiquitous high performance computing (uhpc). Technical Report DARPA-BAA-10-37, DARPA, USA, 2010.

[68] Tony Hey, Stewart Tansley, and Kristin M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[69] 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.

[70] Geoffrey E. Hinton and Simon Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:2006, 2006.

[71] The HiPEAC vision on high-performance and embedded architecture and compilation (2012-2020). http://www.hipeac.net/roadmap, 2012.

[72] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[73] 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.

[74] Kenneth Hoste and Lieven Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[75] Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Joern Renneke, and Grigori 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.

[76] David H. Hubel. *Eye, Brain, and Vision (Scientific American Library, No 22)*. W. H. Freeman, 2nd edition, May 1995.

[77] Engin Ipek, Sally A. McKee, Bronis R. de Supinski, Martin Schulz, and Rich 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.

[78] Yves Janin, Cédric Vincent, and Rémi Duraffort. Care, the comprehensive archiver for reproducible execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, TRUST '14, pages 1:1–1:7, New York, NY, USA, 2014. ACM.

[79] Ranjit Jhala and Koen De Bosschere, editors. *Compiler Construction - 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7791 of *Lecture Notes in Computer Science*. Springer, 2013.

[80] Han Jiawei and Micheline Kamber. Data mining: concepts and techniques. *San Francisco, CA, itd: Morgan Kaufmann*, 5, 2001.

[81] Victor Jimenez, Isaac Gelado, Lluis Vilanova, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.

[82] Yaochu Jin. Fuzzy modeling of high-dimensional systems: complexity reduction and interpretability improvement. *Fuzzy Systems, IEEE Transactions on*, 8(2):212–221, 2000.

[83] Online json introduction. http://www.json.org.

[84] Ece Kamar, Severin Hacker, and Eric Horvitz. Combining human and machine intelligence in large-scale crowdsourcing. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 467–474. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[85] Firas Khatib, Seth Cooper, Michael D Tyka, Kefan Xu, Ilya Makedon, Zoran Popović, David Baker, and Foldit Players. Algorithm discovery by protein folding game players. *Proceedings of the National Academy of Sciences*, 108(47):18949–18953, 2011.

[86] Bongjae Kim, Sangho Yi, Yookun Cho, and Jiman Hong. Impact of function inlining on resource-constrained embedded systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 287–292. ACM, 2009.

[87] Minje Kim and Paris Smaragdis. Collaborative audio enhancement: Crowdsourced audio recording. *Urbana*, 51:61874, 2014.

[88] T. Kisuki, P.M.W. Knijnenburg, and M.F.P. 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.

[89] 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.

[90] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, October 1975.

[91] Pedro Larrañaga and Jose A. Lozano. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[92] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012.

[93] Hugh Leather, Edwin V. Bonilla, and Michael F. P. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO*, pages 81–91. IEEE Computer Society, 2009.

[94] Xiaoming Li, Maria Jesus Garzaran, and David A. Padua. Optimizing sorting with machine learning algorithms. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[95] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[96] Lianjie Luo, Yang Chen, Chengyong Wu, Shun Long, and Grigori 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.

[97] 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.

[98] Abdul Wahid Memon and Grigori Fursin. Crowdtuning: systematizing auto-tuning using predictive modeling and crowdsourcing. In *PARCO mini-symposium on "Application Autotuning for HPC (Architectures)"*, Munich, Allemagne, September 2013.

[99] Khan Minhaj. *Code Specialization Strategies for High Performance Architectures*. PhD thesis, Universit´e de Versailles Saint-Quentin-en-Yvelines, France, 2008.

[100] MISC-cm. Collective Mind: open-source plugin-based infrastructure and repository to enable collaborative, reproducible and systematic research and experimentation in computer engineering. http://c-mind.org, -.

[101] MISC-cmn. Collective Mind Node: Android application connected to Collective Mind repository to crowdsource characterization and optimization of computer systems using off-the-shelf mobile phones and tablets. https://play.google.com/store/apps/details?id=com.collective_mind.node, -.

[102] MISC-google-kg. Google knowledge graph. http://en.wikipedia.org/wiki/Knowledge_Graph, -.

[103] MISC-midatasets. MiDataSets: multiple datasets for cBench/MiBench benchmark. http://cTuning.org/cbench, -.

[104] MISC-repro. Public wiki discussing how to enable collaborative, systematic and reproducible research and experimentation in computer engineering with an open publication model. http://c-mind.org/reproducibility, -.

[105] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[106] 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.

[107] Ryan W. Moore and Bruce R. Childers. Automatic generation of program affinity policies using machine learning. In Jhala and Bosschere [79], pages 184–203.

[108] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari 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.

[109] 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.

[110] Open benchmarking: Automated Testing & Benchmarking On An Open Platform. http://openbenchmarking.org.

[111] Oprofile: system-wide profiler for linux systems, capable of profiling all running code at low overhead. http://oprofile.sourceforge.net.

[112] David Ortiz, Nayda G Santiago, et al. Impact of source code optimizations on power consumption of embedded systems. In *Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on*, pages 133–136. IEEE, 2008.

[113] Zhelong Pan and Rudolf 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.

[114] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using graph-based program characterization for predictive modeling. In Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe, editors, *CGO*, pages 196–206. ACM, 2012.

[115] Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

[116] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, pages 65–74. ACM, 2011.

[117] Pathscale ekopath compilers. http://www.pathscale.com.

[118] PRACE: partnership for advanced computing in europe. http://www.prace-project.eu.

[119] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[120] Pranav Rajpurkar, STANFORD EDU, Toki Migimatsu, Sameep Tandon, EDU Tao Wang, and EDU Andrew Ng. Driverseat: Crowdstrapping learning tasks for autonomous driving. In *32nd International Conference on Machine Learning*, volume 37, pages 1–8, 2015.

[121] H. Roubos and M. Setnes. Compact and transparent fuzzy models and classifiers through iterative complexity reduction. *Fuzzy Systems, IEEE Transactions on*, 9(4):516–524, 2001.

[122] Ricardo Nabinger Sanchez, José Nelson Amaral, Duane Szafron, Marius Pirvu, and Mark Stoodley. Using support vector machines to learn how to compile a method. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 223–230. IEEE, 2010.

[123] Armin Shmilovici. Support vector machines. In *Data Mining and Knowledge Discovery Handbook*, pages 257–276. Springer, 2005.

[124] B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.

[125] Collective Mind: open-source plugin-based infrastructure and repository for systematic and collaborative research, experimentation and management of large scientific data. http://cTuning.org/tools/cm.

[126] ElasticSearch: open source distributed real time search and analytics. http://www.elasticsearch.org.

[127] The Standard Performance Evaluation Corporation. http://www.specbench.org.

[128] 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.

[129] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May 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.

[130] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *TACO*, 8(4):50, 2012.

[131] S. Touati, J. Worms, and S. Briais. The speedup test. In *INRIA Technical Report HAL-inria-00443839*, 2010.

[132] Georgios Tournavitis, Zheng Wang, Bjorn Franke, and Michael F.P. 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.

[133] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.

[134] Konrad Trifunovic, Albert Cohen, David Edelsohn, Li Feng, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjoedin, and Ramakrishna Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *2nd International Workshop on GCC Research Opportunities (GROW)*, 2010.

[135] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I*. Computer Science Press, Inc., New York, NY, USA, 1988.

[136] 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.

[137] R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Conference on High Performance Networking and Computing*, 1998.

[138] Heting Wu, Hailong Sun, Yili Fang, Kefan Hu, Yongqing Xie, Yangqiu Song, and Xudong Liu. Combining machine learning and crowdsourcing for better understanding commodity reviews. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[139] Kashnikov Yuriy. *A Holistic Approach To Predict Effective Compiler Optimizations Using Machine Learning*. PhD thesis, Universit´e de Versailles Saint-Quentin-en-Yvelines, France, 2012.

[140] James Y Zou, Kamalika Chaudhuri, and Adam Tauman Kalai. Crowdsourcing feature discovery via adaptively chosen comparisons. *arXiv preprint arXiv:1504.00064*, 2015.

[141] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.