HiPEAC Internship report: Machine Learning for Compilation and Architecture

Grigori Fursin, Abdul Wahid Memon, Christophe Guillon

Machine Learning for Compilation and Architecture: Myth or Reality?

Grigori Fursin, *INRIA, France*
Abdul Memon, *UVSQ, France*
Christophe Guillon, *STMicroelectronics, France*

Contact: Grigori.Fursin@cTuning.org

15 November, 2013

Abstract

Machine learning shows potential to automatically predict program optimizations for many years. However, it is still far from production use due to "black box" nature, complex and constantly changing experimental setups, very naive and limited experimental scenarios, and lack of large and diverse training sets. Our main contribution is to start cooperative formalization and validation of program optimization (auto-tuning) and machine learning making it understandable, practical, reproducible and scalable. We present a novel experimental methodology implemented as a distributed buildbot to continuously optimize and classify multiple code and dataset samples shared by the community while exposing, analyzing and solving unexpected behavior either automatically or through crowdsourcing.

We present industrial case study using 285 shared code and dataset combinations from 8 popular benchmarks and 5000 combinations of GCC compiler flags demonstrating that statistical models built using limited training sets from existing works can be totally misleading. We demonstrate how our framework can help to expose, isolate and collaboratively fix SVM model mispredictions for a surveillance camera application, while finding and sharing missing relevant features by domain specialists. 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 while producing a new realistic, representative and continuously evolving benchmark currently consisting of 79 distinct optimization classes. We agreed with our industrial partners to release presented framework and all related data for artifact evaluation and to the public at the conference under free, open source license.

Keywords: *Collaborative experimentation, program optimization, predictive modeling, performance tracking buildbot, anomaly detection, feature learning, model sharing, shared experimental pipelines, all research artifacts, continuous complexity reduction, representative benchmark, minimal feature set, experiment validation, reproducible research*

1 Introduction

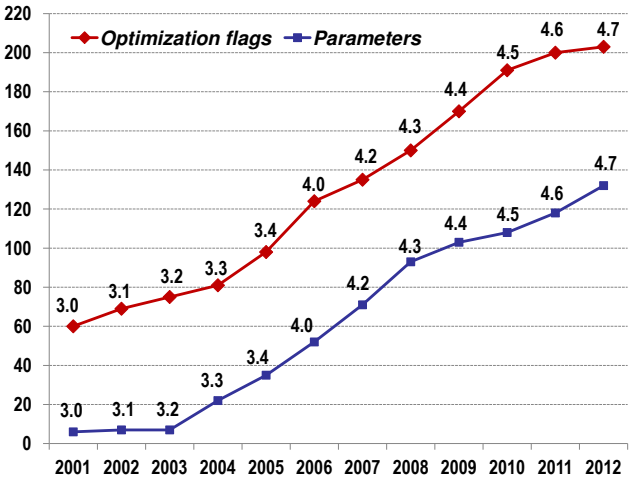

Figure 1: Rising number of optimization dimensions in GCC in the past 12 years (flags and parameters). Obtained by automatically parsing GCC manual pages, therefore small variation is possible.

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 shown in Figure 1 for more than a decade [10, 41, 34, 12, 40, 4, 23, 13, 31, 17, 44, 33, 30, 19, 37, 42, 35, 38, 11]. 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 [8, 21, 29], 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 [8].

In fact, complex, ad-hoc, and continuously changing experimental setups including rapidly evolving architectures, compilers, run-time systems, multiple and often incompatible auxiliary tools, possibly non-representative benchmarks and datasets together with excessively long training phases and rising amount of experimental data make systematic and long-term studies practically impossible. Furthermore, "black box" nature of many machine learning algorithms together with the lack of common experimental methodology and culture of sharing large, diverse and reproducible experimental sets in computer engineering makes it too tedious or sometimes even impossible to validate results of numerous publications and use them to improve compilers, applications and architectures. In the end, all these issues started rising many concerns about practicality and scalability of published machine learning based approaches for compilation and architecture in realistic production scenarios.

The main goal and contribution of the presented work is a novel, scalable and extensible optimization methodology and public framework that attempts to address all above challenges in a cooperative and coherent way while gradually unifying and validating existing ad-hoc techniques and tools. Instead of publishing 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. Now, we can easily distribute various optimization scenarios among many participants and continuously explore available optimization choices for all shared code and dataset 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, dataset 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.

As a proof of concept and to start building a community crucial for our approach, we installed developed buildbot with 2 experimental setups to validate "iterative compilation" and "machine learning" in two major companies (*omitted for blind review*) and on Android-based mobile phones of our colleagues. Within 6 months, our supporters have shared 289 code and dataset samples from major benchmarks including NAS, MiBench, SPEC2000, SPEC2006, Powerstone, UTDSP, SNU-RT, and a few real applications which were continuously optimized in terms of execution time using at least 5000 combinations of GCC compiler optimization flags currently deriving 79 distinct and pruned optimization classes. Popular SVM model [8] have been applied and optimized to separate those classes using available shared features derived from program semantic analysis and hardware counters. Analyzing mispredictions, we show that current limited experimental setups can result in completely meaningless correlations. We also present a case study in an industrial setup, where we exposed 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 dataset features to fix wrong classification. Furthermore, such white box approach helped to deliver minimal representative benchmark to an architecture verification and testing department of our industrial partner. Now, the community has an extensible toolset to continue analyzing all exposed problems and find new features to improve and optimize predictive models.

This paper is organized as follows. In Section 2, we present our novel framework that formalizes current research on auto-tuning and machine learning allowing to implement various research scenarios as shared experimental pipelines. Section 3 describes two experimental scenarios to validate compiler auto-tuning and machine learning combined with continuous and incremental complexity reduction. Section 4 presents a case study demonstrating our methodology in practice to find missing features, improve compiler optimizations and make real image processing application adaptive at run-time. Finally, we summarize related work in Section 5 and conclude in Section 6.

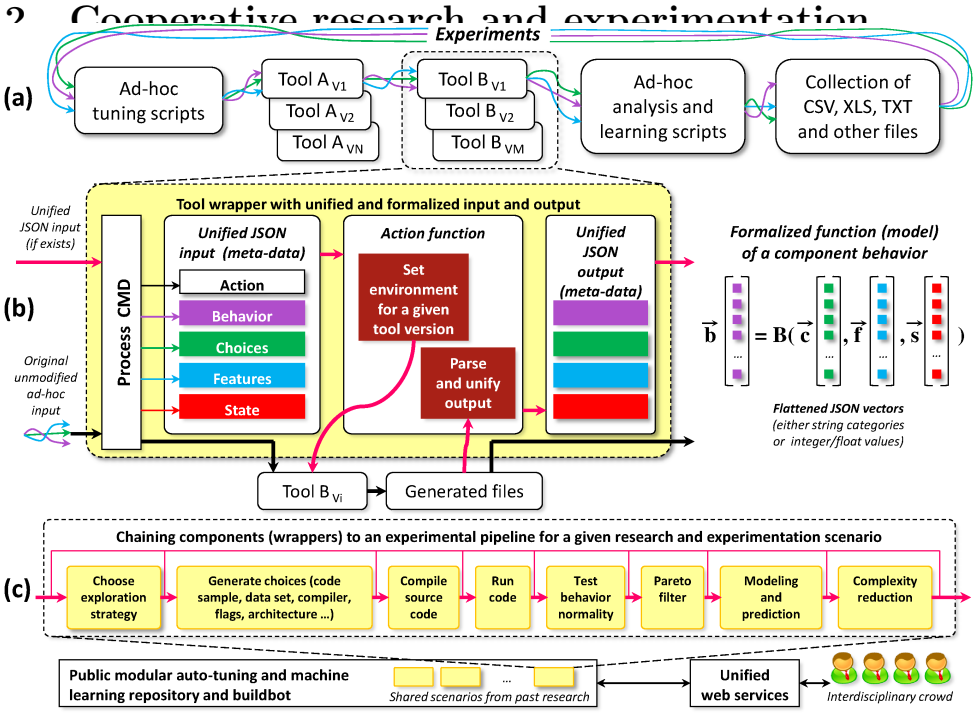

Figure 2: (a) Conceptually depicted current ad-hoc experimentation; (b) wrappers developed by the community around existing tools to gradually expose behavior (characteristics), choices, features and system state using unified JSON input and output format; (c) wrappers and modules chained together as LEGO to implement various experimentation scenarios within a public buildbot that can be collaboratively explored and improved by the community.

Our basic idea is to bring an 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 (see Journal of Statistical Software, for example), 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 that we will discuss in Section 4, and very complex and continuously evolving experimental setups with multiple hardwired 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 2a. Usually, by the end of tedious development and experimentation, new versions of compilers, libraries, OS, architectures are already available making results potentially outdated while problems possibly solved or considerably evolved. The lifespan of such often undocumented and unreleased developments is usually a duration of an MS or a PhD project.

2.1 Back to basics

In order to understand how to solve mentioned problems, we would like to first remind end-users' needs and try to formalize existing research techniques. Very simplistically, computer systems' users rarely care about underlying technology but mainly care about performing their multiple important tasks such as playing games on a console, watching videos on a mobile phone, surfing Web on a tablet, modeling a new critical vaccine on a supercomputer or predicting a new crash of financial markets using cloud services either as fast, realistic, accurate and reliable as possible or with some real-time constraints while minimizing or amortizing all associated costs including power consumption and device or service costs. Therefore, we can formalize research presented in most of the related papers as modeling of a function P that can predict most optimal design or optimization choices for a given computer system $\mathbf{c}$ based on some features (properties) of end-users' tasks and datasets $\mathbf{f}$, set of requirements $\mathbf{r}$, as well as a current state of a used computer system $\mathbf{s}$:

$$\mathbf{c} = P(\mathbf{f}, \mathbf{r}, \mathbf{s})$$

This function is naturally associated with a function B representing behavior of a user task running on a given system depending on properties, choices and a system and program state:

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

Both functions are of particular importance to hardware and software designers to be able to continuously provide and improve choices for a broad range of user tasks, datasets and requirements while trying to improve own ROI and reduce time to market. In order to find optimal choices, this function 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 as it was done in the past for the small kernels and libraries with just one or a few program transformations [45, 3]. For example, $\mathbf{b}$ is a behavior vector that can now include multiple characteristics including execution time, power consumption, accuracy, compilation time, code size, device cost, and any other important characteristic; $\mathbf{p}$ is a vector of features of a task and a system that can include semantic program features [34, 41, 4, 19], dataset features and hardware counters [13, 24], system configuration, and run-time environment parameters among many others; $\mathbf{c}$ 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 $\mathbf{s}$ represents the state of the system such as frequency, cache contentions and so on.

2.2 Agile, wrapper-based framework for cooperative experimentation

A possible revolutionary approach would be to re-design the whole software and hardware stack while exposing all characteristics and optimizations, and

continuously tuning and adapting the whole system. However it is unlikely to be quickly accepted by the community based on the past experience from some related projects that either became too complex and heavy, or too theoretical, cover a very narrow part of computer system, over specialized (for example, for supercomputers), or already last for years and far from being finished [6, 15, 32, 28, 26].

Instead, we present a practical and evolutionary approach based on above formalization of objectives of various research projects where the community gradually provides simple wrappers to the used tools including compilers, source-to-source transformers, code launchers, profilers to transparently monitor all information flow in experimental setups as shown in Figure 2b. At the same time, researchers gradually expose various characteristics of behavior **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 [2] 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 our past experience in building community-based frameworks, we noticed that researchers are not always good programmers and naturally care more about quickly prototyping their research ideas rather than drowning in complex specifications for experiments that may be even thrown away in the end. Therefore, in contrast with other frameworks, we decided to *get rid of pre-defined data specifications and rigid SQL-based databases difficult or even impossible to extend in rapidly evolving projects* in favor of gaining popularity agile methodology [5] 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 [1], 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 (as explained in the next section):

"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 being continuously extended by our partners and we will release full specification at the conference.

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 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 datasets, 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 [1] allowing fast and complex search queries.

2.3 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 li-

braries. 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 format similar to DOI:

⟨*wrapper/module name or UID*⟩:⟨*data entry UID*⟩.

3 Public research scenarios and experimental pipelines

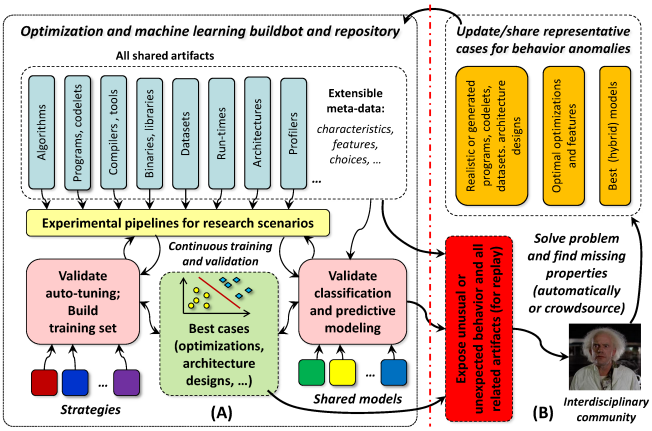

Figure 3: Summary of the presented cooperative approach and practical buildbot to collaboratively and semi-automatically learn and improve behavior of computer systems using complete public experimental pipelines including code and dataset samples, tools, models, features and all other associated artifacts shared, analyzed and improved by the interdisciplinary community.

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 [18], Pareto frontier filter to leave only optimal solutions during multi-objective optimization [27, 22] and complexity reduction and differential analysis techniques [39, 25] to continuously isolate behavior anomalies, com-

pact 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 datasets samples and collaboratively explore large optimization spaces using our public buildbot while making use of machine learning statistically meaningful as conceptually summarized in Figure 3.

However, our approach also requires radical change in mentality of researchers when defining experiments that can be collaboratively explored thorough 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 to expose it to an interdisciplinary community that can help to 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 [21, 29].

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.

3.1 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 (see Figure 1). However, in contrast with 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 monitored characteristics in a provided computer resource such as code and dataset 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 2.2, 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.

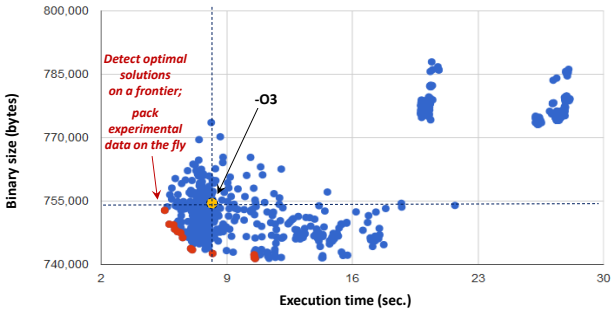

Figure 4: Variation in execution time vs code size when crowdsourcing optimization of an image corner detection application with a fixed dataset on Samsung Galaxy Series mobile phone with ARMv6 830MHz processor when randomly selecting compiler flags for Sourcery GCC 4.7.2. Yellow point represents -O3 and red circles show Pareto frontier. *This data will be available for validation at the conference.*

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, datasets, 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 public 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 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 4 shows 2D visualization of the multi-dimensional optimization and characteristic space using Google Web Services. Note, that before exploring multiple optimization choices on an available resource 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 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 4. Now, a user can easily select optimal cases community depending on the further application usage, i.e. 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.

3.2 Universal complexity reduction, problem isolation and experimental data packing

Though effective, crowdsourcing has a downside - rising amount of data to exchange, store and analyze (also known as a big data problem). Formalization of auto-tuning combined with on-line filters (based on active learning) allows us to elegantly and universally solve this problem at least in our domain by continuously pruning those explored points or dimensions in choices that have description key *'can_be_omitted'* set to *'yes'* and do not degrade any observed past characteristic within some allowed threshold when removed. Therefore, we can now simply skip all blue points in Figure 4 leaving only optimal red solution on the frontier or unexpected behavior for web-based data exchange and storage during online tuning while *effectively packing empirical results by several orders of magnitude*.

The same problem also relates to compiler flag selection: very often optimal random combinations of optimization flags together with global optimization levels such as -O3, -Os and -fast include many "useless" flags (noise) that do not have any effect on the code during compilation. Our unification of choices allows to use standard complexity reduction plugin for this problem too while incrementally, continuously and randomly switching off optimizations using -f**no**-optimization_flag that do not degrade any of the observed characteristics in contrast with just removing them as in some other works [19] since global optimization level flag may still turn them on. Such pruning is often overlooked by the community but is essential to improve machine learning as will be shown further.

3.3 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 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 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

-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 -fguess-branch-probability -fmove-loop-invariants -fsched-pressure -fschedule-insns ***-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 -fgcse -fivopts -fmove-loop-invariants -ftree-dominator-opts -ftree-loop-optimize -funroll-all-loops ***-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 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. *We continue running our buildbot and will release updated list of optimization clusters for other compilers and architectures to the PLDI Artifact Evaluation Committee and at the conference.*

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 from -O3 and continuously cluster all available benchmarks in terms of those optimizations . The natural idea is that 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 an internal optimization heuristic as it is not possibly to simply add these optimization flags to -O3 to improve all 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 combinations 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 optimization combinations or at least 5000 random combinations of flags during 5 months. Whenever a new top combination of optimizations was found outside the pool, we applied it to all shared programs to perform online clustering while simply 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 dataset samples. Table 1 present some of the top performing combinations of flags pruned by the universal complexity reduction plugin as described in the previous section.

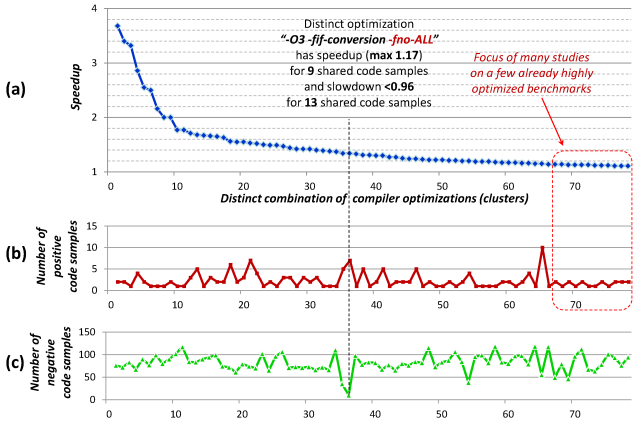

Figure 5: (a) 79 distinct combinations of optimizations (optimization clusters) covering all 285 shared code and dataset samples on Intel E5520, GCC 4.6.3 and at least 5000 random combinations of flags together with maximum speedup achieved within each optimization cluster; (b) number of benchmarks with 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 5 shows maximum speedups achieved for each optimization cluster across all benchmarks together with the number of benchmarks which achieve 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 9 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

Number of code and dataset samples	Prediction accuracy using optimized SVM
12 (from prior work) [19]	87%
285 from current work	56%

Table 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).

other problems in production compilers thus considerably enhancing existing bug buildbots. Furthermore, it helps to automatically systematize and prune large collections of benchmarks and datasets leaving only representative ones for a given research problem (such as leaving only one code and related dataset 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 [19]. 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 2 summarizes results of our modeling. When using just a few benchmarks, prediction accuracy is quite high and supports findings from other papers including [19]. 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 keeping at least not worse prediction accuracy. Naturally, it can also be done using statistical techniques such as ANOVA or PCA [13], but since we would like to isolate possible problem, we need precise analysis. Our pruning left only one semantic feature from MILEPOST GCC (ft29) that counts number of basic blocks where the number of phi-nodes is greater than 3. Visualization at Figure 6 helps us to derive a decision that ft29

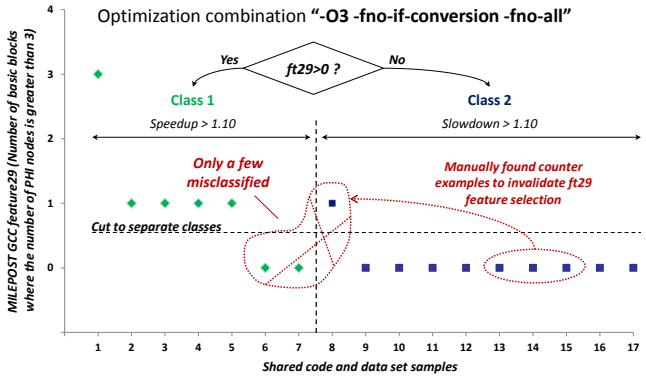

Figure 6: Automatic detection of the relevant feature(s) to predict optimization cluster "-O3 -fno-if-conversion -fno-ALL" using complexity reduction. However, we manually converted several code samples to provide counter examples that invalidated this feature and showed that using small training sets in many current studies can be totally misleading.

is more than 0 can effectively separate two classes with only 3 mispredictions out of 17.

Most of the papers will conclude at this step that relevant feature is found and it is possible to use machine learning to predict optimizations. However, in 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 converting 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 passes the threshold *ft29* 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 bug buildbot where samples causing compiler crashes are continuously collected and analyzed.

Does it mean that machine learning for compilation and architecture is a hoax? No, it just means that our community is often using it 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 hundreds of features and 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 when much more benchmarks available. 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!

4 Crowdsourcing feature learning and model improvement

Most of the papers on auto-tuning and machine learning fight variation in execution time [16, 43, 36] or mispredictions to be able to show only positive results. However, in our approach, *unexpected behavior is of critical importance to find and explain missing features in a system.* Therefore, we have developed a new buildbot module using Shapiro-Wilk normality test from R that can be chained to any experimental pipeline to test any monitored characteristic in a behavior vector **b** for normality. Most of the reported experiments have been executed more than 30 times while passing a normality test with variation less than 3%. Otherwise, we record an experiment in a reproducible wayas "suspicious" for further analysis.

For example, one of the code samples from the previous section (image B&W threshold filter) was wrongly classified because most of the time it belonged to the optimization cluster *-O3 -fif-conversion* while occasionally moving to an opposite class. We managed to identified two distinct datasets that separated the classes. The visual analysis suggests that when image is mostly white, "if conversion" transformation is beneficial while when image is mostly black, "if conversion" considerably degrades performance by 17.3%. As usual, we decided to apply universal complexity reduction here to isolate the cause of this performance regression by iteratively removing instructions one by one from the source code until regression disappeared. Note, that semantics of the code is now changing but it does not matter as long as code is not crashing and we can detect instruction causing performance degradation. This helped us to identify "suspicious instruction" *(temp1 ⟩ T) ? 255 : 0.* "If conversion" added several predicated statements that may degrade performance if additional branches are rarely taken while adding a few additional cycles to check branch condition. However, this is a run-time dependency which no semantic compiler feature can

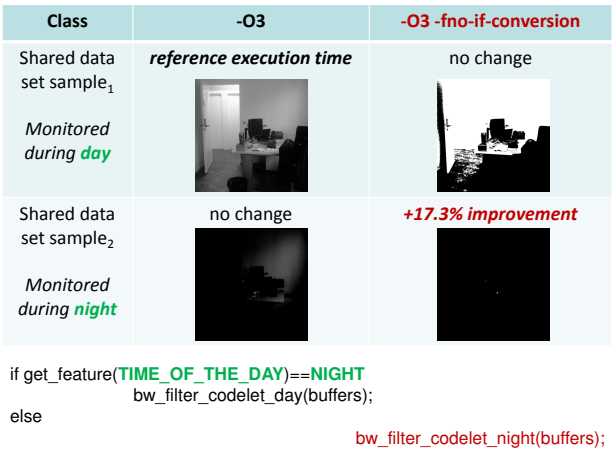

Figure 7: Manually detecting missing feature related to the time of the day which does not exist in the system and thus impossible to derive automatically in available machine learning approaches. It was successfully used to make adaptive application performing well across all available datasets.

capture. However, by preserving and analyzing the whole experimental setup, we noticed that some images where taken during the day and some during the night as shown in Figure 7 helping us to find new relevant feature *"time of the day"* out of the system that can effectively separate 2 classes. We added this feature to our experimental pipeline through our universal feature vector **f** and validated our idea by passing this feature to the application through environment variable *TIME_OF_THE_DAY*. Now, we can clone threshold filter function and applying 2 different optimizations during compilation while adding a decision tree to effectively separates classes at run-time thus delivering an adaptive application to a customer and sharing *missing feature that did not exist in a system* with the community in an experimental pipeline.

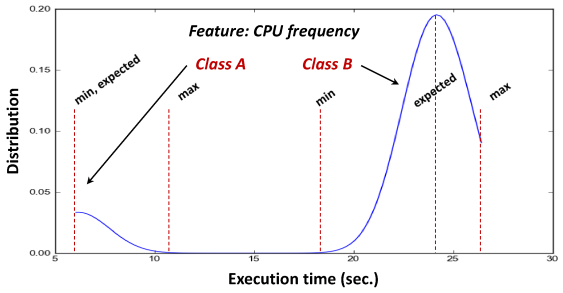

Figure 8: Unexpected behavior helped to identify and share missing feature.

At the same time, when analyzing multiple executions of image corner detection benchmark on a smart phone shown in Figure 4, 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 8. Simple analysis showed

that our phone was often in the low power state at the beginning of experiments and then gradually switched to the high-frequency state (4x difference in frequency). Though obvious, this information allowed us to add CPU frequency to the pipeline and universal feature, state and choice vectors **f**, **s**, and **c** 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 example highlights yet another fundamental problem with current machine learning techniques for compilation and architecture that attempt to automatically correlate some existing features with optimizations - related feature may simply not be even available in the system and will likely require specialized knowledge even though we expect that it may be formalized later using deep learning techniques when enough features or tools to extract features will be shared by the community. It also supports our idea to move away from "black box" machine learning approaches at least at this stage and possibly build specialized hybrid models combining manually found decision trees, and spare computer resources to continuously optimize, cluster, prune and model behavior of computer systems while exposing unusual behavior to a community in a reproducible way!

5 Related work

Nowadays, machine learning has become a buzz word when nearly every conference, journal and workshop in our field includes multiple papers on black box program and architecture automatic tuning and predictive modeling [10, 41, 34, 12, 40, 4, 23, 13, 31, 17, 44, 33, 30, 19, 37, 42, 35, 38, 11]. At the same time, there are very few papers that come with shared tools and data to reproduce results and validate scalability and statistical meaningfulness of techniques. Interestingly, we contacted authors of six referenced papers and only two were able to provide related tools or scripts with one interesting reply that *we should just trust their technique because it was published in a top conference*! Besides obvious reasons of academic competition or worries of mistakes, this is largely due to an existing vicious circle where initiatives to develop common tools and repositories widely available to the research and teaching community to systematize experimentation, share research artifacts (datasets, tools, benchmarks, statistics, models), validate past experimental results and techniques are practically not funded or rewarded academically.

The most close work to ours is MILEPOST GCC [19] that is accompanied by a public learning compiler and database with performance results. However, it is very GCC centric, can not be extended to other experimental setups, have a very few available ad-hoc features, and focuses only on a few positive outcomes (predictions). We demonstrate that their results can be misleading and provide methodology to cooperatively improve them in industrial setup. Another related work is [14] where GCC compiler flags are continuously tuned using a data center, however like most other papers it uses black box auto-tuning with only several programs while focusing on a few speedups and without any released tools. Finally, authors in [30] 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, we demonstrate in Section 4 that very often related features are not even available in a system.

Therefore, to the best of our knowledge, we provide the first simple and practical methodology and public framework to unify, formalize and connect together available ad-hoc techniques and tools for auto-tuning and machine learning using recent advances in agile methodologies, web and crowdsourcing technology, and schema-free repositories. We hope that recent initiatives particularly at OOPSLA and this conference to evaluate research artifacts for the accepted papers can be supported technically by the presented approach to *make the quality and reproducibility of experimental results as important as publications themselves.*

6 Conclusions and Future Work

The fundamental question that we started addressing in this article is *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 recent advances in web technology, JSON-based schema free databases, agile methodology and 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 reimplementing ad-hoc experimental setups from numerous publications. It also helps researchers to quickly prototype their ideas in days rather than months just like assembling LEGO by reusing and customizing shared experimental setups and data while focusing all their effort and creativity on either solving existing problems while reusing, improving and optimizing shared predictive models and finding missing features, or developing truly novel approaches.

Furthermore, we expect that our methodology will eventually help our community to switch focus from publishing only a few positive and possibly misleading or even wrong results to sharing all data including negative results or to validate past research techniques which is practically impossible to publish. It can also fit well recent initiatives on reproducible research and artifact evaluation which can in turn help to restore the attractiveness of computer engineering particularly for new students making it a more systematic, rigorous and reproducible discipline.

The success of this approach depends fully on the active community involvement. Therefore, in the past year, we managed to validate it in two major companies and with several academic partners. All partners have been either using common experimental setups while customizing existing wrappers for their own tools and exposing tuning dimensions, characteristics and features, or developing their own wrappers and learning components compatible with our buildbot. In spite of our framework being recently officially licensed by our industrial partner, we agreed to release it together with a public repository, experimental setups for compiler multi-objective auto-tuning and machine learning for GCC, LLVM, Open64, Rose, and ICC, and all related research artifacts including code

and dataset samples under the free, open-source license for further validation and improvement by the community.

Our future and ambitious goal is to use presented approach and framework to bring a large interdisciplinary community together to systematically optimize and model behavior of various existing computer systems. Our formalization allows to apply top-down learning and modeling approach already effectively used in many other sciences while focusing first on coarse-grain behavior and optimizations such as global optimizations and later move to finer-grain level gradually covering analysis, optimization and adaptation of the whole computer system and making it compatible with our approach including fine-grain program optimizations using pragmas [20], events and plugins in GCC [19]; source-to-source polyhedral transformation tools [9]; adaptive scheduling [33]; just-in-time compilation for Android Dalvik or Oracle JDK, algorithm-level tuning [7] and many others.

At the same time, mathematical formalization allows us to take advantage of recent advances in deep learning techniques when unsupervised feature learning will become fast and practical enough [21, 29]. Furthermore, it allows us to apply standard complexity reduction techniques to find related features, improve and share most accurate predictive models, and build minimal representative and community-driven benchmarks covering various research problems. We expect that community will eventually create a large and diverse training set similar to other sciences to help researchers avoid common statistical pitfalls while practically helping compiler engineers, application developers and system designers to improve tools, architectures, and programs unveiling machine learning as a useful "white-box" support tool rather than "black-box" panacea.

7 Acknowledgments

Grigori Fursin was funded by EU HiPEAC postdoctoral fellowship (2005-2006) and EU MILEPOST project (2007-2010) where he started developing methodology, repository and infrastructure to crowdsource auto-tuning and machine-learning. Abdul Memon was funded by EU HiPEAC industrial internship working for 6 months at STMicroelectronics (Grenoble, France) and by Higher Education Commission of Pakistan. We would like to thank Antoine Moynault and Francois De-Ferriere for interesting discussions, experiments and support during Abdul's internship. We are thankful to Francois Bodin and CAPS Entreprise for sharing codelets from the MILEPOST project. We are very grateful to cTuning and HiPEAC communities for many interesting discussions and feedback during development of the repository and infrastructure presented in this paper. We would like to thank Yuriy Kashnikov for sharing data about number of GCC optimizations per year. We would like to thank Anton Lokhmotov from ARM (Cambridge, UK) for additional evaluation of the framework. Finally, we would like to thank GRID5000 project and community for providing an access to powerful computational resources.

References

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

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

[3] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. Gurd, J. Hoogerbrugge, P. Hu, W. Jalby, P.M.W. Knijnenburg, M.F.P O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E.A. Stöhr, M. Verhoeven, and H.A.G. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1351–1356, 1997.

[4] 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.

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

[6] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone. In *Technical Report 1997-016. Digital Equipment Corporation Systems Research Center, Palo Alto, CA*, 1997.

[7] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

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

[9] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[10] 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)*, 1997.

[11] Rosario Cammarota, Alexandru Nicolau, Alexander V. Veidenbaum, Arun Kejariwal, Debora Donato, and Mukund Madhugiri. On the determination of inlining vectors for program optimization. In *CC*, pages 164–183, 2013.

[12] 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.

[13] 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.

[14] 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.

[15] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[16] Charlie Curtsinger and Emery D. Berger. Stabilizer: statistically sound performance evaluation. *SIGARCH Comput. Archit. News*, 41(1):219–228, March 2013.

[17] 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.

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

[19] 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. OBoyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[20] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11, 2009.

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

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

[23] 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.

[24] 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.

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

[26] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[27] 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.

[28] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[29] 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.

[30] 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, 2009.

[31] 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.

[32] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism*, volume 6, 2004.

[33] 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.

[34] 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.

[35] Ryan W. Moore and Bruce R. Childers. Automatic generation of program affinity policies using machine learning. In *CC*, pages 184–203, 2013.

[36] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGARCH Comput. Archit. News*, 37(1):265–276, March 2009.

[37] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using graph-based program characterization for predictive modeling. In *CGO*, pages 196–206, 2012.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

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

[43] Sid Ahmed Ali Touati, Julien Worms, and Sébastien Briais. The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426, 2013.

[44] Georgios Tournavitis, Zheng Wang, Björn 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 *PLDI*, pages 177–187, 2009.

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