Collective Mind: towards practical and collaborative auto-tuning

Grigori Fursin *(Inria and University of Paris-Sud, Orsay, France; contact: grigori.fursin@inria.fr)*
Renato Miceli *(University of Rennes 1, France and ICHEC, Dublin, Ireland)*
Anton Lokhmotov *(ARM, Cambridge, United Kingdom)*
Michael Gerndt *(Technical University of Munich, Germany)*
Marc Baboulin *(Inria and University of Paris-Sud, Orsay, France)*
Allen D. Malony *(University of Oregon, Eugene, OR, USA)*
Zbigniew Chamski *(Infrasoft IT Solutions, Plock, Poland)*
Diego Novillo *(Google Inc., Toronto, Canada)*
Davide Del Vento *(National Center for Atmospheric Research, Boulder, CO, USA)*

***Abstract*—Empirical auto-tuning and machine learning techniques have been showing high potential to improve execution time, power consumption, code size, reliability and other important metrics of various applications for more than two decades. However, they are still far from widespread production use due to lack of native support for auto-tuning in an ever changing and complex software and hardware stack, large and multi-dimensional optimization spaces, excessively long exploration times, and lack of unified mechanisms for preserving and sharing of optimization knowledge and research material.**

We present a possible collaborative approach to solve above problems using Collective Mind knowledge management system. In contrast with previous cTuning framework, this modular infrastructure allows to preserve and share through the Internet the whole auto-tuning setups with all related artifacts and their software and hardware dependencies besides just performance data. It also allows to gradually structure, systematize and describe all available research material including tools, benchmarks, data sets, search strategies and machine learning models. Researchers can take advantage of shared components and data with extensible meta-description to quickly and collaboratively validate and improve existing auto-tuning and benchmarking techniques or prototype new ones. The community can now gradually learn and improve complex behavior of all existing computer systems while exposing behavior anomalies or model mispredictions to an interdisciplinary community in a reproducible way for further analysis. We present several practical, collaborative and model-driven auto-tuning scenarios. We also decided to release all material at *c-mind.org/repo* to set up an example for a collaborative and reproducible research as well as our new publication model in computer engineering where experimental results are continuously shared and validated by the community.

Keywords*—*high performance computing, systematic auto-tuning, systematic benchmarking, big data driven optimization, modeling of computer behavior, performance prediction, collaborative knowledge management, public repository of knowledge, NoSQL repository, code and data sharing, specification sharing, collaborative experimentation, machine learning, data mining, multi-objective optimization, model driven optimization, agile development, plugin-based tuning, performance regression buildbot, open access publication model, reproducible research

I. Introduction and related work

Computer systems' users are always eager to have faster, smaller, cheaper, more reliable and power efficient computer systems either to improve their every day tasks and quality of life or to continue innovation in science and technology. However, designing and optimizing such systems is becoming excessively time consuming, costly and error prone due to an enormous number of available design and optimization choices and complex interactions between all software and hardware components. Furthermore, multiple characteristics have to be carefully balanced including execution time, code size, compilation time, power consumption and reliability using a growing number of incompatible tools and techniques with many ad-hoc, intuition based heuristics.

At the same time, development methodology for computer systems has hardly changed in the past decades: hardware is first designed and then the compiler is tuned for the new architecture using some ad-hoc benchmarks and heuristics. As a result, nearly peak performance of the new systems is often achieved only for a few previously optimized and not necessarily representative benchmarks while leaving most of the real user applications severely underperforming. Therefore, users are often forced to resort to a tedious and often non-systematic optimization of their programs for each new architecture. This, in turn, leads to an enormous waste of time, expensive computing resources and energy, dramatically increases development costs and time-to-market for new products and slows down innovation [1], [2], [3], [4].

Various off-line and on-line auto-tuning techniques together with run-time adaptation and split compilation have been introduced during the past two decades to address some of the above problems and help users automatically improve performance, power consumption and other characteristics of their applications. These approaches treat rapidly evolving computer system as a black box and explore program and architecture design and optimization spaces empirically [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20].

Since empirical auto-tuning was conceptually simple and did not require deep user knowledge about programs and computer systems, it quickly gained popularity. At the

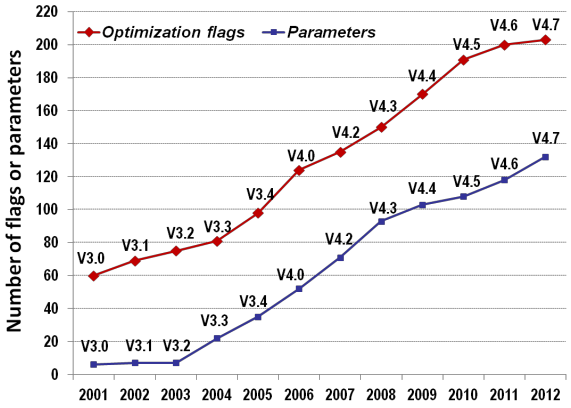

Fig. 1. Rising number of optimization dimensions in GCC in the past 12 years (boolean or parametric flags). Obtained by automatically parsing GCC manual pages, therefore small variation is possible (script was kindly shared by Yuriy Kashnikov).

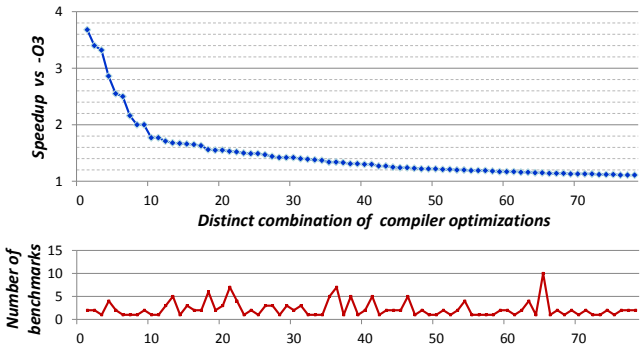

Fig. 2. Number of distinct combinations of compiler optimizations for GCC 4.7.2 with a maximum achievable execution time speedup over -O3 optimization level on Intel Xeon E5520 platform across 285 shared Collective Mind benchmarks after 5000 random iterations (top graph) together with a number of benchmarks where these combinations achieve more than 10% speedup (bottom graph).

same time, users immediately faced a fundamental problem: a continuously growing number of available design and optimization choices makes it impossible to exhaustively explore the whole optimization space. For example, Figure 1 shows a continuously rising number of available boolean and parametric optimizations available in a popular, production, open-source compiler GCC used in practically all Linux and Android based systems. Furthermore, there is no more single combination of flags such as -O3 or -Ofast that could deliver the best execution time across all user programs. Figure 2 demonstrates 79 distinct combinations of optimizations for GCC 4.7.2 that improve execution time across 285 benchmarks with just one data set over -O3 on Intel Xeon E5520 based platform after 5000 explored solutions using traditional iterative compilation [21] (random selection of compiler optimization flags and parameters).

Optimization space explodes even further when considering

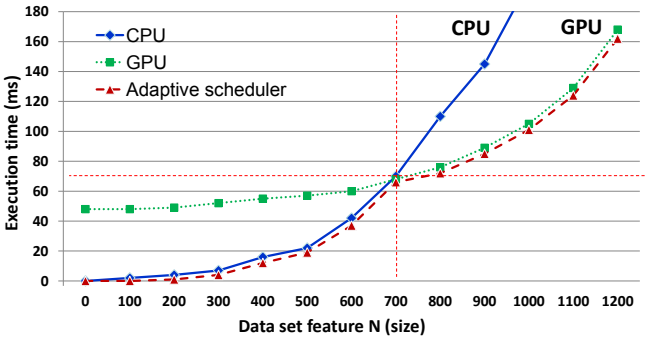

Fig. 3. Execution time of a matrix-matrix multiply kernel when executed on CPU (Intel E6600) and on GPU (NVIDIA 8600 GTS) depending on the size N of square matrix as a motivation for online tuning and adaptive scheduling on heterogeneous architectures [22].

heterogeneous architectures, multiple data sets and fine grain program transformations and parameters including tiling, unrolling, inlining, padding, prefetching, number of threads, processor frequency and MPI communication [11], [23], [24], [21], [25], [26]. For example, Figure 3 shows execution time of a matrix-matrix multiplication kernel for square matrices on CPU (Intel E6600) and GPU (NVIDIA 8600 GTS), depending on their size. It motivates the need for adaptive scheduling since it may be beneficial either to run kernel on CPU or GPU depending on data set parameters (to amortize the cost of data transfers to GPU). However, as we show in [22], [27], [28], the final decision tree is architecture and kernel dependent and requires both off-line kernel cloning and some on-line, automatic and ad-hoc modeling of application behavior.

Machine learning techniques (predictive modeling and classification) have been gradually introduced during the past decade as an attempt to address the above problems [29], [30], [31], [32], [33], [34], [35], [36], [22], [37]. These techniques can help speed up program and architecture analysis, optimization and co-design by narrowing down regions in large optimization spaces with the most likely highest speedup. They usually use prior training similar to Figure 2 in case of compiler tuning and predict optimizations for previously unseen programs based on some code, data set and system features.

During the MILEPOST project in 2006-2009, we made the first practical attempt to move auto-tuning and machine learning to production compilers including GCC by combining a plugin-based compiler framework [38] and a public repository of experimental results (cTuning.org). This approach allowed to substitute and automatically learn default compiler optimization heuristics by crowdsourcing auto-tuning (processing a large amount of performance statistics collected from many users to classify application and build predictive models) [39], [21], [40]. However, this project exposed even more fundamental challenges including:

- Lack of common, large and diverse benchmarks and data sets needed to build statistically meaningful predictive models;

- Lack of common experimental methodology and unified ways to preserve, systematize and share our growing optimization knowledge and research material including benchmarks, data sets, tools, tuning plugins, predictive models and optimization results;
- Problem with continuously changing, "black box" and complex software and hardware stack with many hardwired and hidden optimization choices and heuristics not well suited for auto-tuning and machine learning;
- Difficulty to reproduce performance results from the cTuning.org database submitted by users due to a lack of full software and hardware dependencies;
- Difficulty to validate related auto-tuning and machine learning techniques from existing publications due to a lack of culture of sharing research artifacts with full experiment specifications along with publications in computer engineering.

As a result, we spent a considerable amount of our "research" time on re-engineering existing tools or developing new ones to support auto-tuning and learning. At the same time, we were trying to somehow assemble large and diverse experimental sets to make our research and experimentation on machine learning and data mining statistically meaningful. We spent even more time when struggling to reproduce existing machine learning-based optimization techniques from numerous publications.

Worse, when we were ready to deliver auto-tuning solutions at the end of such tedious developments, experimentation and validation, we were already receiving new versions of compilers, third-party tools, libraries, operating systems and architectures. As a consequence, our developments and results were already potentially outdated even before being released while optimization problems considerably evolved.

We believe that these are major reasons why so many promising research techniques, tools and data sets for auto-tuning and machine learning in computer engineering have a life span of a PhD project, grant funding or publication preparation, and often vanish shortly after. Furthermore, we witness diminishing attractiveness of computer engineering often seen by students as "hacking" rather than systematic science. Some of the recent long-term research visions acknowledge these problems for computer engineering and many research groups search for "holy grail" auto-tuning solutions but no widely adopted solution has been found yet [2], [3].

In this paper, we describe the first, to our knowledge, alternative, orthogonal, community-based and big-data driven approach to address above problems. It may help make auto-tuning a mainstream technology based on our practical experience in the MILEPOST, cTuning and Auto-tune projects, industrial usage of our frameworks and community feedback. Our main contribution is a collaborative knowledge management framework for computer engineering called Collective Mind (or cM for short) that brings interdisciplinary researchers and developers together to organize, systematize, share and validate already available or new tools, techniques and data in a unified format with gradually exposed actions and meta-information required for auto-tuning and learning (optimization choices, features and tuning characteristics).

Our approach should allow to collaboratively prototype, evaluate and improve various auto-tuning techniques while reusing all shared artifacts just like LEGO^{TM}pieces and applying machine learning and data mining techniques to find meaningful relations between all shared material. It can also help crowdsource long tuning and learning process including classification and model building among many participants while using Collective Mind as a performance tracking buildbot. At the same time, any unexpected program behavior or model mispredictions can now be exposed to the community through unified cM web-services for collaborative analysis, explanation and solving. This, in turn, enables reproducibility of experimental results naturally and as a side effect rather than being enforced - interdisciplinary community needs to gradually find and add missing software and hardware dependencies to the Collective Mind (fixing processor frequency, pinning code to specific cores to avoid contentions) or improve analysis and predictive models (statistical normality tests for multiple experiments) whenever abnormal behavior is detected.

We hope that our approach will eventually help the community collaboratively evaluate and derive the most effective auto-tuning and learning strategies. It should also eventually help the community collaboratively learn complex behavior of all existing computer systems using top-down methodology originating from physics. At the same time, continuously collected and systematized knowledge ("big data") should help us make more scientifically motivated advice about how to improve design and optimization of the future computer systems (particularly on our way towards extreme scale computing). Finally, we believe that it can naturally make computer engineering a systematic science while supporting Vinton G. Cerf's recent vision [41].

This paper is organized as follows: the current section provides motivation for our approach and related work. It is followed by Section II presenting possible solution to collaboratively systematize and unify our knowledge about program optimization and auto-tuning using public Collective Mind framework and repository. Section III presents mathematical formalization of auto-tuning techniques. Section IV demonstrates how our collaborative approach can be combined with several existing plugin-based auto-tuning infrastructures including MILEPOST GCC [40], OpenME [42] and Periscope Tuning Framework (PTF) [26] to start systematizing and making practical various auto-tuning scenarios from our industrial partners including continuous benchmarking and comparison of compilers, validation of new hardware designs, crowdsourcing of program optimization using commodity mobile phones and tablets, automatic modeling of application behavior, model driven optimization and adaptive scheduling. It is followed by a section on reproducibility of experimental results in our approach together with a new publication model proposal where all research material is continuously shared and validated by the community. The last section includes conclusions and future work directions.

II. Cleaning up research and experimental mess

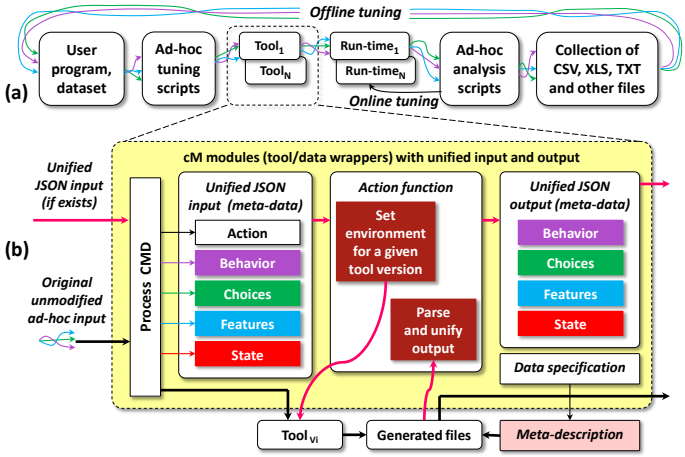

Fig. 4. Converting (a) continuously evolving, ad-hoc, hardwired and difficult to maintain experimental setups to (b) interconnected cM modules (tool wrappers) with unified, dictionary-based inputs and outputs, data meta-description, and gradually exposed characteristics, tuning choices, features and a system state.

Based on our long experience with auto-tuning and machine learning in both academia and industry, we now strongly believe that the missing piece of the puzzle to make these techniques practical is to enable sharing systematization, and reuse of all available optimization knowledge and experience from the community. However, our first attempt to crowdsource auto-tuning and machine learning using cTuning plugin-based framework and MySQL-based repository (cTuning) [40], [43] suffered from many orthogonal engineering issues. For example, we had to spend considerable effort to develop and continuously update ad-hoc research and experimental scenarios using many hardwired scripts and tools while being able to expose only a few dimensions, monitor a few characteristics and extract a few features. At the same time, we struggled with collection, processing and storing of a growing amount of experimental data in many different formats as conceptually shown in Figure 4a. Furthermore, adding a new version of a compiler or comparing multiple compilers at the same time required complex manipulations with numerous environment variables.

Eventually, all these problems motivated the development of a modular Collective Mind framework and NoSQL heterogeneous repository [42], [44] to unify and preserve the whole experimental setups with all related artifacts and dependencies. First of all, to avoid invoking ad-hoc tools directly, we introduced cM modules which serve as wrappers around them to be able to transparently set up all necessary environment variables and validate all software and hardware dependencies before eventually calling these tools. Such an approach allows easy co-existence of multiple versions of tools and libraries while protecting experimental setups from continuous changes in the system. Furthermore, cM modules can now transparently monitor and unify all information flow in the system. For example, we currently monitor tools' command line together with their input and output files to expose measured characteristics (behavior of computer systems), optimization and tuning choices, program, data set and architecture features, and a system state used in all our existing auto-tuning and machine learning scenarios as conceptually shown in Figure 4b.

Since researchers are often eager to quickly prototype their research ideas rather than sink in low-language implementations, complex APIs and data structures that may change over time, we decided to use a researcher friendly and portable Python language as the main language in Collective Mind (though we also provide possibility to use any other language for writing modules through an OpenME interface described later in this paper in Section IV). Therefore, it is possible to run minimal cM on practically any Linux and Windows computer supporting Python. An additional benefit of using Python is a growing collection of useful packages for data management, mining and machine learning.

We also decided to switch from traditional TXT, CSV and XML formats used in the first cTuning framework to a schema-free JSON data format [45] for all module inputs, outputs and meta-description. JSON is a popular, human readable and open standard format that represent data objects as attribute – value pairs. It is now backed up by many companies, supported by most of the recent languages and powerful search engines [46], and can be immediately used for web services and P2P communication during collaborative research and experimentation. Only when the format of data becomes stable or a research technique is validated, the community can provide data specification as will be described later in this paper.

At the same time, we noticed that we can apply exactly the same concept of cM modules to systematize and describe any research and development material (code and data) while making sure that it can be easily found, reused and exposed to the Web. Researchers and developers can now categorize any collections of their files and directories by assigning an existing or adding a new cM module and moving their material to a new directory with a unique ID (UID) and an optional alias. Thus we can now abstract an access to highly heterogeneous and evolving material by gradually adding possible data actions and meta-description required for user's research and development. For example, all cM modules have common actions to manage their data in a unified way similar to any repository such as *add, list, view, copy, move and search*. In addition, module *code.source* abstracts access to programs and has an individual action *build* to compile a given program. In fact, all current cM functionality is implemented as interconnected modules including *kernel, core and repo* that provide main low-level cM functions documented at *c-mind.org/doxygen*.

In contrast with using SQL-based databases, our approach can help systematize, preserve and describe any heterogeneous code and data on any native file system without any need for specialized databases, pre-defined data schema and complex table restructuring as conceptually shown in Figure 5. Since cM modules also have their own UOA (UID or alias), it is now possible to easily reference and find any local user material similar to DOI by a unified Collective ID (CID) of

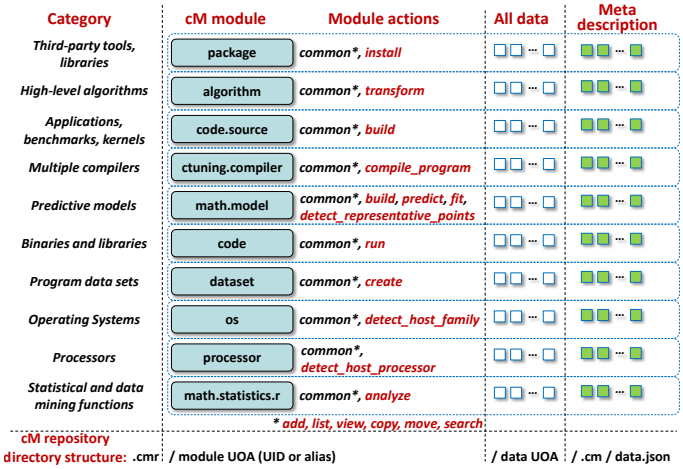

Fig. 5. Gradually categorizing all available user artifacts using cM modules while making them searchable through meta-description and reusable through unified cM module actions. All material from this paper is shared through Collective Mind online live repository at *c-mind.org/browse* and *c-mind.org/github-code-source*.

the following format:

⟨*cM module UOA*⟩:⟨*cM data UOA*⟩.

In addition, cM provides an option to transparently index meta-description of all artifacts using a third-party, open-source and JSON-based ElasticSearch framework implemented on Hadoop engine [46] to enable fast and powerful search queries for such a schema-free repository. Note that other similar NoSQL databases including MongoDB and CouchDB or even SQL-based repositories can be easily connected to cM to cache data or speed up queries, if necessary.

Any cM module with a given action can be executed in a unified way using JSON format as both input and output either through cM command line front-end

cm ⟨*module UOA*⟩ ⟨*action*⟩ *@input.json*

or using one Python function from a cM kernel module

r=cm_kernel.access({'cm_run_module_uoa':⟨*module UOA*⟩*, 'cm_action':*⟨*action*⟩*, action parameters})*

or as a web service when running internal cM web server (also implemented as cM *web.server* module) using the following URL

http://localhost:3333?cm_web_module_uoa=⟨*module UOA*⟩ *&cm_web_action=*⟨*action*⟩ *... .*

For example, a user can list all available programs in the system using *cm code.source list* and then compile a given program using

cm code.source build
work_dir_data_uoa=benchmark-cbench-security-blowfish
build_target_os_uoa=windows-generic-64 .

If some parameters or dependencies are missing, the cM module should be implemented as such to inform users about how to fix these problems.

In order to simplify validation and reuse of shared experimental setups, we provide an option to keep tools inside a cM repository also together with a unified installation mechanism that resolves all software dependencies. Such packages including third-party tools and libraries can now be installed into a different entry in a cM repository with a unique IDs abstracted by cM *code* module. At the same time, OS-dependent script is automatically generated for each version of a package to set up appropriate environment including all paths. This script is automatically called before executing a given tool version inside an associated cM module as shown in Figure 4.

In spite of its relative simplicity, the Collective Mind approach helped us to gradually clean up and systematize our material that can now be easily searched, shared, reused or exposed to the web. It also helps substitute all ad-hoc and hardwired experimental setups with interconnected and unified modules and data that can be protected from continuous changes in computer systems and easily shared among workgroups. Users only need to categorize new material, move related files to a special directory of format *.cmr/*⟨*module UOA*⟩*/*⟨*data UOA*⟩ (where .cmr is an acronym for Collective Mind Repository) to be automatically discovered and indexed by cM, and provide some meta-information in JSON format depending on research scenarios.

In contrast with public web-based sharing services, we provide an open-source, technology-neutral, agile, customizable, and portable knowledge management system which allows both private and public systematization of research and experimentation. To initiate and demonstrate gradual and collaborative systematization of a research material for auto-tuning and machine learning, we decided to release all related code and data at *c-mind.org/browse* to discuss, validate and rank shared artifacts while extending their meta-description and abstract actions with the help of the community. We described and shared multiple benchmarks, kernels and real applications using cM *code.source* module, various data sets using cM *dataset* module, various parameterized classification algorithms and predictive models using cM *math.model* module, and many others.

We also shared packages with exposed dependencies and installation scripts for many popular tools and libraries in our public cM repository at *c-mind.org/repo* including GCC, LLVM, ICC, Microsoft Visual Studio compilers, PGI compilers, Open64/PathScale compilers, ROSE source-to-source compilers, Oracle JDK, VTune, NVIDIA GPU toolkit, perf, gprof, GMP, MPFR, MPC, PPL, LAPACK, and many others. We hope that this will ease the burden of the community to continuously (re-)implement some ad-hoc and often unreleased experimental scenarios. In the next sections, we will show how this approach can be used to systematize and formalize auto-tuning.

III. Formalizing Auto-tuning and Predictive Modeling

Almost all research on auto-tuning can be formalized as finding a function of a behavior of a given user program *B* running on a given computer system with a given data set, selected design and optimization choices including program

transformations and architecture configuration **c**, and a system state **s** ([5], [6], [47], [40]):

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

For example, in our current and past research and experimentation, **b** is a behavior vector that includes execution time, power consumption, accuracy, compilation time, code size, device cost, and other important characteristics; **c** represents the available design and optimization choices including algorithm selection, the compiler and its optimizations, number of threads, scheduling, affinity, processor ISA, cache sizes, memory and interconnect bandwidth, etc; and finally **s** represents a state of the system including processor frequency and cache or network contentions.

Knowing and minimizing this function is of a particular importance to our industrial partners when designing, validating and optimizing the next generation of new hardware and software including compilers for a broad range of customers' applications, data sets and requirements (constraints), since it can help reduce time to market and cost for the new systems while increasing return on investment (ROI). However, the fundamental problem is that *this function* B *is highly non-linear with a multi-dimensional discrete and continuous space of choices [11], [47] which is rarely possible to model analytically or evaluate empirically* using exhaustive search unless really small kernels and libraries are used with just one or a few program transformations [5], [6].

This problem motivated research on automatic and empirical modeling of an associated function P that can quickly predict better design and optimization choices for a given computer system **c** based on some features (properties) of an end-users' program, data set and a given hardware **f**, and a current state of a computer system **s**:

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

For example, in our research on machine-learning based optimization, vector **f** includes semantic or static program features [29], [30], [34], [40], data set features and hardware counters [35], [22], system configuration, and run-time environment parameters among many others. However, when trying to implement practical and industrial scenarios in cTuning framework, we spent most of our time on engineering issues trying to expose characteristics, choices, features and system state using numerous, "black box" and not necessarily documented tools. Furthermore, when colleagues with a machine learning background were trying to help us improve optimization predictions, they were often quickly demotivated when trying to understand our terminology and problems.

The Collective Mind approach helped our colleagues solve this problem by formalizing the problem and gradually exposing characteristics **b**, choices **c**, system state **s** and features **f** (meta information) in experimental setups using JSON format as shown in the following real 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", ...}
}
```

Furthermore, we can easily convert JSON hierarchical data into a *flat vector format* to apply above mathematical formalization of auto-tuning and learning problem while making it easily understandable to an interdisciplinary community particularly with a background in mathematics and physics. In our flat format, a *flat key* can reference any key in a complex JSON hierarchy as one string. Such *flat 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, flat key for the second execution time "10.1" in one of the previous examples of information flow can be referenced as *"##characteristics#execution_time@1"*. Finally, users can gradually provide the following cM data specification for the flat keys in information flow to fully automate program optimization and learning (kept together with a given cM module):

```
"flattened_json_key":{
  "type": "text" | "integer" | "float" | "dict" | "list" | "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",
  "default_value": "string"
  ...
}
```

Of course, such format may have some limitations, but it supports well our current research and experimentation on auto-tuning and will be extended only when needed. Furthermore, such implementation allowed us and our colleagues to collaboratively prototype, validate and improve various auto-tuning and learning scenarios simply by chaining available cM modules similar to components and filters in electronics (cM experimental pipelines) and reusing all shared artifacts. For example, we converted our ad-hoc build and run scripts from cTuning framework to a unified cM pipeline consisting of chained cM modules as shown in Figure 6. This pipeline (*ctuning.pipeline.build_and_run*) is implemented and executed as any other cM module to help researchers simplify the following operations during experimentation:

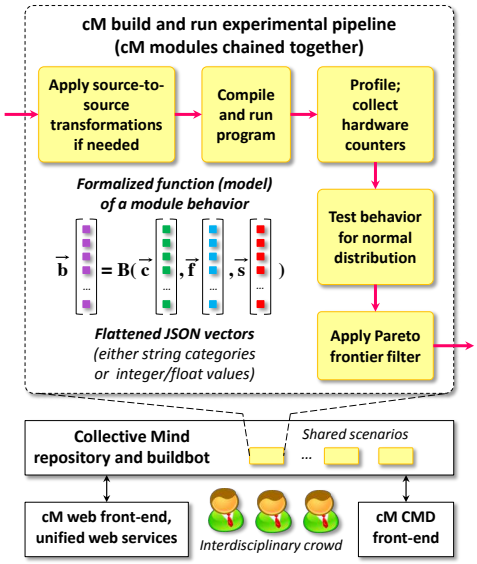

Fig. 6. Unified build and run cM pipeline implemented as chained cM modules.

- Source to source program transformation and instrumentation (if required). For example, we added support for PLUTO polyhedral compiler to enable automatic restructuring and parallelization of loops [48].
- Compilation and execution of any shared programs (real applications, benchmarks and kernels) using *code.source* module. Meta-description of these programs includes information about how to build and execute them. Code can be executed with any shared data set as an input (*dataset* module). The community can gradually share more data sets together with the unified descriptions of their features such as dimensions of images, sizes of matrices and so on.
- Testing of measured characteristics from repeated executions for normal distribution [49] using shared cM module *ctuning.filter.variation* to be able to expose unusual behavior in a reproducible way to the community for further analysis. Unexpected behavior often means that some feature is missing in the experimental pipeline such as frequency or cache contention that can be gradually added by the community to separate executions with different contexts as described further in Section V.
- Applying Pareto frontier filter [50], [19], [40] to leave only optimal solutions during multi-objective optimization when multiple characteristics have to be balanced at the same time such as execution time vs code size vs power consumption vs compilation time. This, in turn, can help to avoid collecting large amounts of off-line and often unnecessary experimental data that can easily saturate repositories and make data analysis too time consuming or even impossible (as happened several times with a public cTuning repository) .

In the next section we show how we can reuse and customize this pipeline (demonstrated online at *c-mind.org/ctuning-pipeline*) to systematize and run some existing auto-tuning scenarios from our industrial partners.

IV. Systematizing Auto-Tuning and Learning Scenarios

Unified cM build and run pipeline combined with mathematical formalization allows researchers and engineers to focus their effort on implementing and extending universal auto-tuning and learning scenarios rather than hardwiring them to specific systems, compilers, optimizations or tuned characteristics. This, in turn, allows to distribute long tuning process across multiple users while potentially solving an old and well-known problem of using a few possibly non-representative benchmarks and a limited number of architectures when developing and validating new optimization techniques.

Gradually extend cM build and run pipeline module		Gradually expose characteristics	Gradually expose design and optimization choices, features
Select algorithm		(time) productivity, variable-accuracy, complexity ...	Language, MPI, OpenMP, TBB, MapReduce ...
Analyze and transform program	Process	time; memory usage; code size ...	transformation ordering; polyhedral transformations; transformation parameters; instruction ordering; MPI parameters; number of threads;
	Function		
	Kernel		
	Loop		
	Instruction		
Build program		time ...	compiler flags; pragmas ...
Run code	Run-time environment	time; power consumption ...	pinning/scheduling ...
	System	cost; size ...	CPU/GPU; frequency; memory ...
	Data set	size; values; description ...	precision ...
	Run-time analysis	time; precision ...	hardware counters; power meters ...
	Run-time state	processor state; cache state ...	helper threads; hardware counters ...
Analyze profile	Statistical analysis	time; size ...	instrumentation; profiling ...
Model behavior		size; precision	model type ...

Fig. 7. Gradual and collaborative top-down decomposition of computer system software and hardware using cM modules (wrappers) similar to methodology in physics. First, coarse-grain design and optimization choices and features are exposed and tuned, and later more fine-grain choices are exposed depending on the available tuning time budget and expected return on investment.

Furthermore, it is now possible to take advantage of mature interdisciplinary methodologies from other sciences such as physics and biology to analyze and learn the behavior of complex systems. Therefore, cM uses a top-down methodology to decompose software and hardware into simple sub-components to be able to start learning and tuning of a global, coarse-grain program behavior with respect to exposed coarse-grain tuning choices and features. Later, depending on user requirements, time budget and expected return on

investment during optimization, the community can extend components to cover finer-grain tuning choices and behavior as conceptually shown in Figure 7. Note, that when analyzing a application at a finer-grain levels such as code regions, we consider them as interacting cM components with their own vectors of tuning choices, characteristics, features and internal states. In doing so, we can analyze and learn their behavior using methodologies from quantum mechanics or agent-based modeling [51].

A. Unifying design and optimization space exploration

As the first practical usage scenario, we developed a universal and customizable design and optimization space exploration as cM module *ctuning.scenario.exploration* on top of *ctuning.pipeline.build_and_run_program* module to substitute most ad-hoc tuning scripts and frameworks from our past research. This scenario can be executed from the command line as any other cM module thus enabling relatively easy integration with third-party tools including compiler regression buildbots or Eclipse-based framework. However, the most user friendly way to run scenarios is through the cM web interface as demonstrated at *c-mind.org/ctuning-exploration* (note that we plan to improve the usability of this interface with dynamic HTML, JavaScript and Ajax technology [52] while hiding unnecessary information from users and avoiding costly page refreshes). In such a way, cM will query all chained modules for this scenario to automatically visualize all available tuning choices, characteristics, features and system states. cM will also preset all default values (if provided by specification) while allowing a user to select which choices to explore, characteristics to measure, search strategy to use, and statistical analysis for experimental results to apply.

We currently implemented and shared *uniform random and exhaustive* exploration strategies. We also plan to add adaptive, probabilistic and hill climbing sampling from our past research [17], [21] or let users develop and share any other universal strategy which is not hardwired to any specific tool but can explore any available choices exposed by the scenario.

Next, we present several practical and industrial auto-tuning scenarios using above customized exploration module.

B. Systematizing compiler benchmarking

Validating new architecture designs across multiple benchmarks, tuning optimization heuristics of multiple versions of compilers, or tuning compiler flags for a customer application is a tedious, time consuming and often ad-hoc process that is far from being solved. In fact, it becomes even tougher with time due to the ever rising number of available optimizations (Figure 1) and many strict requirements placed on compilers such as generating fast and small code for all possible existing architectures within a reasonable amount of time.

Collective Mind helps unify and distribute this process among many machines as a performance tracking buildbot. For this purpose, we customized universal cM exploration module for compiler flag tuning as a new *ctuning.scenario.compiler.optimizations* module. Users just need to choose a compiler version and related description of flags (example is available at *c-mind.org/ctuning-compiler-desc*) as an input and select either to explore a compiler flag optimization space for a given program or distribute tuning of a default compiler optimization heuristic across many machines using a set of shared benchmarks. Note, that it is possible to use Collective Mind not only on desktop machines, servers, data centers and cloud services but also on bare metal hardware or Android-based mobile devices (either through SSH or using a special *Collective Mind Node* application available in Google Play Store [53] to help deploy and crowdsource experiments on mobile phones and tablets while aggregating results in web-based cM repositories).

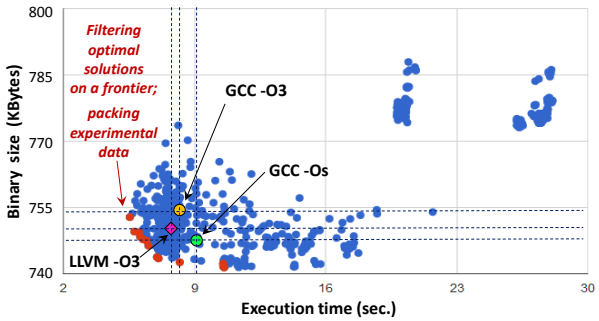

Fig. 8. Compiler flag auto-tuning to improve execution time and code size of a shared image corner detection program with a fixed data set on Samsung Galaxy Series mobile phone using cM for Android. Highlighted points represent frontier of optimal solutions as well as GCC with -O3 and -Os optimization flags versus LLVM with -O3 flag (*c-mind.org/interactive-graph-demo*).

To demonstrate this scenario, we optimized a real image corner detection program on a commodity Samsung Galaxy Series mobile phone with ARMv6 830MHz processor using Sourcery GCC v4.7.2 with randomly generated combinations of compiler flags of format *-O3 -f(no-)optimization_flag –parameter param=random_number_from_range*, LLVM v3.2 with *-O3* flag, and a chained Pareto frontier filter (cM module *ctuning.filter.frontier*) for multi-objective optimization (balancing execution time, code size and compilation time).

Experimental results during such exploration (cM module output) are continuously recorded in a repository in a unified flat vector format making it possible to immediately take advantage of numerous and powerful public web services for visualization, data mining and analytics (for example from Google, Microsoft, Oracle and IBM) or available as packages for Python, Weka, MATLAB, SciLab, and R. For example, Figure 8 shows 2D visualization of these experimental results using public Google Web Services integrated with cM. Such interactive graphs are particularly useful when working in workgroups or for interactive publications (as demonstrated at *c-mind.org/interactive-graph-demo*).

Note, that we always suggest to run optimized code several times to check variation and test distribution for normality as we used to do in physics and electronics. If such a test

fails or the variation of any characteristic dimension is more than some threshold (currently set as 2%), we do not skip such case but record it as *suspicious* including all inputs and outputs for further validation and analysis by the community as described in Section V. At the same time, using a Pareto frontier filter allows users to easily select the most appropriate solution depending on the further intended usage of their applications, i.e. the fastest variant if used for HPC systems, the smallest variant if used for embedded devices with very limited resources, such as credit card chips or the future "Internet of Things" devices, or balanced for both speed and size when used in mobile phones and tablets.

Since Collective Mind also enables co-existence of multiple versions of different compilers, checks output of programs for correct execution during optimization, and supports multiple shared benchmarks and data sets, it can be easily used as a distributed and public buildbot for rigorous performance tracking and simultaneous tuning of compilers (as shown in Figure 2) while taking advantage of a growing number of shared benchmarks and data sets [54]. Longer term, we expect that such an approach will help the community fully automate compiler tuning for new architectures or even validate new processor designs for errors. It can also help derive a realistic, diverse and representative training set of benchmarks and data sets [24] to systematize and speed up training for machine learning based optimization prediction for previously unseen programs and architectures [40].

To continue this collaborative effort, we shared the description of all parametric and boolean (on or off) compiler flags in JSON format as "choices" for a number of popular compilers including GCC, LLVM, Open64, PathScale, PGI and ICC under *ctuning.compiler* module. We also implemented and shared several off-the-shelf classification and predictive models including KNN and SVM from our past research [34], [35], [40] using *math.model* module to be able to automatically predict better compiler optimization using semantic and dynamic program features. Finally, we started implementing standard complexity reduction and differential analysis techniques [55], [56] in cM to iteratively isolate unusual program behavior [57] or to find minimal set of representative benchmarks, data sets and correlating features [24], [42]. Users can now collaboratively analyze unexpected program behavior, improve predictive models, find best tuning strategies and collect minimal set of influential optimizations, representative features, most accurate models, benchmarks and data sets.

C. Systematizing modeling of application behavior to focus optimizations

Since programs may potentially have an infinite number of data sets while auto-tuning is already time consuming, it is usually performed for one or a few and not necessarily representative data sets. Collective Mind can help to systematize and automate modeling of a behavior of a given application across multiple data sets to suggest where to focus further tuning (adaptive sampling and online learning) [34], [58], [21]. We just needed to customize previously introduced

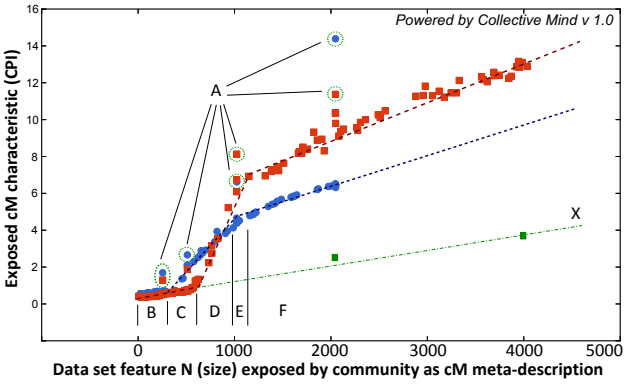

Fig. 9. Online learning (predictive modeling) of a CPI behavior of a shared LU-decomposition benchmark on 2 different platforms (Intel Core2 shown in red vs Intel i5 shown in blue) vs vector size N (data set feature).

auto-tuning pipeline to explore data set parameters (already exposed through *dataset* module) and model program behavior at the same time using either off-the-shelf predictive models including linear regression, Support Vector Machines (SVM), Multivariate Adaptive Regression Splines (MARS), and neural networks available for R language and abstracted by cM module *math.model.r*, or shared user-defined hybrid models specific for a given application.

For example, Figure 9 demonstrates how such exploration and online learning is performed using cM together with shared LU-decomposition benchmark versus size of input vector (N), measured CPI characteristic, and 2 Intel-based platforms (Intel Core2 Centrino T7500 Merom 2.2GHz L1=32KB 8-way set-associative, L2=4MB 16-way set associative - red dots vs. Intel Core i5 2540M 2.6GHz Sandy Bridge L1=32KB 8-way set associative, L2=256KB 8-way set associative, L3=3MB 12-way set associative - blue dots).

In the beginning, cM does not have any knowledge about behavior of this (or any other) benchmark, so it simply observes and stores available characteristics along with the data set features. At each exploration (sampling) step, cM processes all historical observations using various available or shared predictive models such as SVM or MARS in order to find correlations between data set features and characteristics. At the same time it attempts to minimize Root-Mean-Square Deviation (RMSE) between predicted and measured values for all available points. Even if RMSE is relatively low, cM can continue exploring and observing behavior in order to detect discrepancies (failed predictions).

Interestingly, in our example, practically no off-the-shelf model could detect the A outliers (singularities) which appear due to cache alignment problems. However, having mathematical formalization helps interdisciplinary community to find and share better models that minimized RMSE and model size at the same time. In the presented case, our colleagues from machine learning department managed to fit and share a hybrid, parameterized, rule-based model that first validates cases where data set size is a power of 2, otherwise it uses linear models as functions of a data set and cache size.

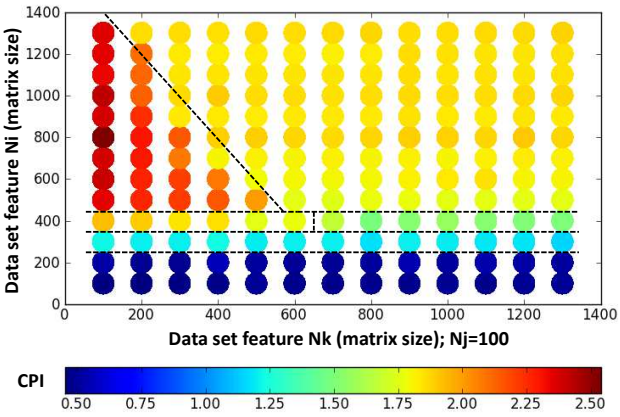

Fig. 10. CPI behavior of a matrix-matrix multiply benchmark (CID=45741e3fbcf4024b:116a9c375e7d7e14) on Intel i5 platform vs different matrix sizes. Hyperplanes separate areas with similar behavior found using multivariate adaptive regression splines (MARS).

This model resembles reversed analytical roofline model [59] though is continuously and empirically refined to capture even fine-grain effects. In contrast, standard MARS model managed to predict the behavior of a matrix-matrix multiplication kernel for different matrix sizes as shown in Figure 10.

Such models can help focus auto-tuning on areas with distinct behavior as described in [17], [58], [21]. For example presented in Figure 9, outlier points A can be optimized using array padding; area B can profit from parallelization and traditional compiler optimizations targeting ILP; areas C-E can benefit from loop tiling; points A saturate memory bus and can also benefit from reduced processor frequency to save energy. Such optimizations can be performed automatically if exposed through cM or provided by the community as shared advices using *ctuning.advice* module.

In the end, multiple customizable models can be shared as parameterized cM modules along with applications thus allowing the community to continuously refine them or even reuse them for similar classes of applications. Finally, such predictive models can be used for effective and online compaction of experiments while avoiding collection of a large amount of data (known in other fields as a "big data" problem) and leaving only representative or unexpected behavior. It can, in turn, minimize communications between cM nodes while making Collective Mind a giant and distributed learning and decision making network to some extent similar to the brain [42].

D. Enabling fine-grain auto-tuning through plugins

After learning and tuning coarse-grain behavior, we gradually move to finer-grain levels including selected code regions, loop transformations, MPI parameters and so on, as shown in Figure 7. However, in our past research, it required messy instrumentation of applications and development of complex source-to-source transformation tools and pragma-based languages.

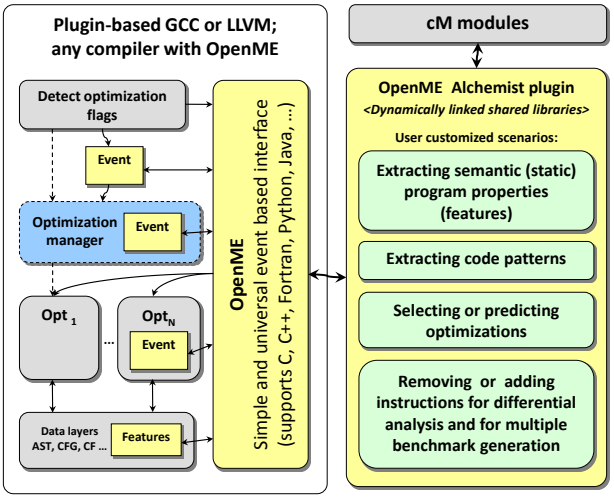

Fig. 11. Conceptual structure of compilers supporting plugin and event based interface to enable fine-grain tuning and learning of their internal and often hidden heuristics through external plugins [38].

As an alternative and simpler solution, we developed an event-based plugin framework (Interactive Compilation Interface and was recently substituted by a new and universal OpenME plugin-based framework connected to cM) to expose tuning choices and semantic program features from production compilers such as GCC and LLVM through external plugins [60], [43], [38]. This plugin-based tuning technique helped us to start unifying, cleaning up and converting rigid compilers into powerful and flexible research toolsets. Such an approach also helped companies and end-users to develop their own plugins with customized optimization and tuning scenarios without rebuilding compilers and instrumenting applications thus keeping them clean and portable.

This framework also allowed to easily expose multiple semantic code features to automatically learn and improve all optimization and tuning decisions using standard machine learning techniques as conceptually shown in Figure 11. This plugin-based tuning technique was successfully used in the MILEPOST project to automate online learning and tuning of the default optimization heuristic of GCC for new reconfigurable processors from ARC during software and hardware co-design [40]. The plugin framework was eventually added to mainline GCC since version 4.6. We are gradually adding it to cM to support plugin-based selection and ordering of internal compiler passes, tuning and learning of internal compiler decisions, and aggregation of semantic program features in a unified format using *ctuning.scenario.program.features.milepost* module.

Plugin-based static compilers can help users automatically or interactively tune a given application with a given data set for a given architecture. However, different data sets or run-time system state often require different optimizations and tuning parameters that should be dynamically selected during

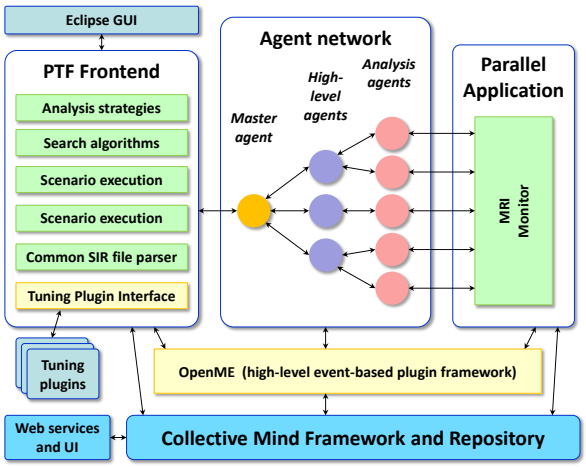

Fig. 12. PTF plugin-based application online tuning framework.

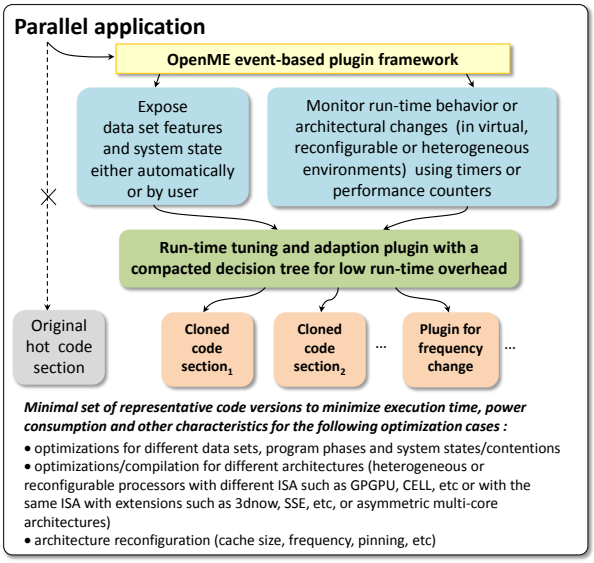

Fig. 13. Making run-time adaptation and tuning practical using static multi-versioning, features exposed by users or automatically detected, and predictive modeling (decision trees) while avoiding complex dynamic recompilation frameworks.

execution. Therefore, Periscope Tuning Framework (PTF) [26] was designed to enable and automate online tuning of parallel applications using external plugins with integrated tuning strategies. Users need to instrument application to expose required tuning parameters and measured characteristics for a given application. At the same time, tuning space can be considerably reduced inside such plugins per given application using previous compiler analysis or expert knowledge about typical performance bottlenecks and ways to detect and improve them as conceptually shown in Figure 12.

Once the online tuning process is finished, PTF generates a report with the recommended tuning actions which can be integrated either manually or automatically into the application for further production runs. Currently, PTF includes plugins to tune execution time of high-level parallel kernels for GPGPUs, balance energy consumption via CPU frequency scaling, optimize MPI runtime parameters among many other scenarios in development.

Collective Mind can help PTF distribute tuning of shared benchmarks and data sets among many users, aggregate results in a common repository, apply data mining and machine learning plugins to prune tuning spaces, and automate prediction of optimal tuning parameters. PTF and cM can also complement each other in terms of tuning coverage since cM currently focuses on global, high-level, machine-learning guided optimizations and compiler tuning while PTF currently focuses on finer-grain online application tuning. In our future work we plan to connect PTF and cM together using cM OpenME interface.

E. Systematizing split compilation and adaptive scheduling

Many current online auto-tuning techniques have a limitation - they usually do not support arbitrary online code restructuring unless complex just-in-time (JIT) compilers are used. As a possible solution to this problem, we introduced split compilation to statically enable dynamic optimizations and adaptation by cloning hot functions or kernels during compilation and providing run-time selection mechanism depending on data set features, target architecture features and a system state [61], [24], [62], [22]. However, since this approach still requires a long and off-line training phase, we can now use Collective Mind to systematize off-line tuning and learning of a program behavior across many data sets and computer systems as conceptually shown in Figure 13.

```
2mm.c / 2mm.cu
...
#ifdef OPENME
#include <openme.h>
#endif
...

int main(void) {
...
#ifdef OPENME
 openme_init(NULL,NULL,NULL,0);
 openme_callback("PROGRAM_START", NULL);
#endif
...

#ifdef OPENME
 openme_callback("SELECT_KERNEL", &adapt);
#endif

#ifdef OPENME
 openme_callback("KERNEL_START", NULL);
#endif

if (adaptive_select==0) mm2_cpu(A, B, C, D, E);
elif (adaptive_select==1) cl_launch_kernel(A,B,C,D,E);
elif (adaptive_select==2) mm2Cuda(A, B, C, D, E,
                                  E_outputFromGpu);

#ifdef OPENME
 openme_callback("KERNEL_END", NULL);
#endif
...

#ifdef OPENME
 openme_callback("PROGRAM_END", NULL);
#endif
...
}
```

Fig. 14. Example of predictive scheduling of matrix-matrix multiplication kernel for heterogeneous architectures using OpenME interface and statically generated kernel clones with different algorithm implementations and optimizations to find the winning one at run-time.

Now, users can take advantage of continuously collected knowledge about program behavior and optimization in the repository to derive a minimal set of representative optimizations or tuning parameters covering application behavior across as many data sets and architectures as possible [24]. Furthermore, it is now possible to reuse machine learning techniques from cM to automatically derive small and fast decision trees needed for realistic cases shown in

Figures 3, 9 and 10. Such decision trees can now be integrated with the application through OpenME or PTF plugins to dynamically select appropriate clones and automatically adapt for heterogeneous architectures particularly in supercomputers and data centers, or even execute some external tools to reconfigure architecture (change frequency, for example) based on exposed features to minimize execution time, power consumption and other user objectives. These data set, program and architecture features can also be exposed through plugins either automatically using OpenME-based compilers or manually through application annotation and instrumentation.

OpenME was designed especially to be very easy to use for researchers and provide a simple connection between Python-based Collective Mind and other modules or plugins written in other languages including C, C++, Fortran and Java. It has only two functions to initialize an event with an arbitrary string name, and to call it with a *void type* argument that will be handled by a user plugin and can range from a simple integer to a cM JSON dictionary. However, since such implementation of OpenME can be relatively slow, we use fast Periscope Tuning Framework for fine-grain tuning. Possible example of such implementation for predictive scheduling of matrix multiply using OpenME interface and several clones for heterogeneous architectures [22] is presented in Figure 14.

Our static function cloning approach with dynamic adaptation was recently added to mainline GCC since version 4.8. We hope that together with OpenME, PTF and cM, it will help systematize research on split compilation while focusing on finding and exposing the most appropriate features to improve run-time adaptation decisions [24] using recent advances in machine learning, data mining and decision making [63], [64], [65], [66].

F. Automating benchmark generation and differential analysis

Our past research on machine learning to speed up auto-tuning suffered from yet another well-known problem: lack of large and diverse benchmarks. Though Collective Mind helps share multiple programs and data sets including ones from [23], [67], [40], it may still not be enough to cover all possible program behavior and features. One possibility is to generate many synthetic benchmarks and data sets but it always result in explosion in tuning and training times. Instead, we propose to use Alchemist plugin [42] together with plugin-enabled compilers such as GCC to use existing benchmarks, kernels and even data sets as templates and randomly modify them by removing, modifying or adding various instructions, basic blocks, loops and so on. Naturally, we can ignore crashing variants of the code and continue evolving only the working ones.

We can use such an approach not only to gradually extend realistic training sets, but also to iteratively identify various behavior anomalies or detect missing code features to explain unexpected behavior similar to differential analysis from electronics [55], [56]. For example, we are adding support to Alchemist plugin to iteratively scalarize memory accesses to characterize code and data set as CPU or memory bound [57], [47]. Its prototype was used to obtain line X in Figure 9 showing ideal code behavior when all floating point memory accesses are NOPed. Additionally, we use Alchemist plugin to unify extraction of code structure, patterns and other features to collaboratively improve prediction during software/hardware co-design [36].

V. Enabling and Improving Reproducibility of Experimental Results

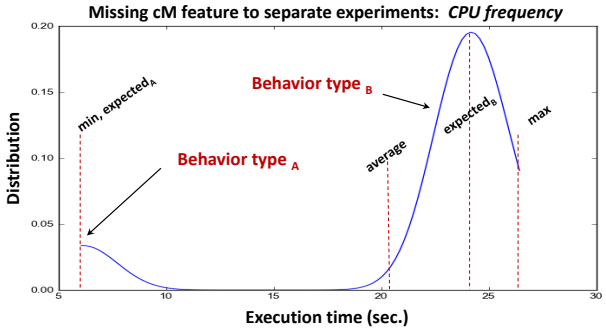

Fig. 15. Unexpected behavior helped to identify and add a missing feature to cM (processor frequency) as well as software dependency (cpufreq) that ensures reproducibility of experimental results.

Since cM allows to implement, preserve and share the whole experimental setup, it can also be used for reproducible research and experimentation. For example, unified module invocation in cM makes it possible to reproduce (replay) any experiment by saving JSON input for a given module and an action, and comparing JSON output. At the same time, since execution time and other characteristics often vary, we developed and shared cM module that applies Shapiro-Wilk test from R to test monitored characteristic for normality. However, in contrast with current experimental methodologies where results not passing such test are simply skipped, we record them in a reproducible way to find and explain missing features in the system. For example, when analyzing multiple executions of image corner detection benchmark on a smart phone shown in Figure 8, we noticed an occasional 4x difference in execution times as shown in Figure 15. 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 relatively obvious, this information allowed us to add CPU frequency to the build and run pipeline using *cpufreq* module and thus separate such experiments. Therefore, Collective Mind research methodology can gradually improve reproducibility as a side effect and with the help of the community rather than trying to somehow enforce it from the start.

VI. Conclusions and Future Work

This paper presents our novel, community-driven approach to make auto-tuning practical and move it to mainstream production environments. However, rather than searching for yet another “holy grail” auto-tuning technique, we propose to

start preserving, sharing and reusing already available practical knowledge and experience about program optimization and hardware co-design using Collective Mind framework and repository. Such approach helps researchers and engineers quickly prototype and validate various auto-tuning and learning techniques as plugins connected into experimental pipelines while reusing all shared artifacts. Such pipelines can be distributed among many users to collaboratively learn, model and tune program behavior using standard top-down methodology from mature sciences such as physics by decomposing complex software into interconnected components while capturing first coarse-grain effects and later moving to finer-grain levels. At the same time, any unexpected behavior and optimization mispredictions are exposed to the community in a reproducible way to be explained and improved. Therefore, we can collaboratively search for profitable optimizations, efficient auto-tuning strategies, truly representative benchmarks, and most accurate models to predict optimizations together with minimal set of relevant semantic and dynamic features.

Our future collaborative work includes exposing more tuning dimensions, characteristics and features using Collective Mind and Periscope tuning frameworks to eventually tune the whole computer system while extrapolating collected knowledge to build faster, more power efficient and reliable self-tuning computer systems. We are working with the community to gradually unify existing techniques and tools including pragma-based source-to-source transformations [68], [69], plugin-based GCC and LLVM to expose and tune all internal optimization decisions [40], [42]; polyhedral source-to-source transformation tools [48]; differential analysis to detect performance anomalies and CPU/memory bounds [57], [47]; just-in-time compilation for Android Dalvik or Oracle JDK; algorithm-level tuning [70]; techniques to balance communication and computation in numerical codes particularly for heterogeneous architectures [71], [27]; Scalasca framework to automate analysis and modeling of scalability of HPC applications [72], [73]; LIKWID for light-weight collection of hardware counters [74]; HPCC and HPCG benchmarks to collaboratively rank HPC systems [75], [76]; benchmarks from GCC and LLVM, TAU performance tuning framework [77]; and all recent Periscope application tuning plugins [25], [26].

At the same time we plan to use collected and unified knowledge to improve our past techniques on decomposition of complex programs into interconnected kernels, predictive modeling of program behavior, and run-time tuning and adaptation [61], [51], [24], [78], [58], [22], [79], [80], [81]. Finally, we are extending Collective Mind to assist recent initiatives on reproducible research and new publication models in computer engineering where all experimental results and related research artifacts with all dependencies are continuously shared along with publications to be validated and improved by the community [82].

ACKNOWLEDGMENTS

We would like to thank David Kuck, David Wong, Abdul Memon, Yuriy Kashnikov, Michael Pankov, Siegfried Benkner, David Padua, Olivier Zendra, Arnaud Legrand, Sascha Hunold, Jack Davidson, Bruce Childers, Alex K. Jones, Daniel Mosse, Jean-Luc Gaudiot, Christian Poli, Egor Pasko, Mike O'Boyle, Marco Cornero, Koen De Bosschere, Lieven Eeckhout, Francois Bodin, Thomas Fahringer, I-hsin Chung, Paul Hovland, Prasanna Balaprakash, Stefan Wild, Hal Finkel, Bernd Mohr, Arutyun Avetisyan, Anton Korzh, Andrey Slepuhin, Vladimir Voevodin, Christophe Guillon, Christian Bertin, Antoine Moynault, Francois De-Ferriere, Reiji Suda, Takahiro Katagiri, Weichung Wang, Chengyong Wu, Marisa Gil, Lasse Natvig, Nacho Navarro, Vittorio Zaccaria, Chris Fensch, Ayal Zaks, Pooyan Dadvand, Paolo Faraboschi, Ana Lucia Varbanescu, Cosmin Oancea, Petar Radojkovic, Christian Collberg, Olivier Temam and Panos Tsarchopoulos, as well as cTuning, Collective Mind, Periscope and HiPEAC communities for interesting related discussions and feedback. We are also very grateful to anonymous reviewers for their insightful comments.

REFERENCES

[1] 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.

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

[3] Jack Dongarra et.al. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[4] PRACE: Partnership for Advanced Computing in Europe. http://www.prace-project.eu.

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

[6] B. Aarts et.al. OCEANS: Optimizing compilers for embedded applications. In *Proc. Euro-Par 97*, volume 1300 of *Lecture Notes in Computer Science*, pages 1351–1356, 1997.

[7] 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.

[8] 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.

[9] M.J. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of International Conference on Parallel Processing*, 2000.

[10] 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.

[11] G.G. Fursin, M.F.P. O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.

[12] 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.

[13] 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.

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

[15] 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, pages 1–24, 2004.

[16] 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.

[17] 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.

[18] 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.

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

[20] David H. Bailey et.al. Peri auto-tuning. *Journal of Physics: Conference Series (SciDAC 2008)*, 125:1–6, 2008.

[21] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):20:1–20:29, December 2010.

[22] 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.

[23] 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.

[24] 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.

[25] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16, 2010.

[26] Renato Miceli et.al. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In *Proceedings of the 11th International Conference on Applied Parallel and Scientific Computing*, PARA'12, pages 328–342, Berlin, Heidelberg, 2013. Springer-Verlag.

[27] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Comput.*, 36(5-6):232–240, June 2010.

[28] M. Baboulin, D. Becker, and J. Dongarra. A parallel tiled solver for dense symmetric indefinite systems on multicore architectures. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 14–24, May 2012.

[29] 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.

[30] 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.

[31] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Perform. Eval. Rev.*, 32(1):2–13, June 2004.

[32] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2005.

[33] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.

[34] 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.

[35] 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.

[36] 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.

[37] Jie Shen, Ana Lucia Varbanescu, Henk J. Sips, Michael Arntzen, and Dick G. Simons. Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In *Conf. Computing Frontiers*, page 14, 2013.

[38] 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.

[39] MILEPOST project archive (MachIne Learning for Embedded PrOgramS opTimization). http://cTuning.org/project-milepost.

[40] Grigori Fursin et.al. MILEPOST GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[41] Vinton G. Cerf. Where is the science in computer science? *Communications of the ACM*, 55(10):5–5, 2012.

[42] Grigori Fursin. Collective Mind: cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning. *CoRR*, abs/1308.2410, 2013.

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

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

[45] Online JSON introduction. http://www.json.org.

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

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

[48] 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.

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

[50] 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.

[51] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, New York, NY, USA, 2008.

[52] Jesse James Garrett. Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications.

[53] Collective Mind Node: deploying experiments on Android-based mobile computer systems. https://play.google.com/store/apps/details?id=com.collective_mind.node.

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

[55] 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.

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

[57] Grigori Fursin, Michael F. P. O'Boyle, Olivier Temam, and Gregory Watts. A fast and accurate method for determining a lower bound on execution time: Research articles. *Concurrency: Practice and Experience*, 16(2-3):271–292, January 2004.

[58] Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136 – 2145, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011.

[59] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[60] Grigori Fursin and Albert Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007.

[61] 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.

[62] Jason Mars and Robert Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO*, pages 169–179, 2009.

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

[64] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[65] 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.

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

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

[68] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *Proceedins of the Workshop on Performance Optimization of High-level Languages and Libraries (POHLL) co-located with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[69] 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.

[70] 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.

[71] Marc Baboulin, Simplice Donfack, Jack Dongarra, Laura Grigori, Adrien Rémy, and Stanimire Tomov. A Class of Communication-avoiding Algorithms for Solving General Dense Linear Systems on CPU/GPU Parallel Machines. In *ICCS*, pages 17–26, 2012.

[72] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[73] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *SC*, page 45, 2013.

[74] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. *CoRR*, abs/1004.4431, 2010.

[75] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) Benchmark Suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[76] Michael Allen Heroux and Jack Dongarra. Toward a new metric for ranking high performance computing systems. http://0-www.osti.gov.iii-server.ualr.edu/scitech/servlets/purl/1089988, Jun 2013.

[77] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[78] 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.

[79] 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.

[80] 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.

[81] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[82] Grigori Fursin and Christophe Dubach. Experience report: community-driven reviewing and validation of publications. In *Proceedings of the 1st Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering (ACM SIGPLAN TRUST'14)*. ACM, 2014.