Collective Mind: cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning

Grigori Fursin
INRIA, France
Grigori.Fursin@cTuning.org

Abstract

Software and hardware co-design and optimization of HPC systems has become intolerably complex, ad-hoc, time consuming and error prone due to enormous number of available design and optimization choices, complex interactions between all software and hardware components, and multiple strict requirements placed on performance, power consumption, size, reliability and cost.

We present our novel long-term holistic and practical solution to this problem based on customizable, plugin-based, schema-free, heterogeneous, open-source Collective Mind repository and infrastructure with unified web interfaces and on-line advise system. This collaborative framework distributes analysis and multi-objective off-line and on-line auto-tuning of computer systems among many participants while utilizing any available smart phone, tablet, laptop, cluster or data center, and continuously observing, classifying and modeling their realistic behavior. Any unexpected behavior is analyzed using shared data mining and predictive modeling plugins or exposed to the community at cTuning.org for collaborative explanation, top-down complexity reduction, incremental problem decomposition and detection of correlating program, architecture or run-time properties (features). Gradually increasing optimization knowledge helps to continuously improve optimization heuristics of any compiler, predict optimizations for new programs or suggest efficient run-time (online) tuning and adaptation strategies depending on end-user requirements. We decided to share all our past research artifacts including hundreds of codelets, numerical applications, data sets, models, universal experimental analysis and auto-tuning pipelines, self-tuning machine learning based meta compiler, and unified statistical analysis and machine learning plugins in a public repository to initiate systematic, reproducible and collaborative research, development and experimentation with a new publication model where experiments and techniques are validated, ranked and improved by the community.

Keywords *Collective Mind, crowdtuning, crowdsourcing auto-tuning and co-design, software and hardware co-design and co-optimization, on-line tuning and learning, systematic behavior modeling, predictive modeling, data mining, machine learning, on-line advice system, metadata, top-down optimization, incremental problem decomposition, decremental (differential) characterization, decremental analysis, complexity reduction, tuning dimension reduction, customizable plugin-based infrastructure, public repository of knowledge, big data processing and compaction, agile research and development, cTuning.org, c-mind.org , systematic and reproducible research and experimentation, validation by community*

1 Introduction, major challenges, and related work

1.1 Motivation

Continuing innovation in science and technology is vital for our society and requires ever increasing computational resources. However, delivering such resources particularly with exascale performance for HPC or ultra low power for embedded systems is becoming intolerably complex, costly and error prone due to limitations of available technology, enormous number of available design and optimization choices, complex interactions between all software and hardware components, and growing number of incompatible tools and techniques with ad-hoc, intuition based heuristics. As a result, understanding and modeling of the overall relationship between end-user algorithms, applications, compiler optimizations, hardware designs, data sets and run-time behavior, essential for providing better solutions and computational resources, became simply infeasible as confirmed by numerous recent long-term international research visions about future computer systems [12, 9, 19, 30, 8, 7]. On the other hand, the research and development methodology for computer systems has hardly changed in the past decades: computer architecture is first designed and later compiler is being tuned and adapted to the new architecture using some ad-hoc benchmarks and heuristics. As a result, peak performance of the new systems is often achieved only for a few previously optimized and not necessarily representative benchmarks such as SPEC for desktops and servers or LINPACK for TOP500 supercomputer ranking, while leaving most of the systems severely underperforming and wasting expensive resources and power.

Automatic off-line and on-line performance tuning techniques were introduced nearly two decades ago in an attempt to solve some of the above problems. These approaches treat computer systems as a black box and explore their optimization parameter space empirically, i.e. compiling and executing a user program multiple times with varying optimizations or designs (compiler flags and passes, fine-grain transformations, frequency adaptation, cache reconfiguration, parallelization, etc) to empirically find better solutions that improve execution and compilation time, code size, power consumption and other characteristics [46, 37, 14, 17, 33, 26, 18, 35, 45, 42, 39, 40, 31]. Such techniques require little or no knowledge of the current platform and can adapt programs to any given architecture automatically. With time, auto-tuning has been accelerated with various adaptive exploration techniques including genetic algorithms, hill-climbing and probabilistic focused search. However, the main disadvantage of these techniques is an excessively long exploration time of large optimization spaces and lack of optimization knowledge reuse among different programs, data sets and architectures. Moreover, all these exploration steps (compilation and execution) must be performed with exactly the same setup by a given user including the same program, generated with the same compiler on the same architecture with the same data set, and repeated a large number of times to become statistically meaningful.

Statistical analysis and machine learning have been introduced nearly a decade ago to speed up exploration and predict program and architecture behavior, optimizations or system configurations by automatically learning correlations between properties of multiple programs, data sets and architectures, available optimizations or design choices, and observed characteristics [46, 37, 14, 17, 33, 26, 18, 35, 45, 42, 39, 40, 31, 44, 27, 28]. Often used by non-specialists, these approaches mainly demonstrate a potential to predict optimizations or adaptation scenario in some limited cases, but they do not include deep analysis about machine learning algorithms, their selection and scalability for ever growing training sets, optimization choices and available features which are often problem dependent, are the major research challenges in the field of machine learning for several decades, and far from being solved.

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

2 Collective Mind approach

2.1 Back to basics

We would like to start with the formalization of the eventual needs of end-users and system developers or providers. End-users generally need to perform some tasks (playing games on a console, watching videos on mobile or tablet, surfing Web, modeling a new critical vaccine on a supercomputer or predicting a new crash of financial markets using cloud services) either as fast as possible or with some real-time constraints while minimizing or amortizing all associated costs including power consumption, soft and hard errors, and device or service price. Therefore, end-users or adaptive software require a function that can suggest most optimal design or optimization choices **c** based on properties of their tasks and data sets **p**, set of requirements **r**, as well as current state of a used computing system **s**:

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

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

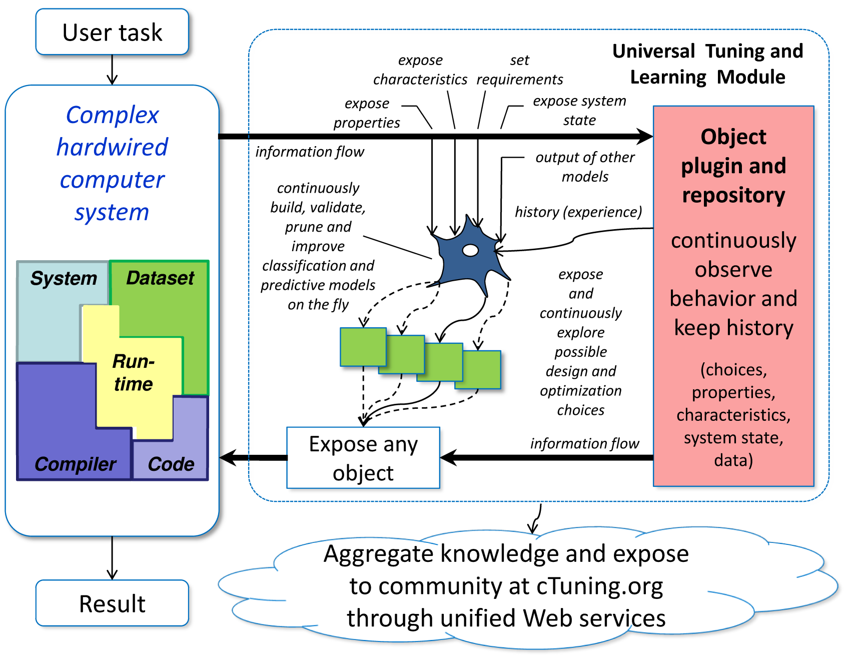
$$\mathbf{b} = B(\mathbf{p}, \mathbf{c}, \mathbf{s})$$

This function is of particular importance for hardware and software designers that need to continuously provide and improve choices (solutions) for a broad range of user tasks, data sets and requirements while trying to improve own ROI and reduce time to market. In order to find optimal choices, it should be minimized in presence of possible end-user requirements (constraints). However, the fundamental problem is that nowadays this function is highly non-linear with such a multi-dimensional discrete and continuous parameter space which is not anymore possible to model analytically or evaluate empirically using exhaustive search [10, 46, 21]. For example, **b** is a behavior vector that can now include execution time, power consumption, compilation time, code size, device cost, and any other important characteristic; **p** is a vector of properties of a task and a system that can include semantic program features [38, 43, 11, 25], dataset properties, hardware counters [15, 32], system configuration, and run-time environment parameters among many others; **c** represents available design and optimization choices including algorithm selection, compiler and its optimizations, number of threads, scheduling, processor ISA, cache sizes, memory and interconnect bandwidth, frequency, etc; and finally **s** represents the state of the system during parallel execution of other programs, system or core frequency, cache contentions and so on.

2.2 Interdisciplinary collaborative methodology

Current multiple research projects mainly show that it is possible to use some off-the-shelf on-line or off-line adaptive exploration (sampling) algorithms combined with some existing models to approximate above function and predict behavior, design and optimization choices for 70-90% cases but in a very limited experimental setup. In contrast, our ambitious long-term goal is to understand how to continuously build, enhance, systematize and optimize hybrid models that can *explain and predict all possible behaviors and choices* while selecting minimal set of representative properties, benchmarks and data sets for predictive modeling [36]. We reuse our interdisciplinary knowledge in physics, quantum electronics and machine learning to build a new methodology that can effectively deal with rising complexity of computer systems through gradual and continuous top-down problem decomposition, analysis and learning. We also develop a modular infrastructure and repository that allows to easily interconnect various available tools and techniques to distribute adaptive probabilistic exploration, analysis and optimization of computer systems among many users [3, 1] while exposing unexpected or unexplained behavior to the community with interdisciplinary backgrounds particularly in machine learning and data mining through unified web interfaces for collaborative solving and systematization.

2.3 Collective Mind infrastructure and repository



User task
Complex hardwired computer system
System
Dataset
Run-time
Compiler
Code
Result
Universal Tuning and Learning Module
expose characteristics
set requirements
expose properties
expose system state
information flow
output of other models
history (experience)
continuously build, validate, prune and improve classification and predictive models on the fly
expose and continuously explore possible design and optimization choices
Object plugin and repository
continuously observe behavior and keep history
(choices, properties, characteristics, system state, data)
information flow
Expose any object
Aggregate knowledge and expose to community at cTuning.org through unified Web services

Figure 1: *Gradual decomposition, parameterization, observation, tuning and learning of complex hardwired computer systems.*

Collective Mind framework and repository (cM for short) enables continuous, collaborative and agile top-down decomposition of the whole complex hardwired computer systems into unified and connected subcomponents (modules) with gradually exposed various characteristics, tuning choices (optimizations), properties and system state as conceptually shown in Figure 1. At a coarse-grain level, modules serve as wrappers around existing command line tools such as compilers, source-to-source transformers, code launchers, profilers, among many others. Such modules are written in python for productivity and portability reasons, and can be launched from command line in a unified way using Collective Mind front-end *cm* as following:

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

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

```
r=cm_kernel.access({'cm_run_module_uoa':<module name or UID>,
                    'cm_action':<command>,
                    parameters})
if r['cm_return']>0:
   print 'Error:'+r['cm_error']
   exit(r['cm_return'])
```

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

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

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

where .cmr is an acronym for Collective Mind Repository. In contrast with using SQL-based database in the first cTuning version that was fast but very complex for data sharing or extensions of structure and relations, a new open format allows users to be database and technology-independent with the possibility to modify, add, delete or share entries and whole repositories using standard OS functions and tools like SVN, GIT or Mercury, or easily convert them to any other format or database when necessary. Furthermore, cM can transparently use open source JSON-based indexing tools such as ElasticSearch [4] to enable fast and powerful queries over schema-free meta information. Now, any research artifact will not be lost and can now be referenced and directly found using the so called cID (Collective ID) of the format: ⟨ *module name or UID* ⟩*:*⟨ *data entry or UID* ⟩.

Such infrastructure allows researchers and engineers to connect existing or new modules into experimental pipelines like "research LEGO" with exposed characteristics, properties, constraints and states to quickly and collaboratively prototype and crowdsource their ideas or production scenarios such as traditional adaptive exploration of large experimental spaces, multi-objective program and architecture optimization or continuous on-line learning and run-time adaptation while easily utilizing all available benchmarks, data sets, tools and models provided by the community. Additionally, single and unified access function enables transparent reproducibility and validation of any experiment by preserving input and output dictionaries for a given experimental pipeline module. Furthermore, we decided to keep all modules inside repository thus substituting various ad-hoc scripts and tools. With an additional cM possibility to install various packages and their dependencies automatically (compilers, libraries, profilers, etc) from the repository or keep all produced binaries in the repository, researchers now have an opportunity to preserve and share the whole experimental setup in a private or public repository possibly with a publication.

We started collaborative and gradual decomposition of large, coarse-grain components into simpler sub-modules including decomposition of programs into kernels or codelets [48] as shown in Figure 2 to keep complexity under control and possibly use multi-agent based or brain inspired modeling and adaptation of the behavior of the whole computer system locally or during P2P

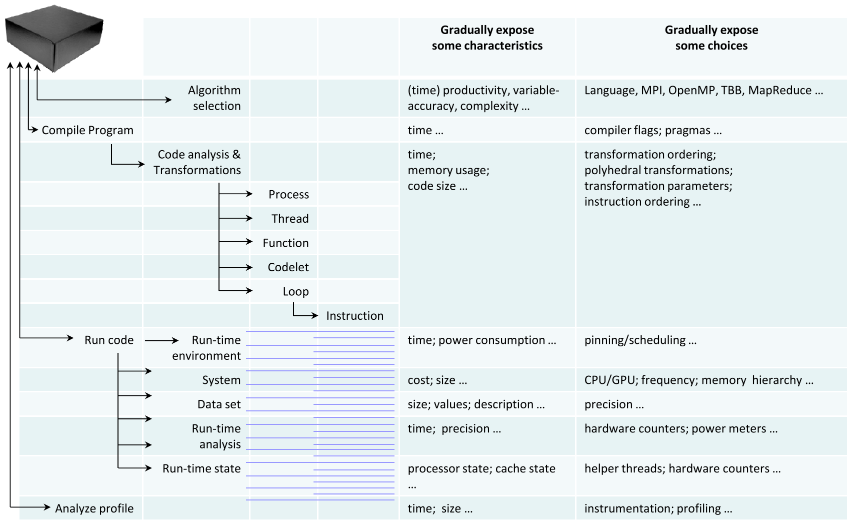

Gradually expose some characteristics
Gradually expose some choices
Algorithm selection
(time) productivity, variable-accuracy, complexity …
Language, MPI, OpenMP, TBB, MapReduce …
Compile Program
time …
compiler flags; pragmas …
Code analysis & Transformations
time; memory usage; code size …
transformation ordering; polyhedral transformations; transformation parameters; instruction ordering …
Process
Thread
Function
Codelet
Loop
Instruction
Run code
Run-time environment
time; power consumption …
pinning/scheduling …
System
cost; size …
CPU/GPU; frequency; memory hierarchy …
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
time; size …
instrumentation; profiling …

Figure 2: *Gradual top-down decomposition of computer systems to balance coarse-grain vs. fine-grain analysis and tuning depending on user requirements and expected ROI*

crowdsourcing. Such decomposition also allows community to first learn and optimize coarse-grain behavior, and later add more fine-grain effects depending on user requirements, time constraints and expected return on investment (ROI) similar to existing analysis methodologies in physics, electronics or finances.

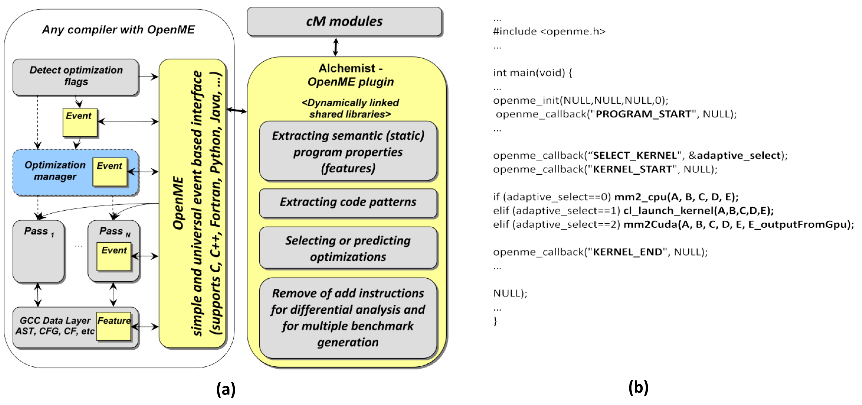
2.4 Data and parameter description and classification

In traditional software engineering, all software components and their API are usually defined at the beginning of the project to avoid modifications later. However, in our case, due to ever evolving tools, APIs and data formats, we decided to use agile methodology together with type-free inputs and outputs for all functions focusing on quick and simple prototyping of research ideas. Only when modules and their inputs and outputs become mature or validated, then (meta)data and interface are defined, systematized and classified. However, they can still be extended and reclassified at any time later.

For example, any key in an input or output dictionary of a given function and a given module can be described as ”choice”, ”(statistical) characteristic”, ”property” and ”state”, besides a few internal types including ”module UID” or ”data UID” or ”class UID” to provide direct or semantic class-based connections between data and modules. Parameters can be discrete or continuous with a given range to enable automatic exploration. Thus, we can easily describe compiler optimizations; dataset properties such as image or matrix size, architecture properties such as cache size or frequency, represent execution time, power consumption, code size, hardware counters; categorize benchmarks and codelets in terms of reaction to optimizations or as CPU or memory bound, and so on.

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

Most of current compilers, applications and run-time systems are not prepared for easy and straightforward fine-grain analysis and tuning due to associated software engineering complexity, sometimes proprietary internals, possible compile or run-time overheads, and still occasional


Any compiler with OpenME
Detect optimization flags
Event
Optimization manager
Event
Pass 1
Pass N
Event
GCC Data Layer AST, CFG, CF, etc
Feature
OpenME
simple and universal event based interface
(supports C, C++, Fortran, Python, Java, ...)
cM modules
Alchemist - OpenME plugin
<Dynamically linked shared libraries>
Extracting semantic (static) program properties (features)
Extracting code patterns
Selecting or predicting optimizations
Remove of add instructions for differential analysis and for multiple benchmark generation
(a)
(b)

```
...
#include <openme.h>
...

int main(void) {
...
openme_init(NULL,NULL,NULL,0);
 openme_callback("PROGRAM_START", NULL);
...

openme_callback("SELECT_KERNEL", &adaptive_select);
openme_callback("KERNEL_START", NULL);

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);

openme_callback("KERNEL_END", NULL);
...

NULL);
...
}
```

Figure 3: *Event and plugin-based OpenME interface to "open up" rigid tools (a) and applications (b) for external fine-grain analysis, tuning and adaptation, and connect them to cM*

disbeliefs in effective run-time adaptation. Some extremes included either fixing, hardwiring and hiding all optimization heuristics from end-users or oppositely exposing all possible optimizations, scheduling parameters, hardware counters, etc. Some other available mechanisms to control fine-grain compiler optimization through pragmas can also be very misleading since it is not always easy or possible to validate whether optimization was actually performed or not.

Instead of developing yet another source-to-source tools or binary translators and analyzers that always require enormous resources and effort often to reimplement functionality of existing and evolving compilers and support evolving architectures, we developed a simple event and plugin-based interface called Interactive Compilation Interface (ICI) to "open up" previously hardwired tools for external analysis and tuning. ICI was written in plain C originally for Open64 and later for GCC, requires minimal instrumentation of a compiler and helps to expose or modify only a subset of program properties or compiler optimization decisions through external dynamic plugins based on researcher needs and usage scenario. This interface can easily evolve with the compiler itself, has been successfully used in the MILEPOST project to build machine-learning self-tuning compiler [25], and is now available in mainline GCC.

Based on this experience, we developed a new version of this interface (OpenME) [3] that is used to "open up" any available tool such as GCC, LLVM, Open64, architecture simulator, etc in a unified way as shown in Figure 3(a), or any application for example to train predictive scheduler on heterogeneous many-core architectures [32] as shown in Figure 3(b). It can be connected to cM to monitor application behavior in realistic environments or utilize on-line learning modules to quickly prototype research ideas when developing self-tuning applications that can automatically adapt to different datasets, underlying architectures particularly in virtual and cloud environments, or react to changes in environment and run-time behavior. Since there are some natural overheads associated with event invocation, users can substitute them with hardwired fast calls after research idea has been validated. We are developing associated Alchemist plugin [3] for GCC to extract code structure, patterns and various properties to substitute and unify outdated MILEPOST GCC plugin [5] for machine-learning based meta compilers.

3 Possible usage scenarios

We decided first to re-implement various analysis, tuning and learning scenarios from our past research as cM modules combined into universal compilation and execution pipeline to give the community a common reproducible base for further research and experimentation. Furthermore, rather than just showing speedups, our main focus is also to use our distributed framework similar to web crawlers to search for unusual or unexpected behavior that can be exposed to the community for further analysis, advise, ranking, commenting, and eventual improvement of cM to take such behavior into account.

3.1 Collaborative observation and exploration

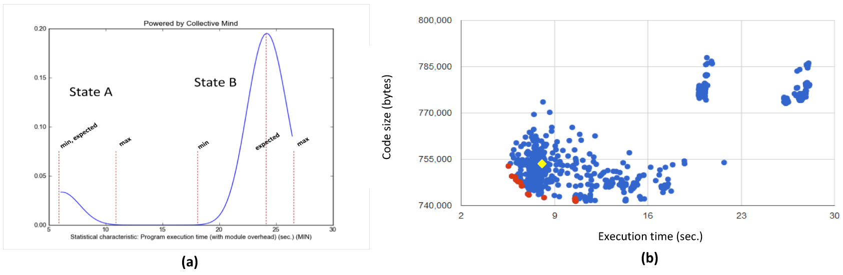


Figure 4: *(a) execution time variation of a susan corner codelet with the same dataset on Samsung Galaxy Y for 2 frequency states; (b) variation in execution time vs code size during GCC 4.7.2 compiler flag auto-tuning for the same codelet on the same mobile phone where yellow rhombus represents -O3 and red circles show Pareto frontier - all data and modules are available for reproduction at c-mind.org/repo*

Having common and portable framework with exposed characteristics, properties, choices and state in a unified way allows us to collaboratively observe, optimize, learn and predict program behavior on any existing system including mobile phones, tablets, desktops, servers, cluster or cloud nodes in realistic environment transparently and on the fly instead of using a few ad-hoc and often non-representative benchmarks, data sets and platforms. Furthermore, it elegantly solves a common problem of a lack of experimental data to be able to properly apply machine learning techniques and make statistically meaningful assumptions that slowed down many recent projects on applying machine learning to compilation and architecture.

For example, Figure 4(a) shows variation of an execution time of an image corner detection codelet with the same dataset (image) on a Samsung Galaxy Y mobile phone with ARM processor using modified Collective Mind Node [2]. Our cM R-based statistical module reports that distribution is not normal that usually results in discarding this experiment in most of the research projects. However, by exposing and analyzing this relatively simple case, we found that processor frequency was responsible for this behavior thus adding it as a new parameter to the ”state” vector of our experimental pipeline to effectively separate such cases. Furthermore, we can use minimal execution time for SW/HW co-design as the best what a given code can achieve on a given architecture, or expected execution time for realistic end-user program optimization and adaptation.

3.2 Adaptive exploration (sampling) and dimensionality reduction

Now, we can easily distribute exploration of any set of choices vs multiple properties and characteristics in a computer system among many users. For example, Figure 4(b) shows random exploration of Sourcery GCC compiler flags versus execution time and binary size on off-the-shelf Samsung mobile phones with ARMv7 processor for image processing codelet while Figure 5 shows exploration of dataset parameters for LU-decomposition numerical kernel on GRID5000 machines with Intel Core2 and SandyBridge processors. Since all characteristics are usually dependent, we can apply cM plugin (module) to detect universal Paretto fronter on the fly in multi-dimensional space (currently not optimal) during on-line exploration and filter all other cases. A user can choose to explore any other available characteristic in a similar way such as power consumption, compilation time, etc depending on usage scenario and requirements. In order to speed up random exploration further, we use probabilistic focused search similar to ANOVA and PCA described in [20, 28] that can suggest most important tuning/analysis dimensions with likely highest speedup or unusual behavior, and guide further finer-grain exploration in those areas. Collective exploration is critical to build and update a realistic training set for machine-learning based self-tuning meta-compiler cTuning-CC to automatically and continuously improve default optimization heuristic of GCC, LLVM, ICC, Open64 or any other compiler connected to cM [25, 6].

3.3 On-line learning

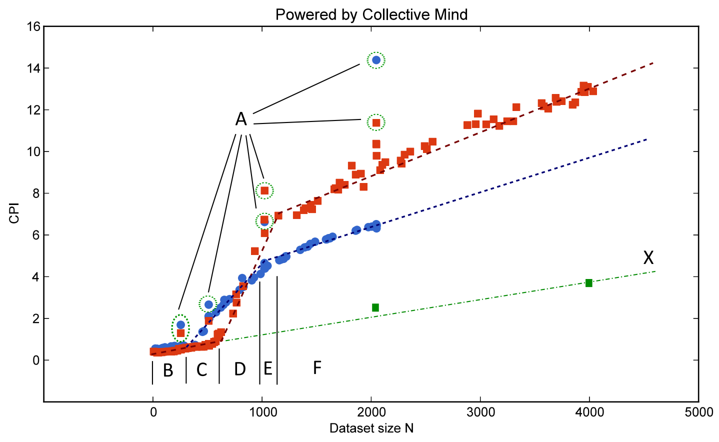


Figure 5: *On-line learning (predictive modeling) of a CPI behavior of ludcmp on 2 different platforms (Intel Core2 vs Intel i5) vs matrix size N and cache size*

Crowdtuning has a side effect - generation and processing of huge amount of data that is well-known in other fields as a "big data" problem. However, in our past work on online tuning, we showed that it is possible not only to learn behavior and find correlations between characteristics, properties and choices to build models of behavior on the fly at each client or program, but also to effectively compact experimental data keeping only representative or unexpected points, and minimize communications between cM nodes thus making cM a giant, distributed learning network to some extent similar to brain [24, 36, 28].

Figure 5 demonstrates how on-line learning is performed in our framework using LU-decomposition benchmark as an example, 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). At the beginning, our system does not have any knowledge about behavior of this (or any other) benchmark, so it simply observes and stores available characteristics while collecting as many properties of the whole system as possible (exposed by a researcher or user). At each step, system processes all historical observations using various available predictive models such as SVM or MARS in order to find correlations between properties and characteristics. In our example, after sufficient amount of observations, system can build a model that automatically correlated data set size N, cache size and CPI (in our case combination of linear models B-F that reflect memory hierarchy of a particular system) with nearly 100% prediction. However, system always continue observing behavior to continuously validate it against existing model in order to detect discrepancies (failed predictions). In our case, the system eventually detects outliers A that are due to cache alignment problems. Since off-the-shelf models rarely handle such cases, our framework allows to exclude such cases from modeling and expose them to the community through the unified Web services to reproduce and explain this behavior, find relevant features and improve or optimize existing models. In our case, we managed to fit a hybrid 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.

Systematic characterization (modeling) of a program behavior across many systems and data sets allows researchers or end-users to focus further optimizations (tuning) only on representative areas with distinct behavior while collaboratively building an on-line advice system. In the above example, we evaluated and prepared the following advices for optimizations: 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; area F and points A can benefit from reduced processor frequency to reduce power consumption using cM online adaptation plugin. Since auto-tuning is continuously performed, we will release final optimization details at cM live repository [1] during symposium.

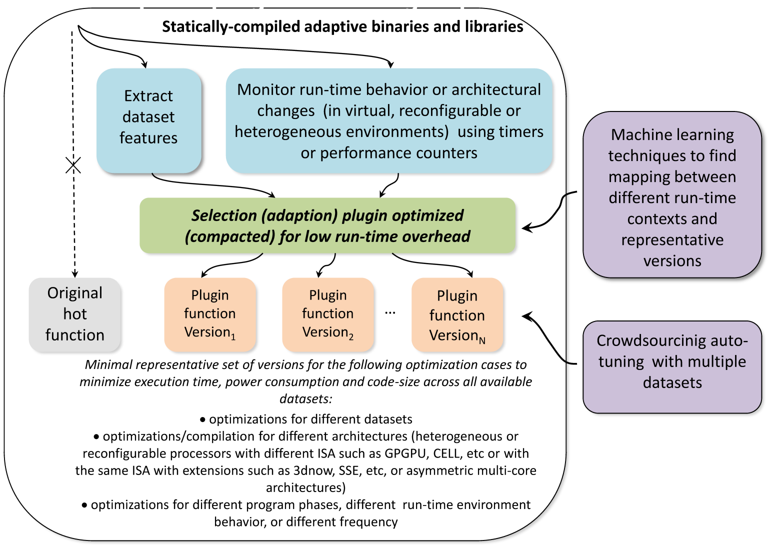

Statically-compiled adaptive binaries and libraries
Extract dataset features
Monitor run-time behavior or architectural changes (in virtual, reconfigurable or heterogeneous environments) using timers or performance counters
Machine learning techniques to find mapping between different run-time contexts and representative versions
Selection (adaption) plugin optimized (compacted) for low run-time overhead
Original hot function
Plugin function Version$_1$
Plugin function Version$_2$
...
Plugin function Version$_N$
Crowdsourcinig auto-tuning with multiple datasets
Minimal representative set of versions for the following optimization cases to minimize execution time, power consumption and code-size across all available datasets:
• optimizations for different datasets
• optimizations/compilation for different architectures (heterogeneous or reconfigurable processors with different ISA such as GPGPU, CELL, etc or with the same ISA with extensions such as 3dnow, SSE, etc, or asymmetric multi-core architectures)
• optimizations for different program phases, different run-time environment behavior, or different frequency

Figure 6: *Systematizing and unifying split (staged) compilation for statically built adaptive applications using crowdtuning and machine learning*

Gradually increasing and systematized knowledge in the repository in form of models can now be used to detect and characterize an abnormal program or system behavior, suggest future architectural improvements, or predict most profitable program optimizations, run-time adaptation scenarios and architecture configurations depending on user requirements. For example, this knowledge can be effecitvely used for split (staged) tuning to build static multi-versioning applications with cM plugins for phase-based adaptation [24] or predictive scheduling [32] in heterogeneous systems that can automatically adjust their behavior at run-time to varying data sets, environments, architectures and system state by selecting appropriate versions or changing frequency to maximize performance and minimize power consumption, while avoiding complex recompilation frameworks as conceptually shown in Figure 6.

3.4 Benchmark automatic generation and decremental characterization

Projects on applying machine learning to auto-tuning suffer from yet another well-known problem: lack of benchmarks. Our experience with hundreds of codelets and thousands of data sets [23, 16, 25] shows that they are still not enough to cover all possible properties and behavior of computer systems. Generating numerous synthetic benchmarks and data sets is theoretically possible but will result in additional explosion in analysis and tuning dimensions. Instead, we use existing benchmarks, codelets and even data sets as templates, and utilize Alchemist plugin [3] for GCC to randomly or incrementally modify them by removing, modifying or adding various instructions, basic blocks, loops, and thus generating. Naturally, we ignore crashing variants of the code and continue evolving only the working ones. Importantly, we use this approach not only to extend realistic training sets, but also to gradually (iteratively) identify various behavior anomalies and detect code properties to explain these anomalies and improve predicting modeling without any need for slow and possibly imprecise system/architecture simulator or numerous and sometimes misleading hardware counters as originally presented in [27, 21]. For example, we can iteratively scalarize memory accesses to characterize code and data set as CPU or memory bound [21] (line X in Figure 5 shows ideal codelet behavior when all floating point memory accesses are NOPed). Additionally, we use Alchemist plugin to extract code structure, patterns and other properties to improve our cTuning CC machine-learning based meta compiler connected to GCC, LLVM, Open64, Intel and Microsoft compilers, and to guide SW/HW co-design.

4 Conclusions and future work

With the continuously rising number of workshops, conferences, journals, symposiums, consortiums, networks of excellence, publications, tools and experimental data, and at the same time decreasing number of fundamentally new ideas and reproducible research in computer engineering, we strongly believe that the only way forward now is to start collaborative systematization and unification of available knowledge about design and optimization of computer systems. However, unlike some existing projects that mainly suggest or attempt to share raw experimental data and related tools, and somehow validate results by the community, or redesign the whole software and hardware stack from scratch, we use our interdisciplinary background and experience to develop the first to our knowledge integrated, extensible and collaborative infrastructure and repository (Collective Mind) that can represent, preserve and connect directly or semantically all research artifacts including data, executable code and interfaces in a unified way.

We hope that our collaborative, evolutionary and agile methodology, and extensible plugin-based Lego-like framework can help to address current fundamental challenges in computer engineering while bringing together interdisciplinary communities similar to Wikipedia to continuously validate, systematize and improve collective knowledge about designing and optimizing whole computer systems, and extrapolate it to build faster, more power efficient, reliable and adaptive devices and software. We hope that community will continue developing more plugins (modules) to plug various third-party tools including TAU [41], Periscope [13], Scalasca [29], Intel vTune and many others to cM, or continue gradual decomposition of programs into codelets and complex tools into simpler connected self-tuning modules while crowdsourcing learning, tuning and classifying of their behavior. We started building a large public repository of realistic behavior of multiple programs in realistic environments with realistic data sets (”big data”) that should allow the community to quickly reproduce and validate existing results, and focus their effort on developing novel tuning techniques combined with data mining, classification and predictive modeling rather than wasting time on building individual experimental setups. It can also be used to address the challenge of collaboratively finding minimal representative set of benchmarks, codelets and datasets covering behavior of most of existing computer systems, detecting correlations in a collected data together with combinations of relevant properties (features), pruning irrelevant ones, systematizing and compacting existing experimental data, removing or exposing noisy or wrong experimental results. It can also be effectively used to validate and compact existing models including roofline [47] or capacity [34] ones, and adaptation techniques including multi-agent based using cM P2P communication, classify programs by similarities in models, by reactions to optimizations [28] and to semantically non-equivalent changes [27], or collaboratively develop and optimize new complex hybrid predictive models that from our past practical experience can not yet be fully automated thus using data mining and machine learning as a helper rather than panacea at least at this stage.

Beta proof-of-concept version of a presented infrastructure and its documentation is available for download at [3], while pilot Collective Mind repository is now live at *c-mind.org/repo* [1] and currently being populated with our past research artifacts including hundreds of codelets and benchmarks [25], thousands of data sets [23, 16], universal compilation and execution pipeline with adaptive exploration (tuning), dimension reduction and statistical analysis modules, and classical off-the-shelf or hybrid predictive models. Importantly, presented concepts have already been successfully validated in several academic and industrial projects with IBM, ARC (Synopsys), CAPS, CEA and Intel, and we gradually release all our experimental data from these projects including unexplained behavior of computer systems and misbehaving models. Finally, the example of a Collective Mind Node to crowdsource auto-tuning and learning using Android mobile phones and tables is available at Google Play [2].

We hope that our approach will help to shift current focus from publishing only good experimental results or speedups, to sharing all research artifacts, validating past techniques, and exposing unexplained behavior or encountered problems to the interdisciplinary community for reproducibility and collaborative solving and ranking. We also hope that Collective Mind framework will be of help to a broad range of researchers even outside of computer engineering not to drawn in their experimentation while processing, systematizing, and sharing their scientific data, code and models. Finally, we hope that Collective Mind methodology will help to restore the attractiveness of computer engineering making it a more systematic and rigorous discipline [22].

Acknowledgments

Grigori Fursin was funded by EU HiPEAC postdoctoral fellowship (2005-2006) and by the EU MILEPOST project (2007-2010) where he originally developed and applied his statistical plugin-based crowdtuning technology to enable realistic and on-line tuning (training) of a machine-learning based meta-compiler cTuning CC together with MILEPOST GCC, and later by French Intel/CEA Exascale Lab (2010-2011) where he extended this concept and developed customized codelet repository and auto-tuning infrastructure for software and hardware co-design and co-optimization of Exascale systems together with his team (Yuriy Kashnikov, Franck Talbart, and Pablo Oliveira). Grigori is grateful to Francois Bodin and CAPS Entreprise for sharing codelets from the MILEPOST project and for providing an access to the latest Codelet Finder tool, to David Kuck and David Wong from Intel Illinois, and Davide del Vento from NCAR for interesting discussions and feedback during development of cTuning technology. Grigori is also very thankful to cTuning and HiPEAC communities as well as his colleagues from ARM and STMicroelectronics for motivation, many interesting discussions and feedback during development of Collective Mind repository and infrastructure presented in this paper. Finally he would like to thank GRID5000 project and community for providing an access to powerful computational resources.

References

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

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

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

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

[5] MILEPOST GCC: public collaborative R&D website. `http://cTuning.org/milepost-gcc`.

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

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

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

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

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

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

[12] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. 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, Dec. 2006.

[13] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. pages 1–16, 2010.

[14] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.

[15] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, March 2007.

[16] Y. Chen, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating iterative optimization across 1000 data sets. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI*, 2010.

[17] K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[18] K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.

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

[20] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.

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

[22] G. Fursin. HiPEAC thematic session at ACM FCRC'13: Making computer engineering a science. http://www.hipeac.net/thematic-session/making-computer-engineering-science, 2013.

[23] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.

[24] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.

[25] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. 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.

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

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

[28] G. Fursin and O. Temam. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):20:1–20:29, Dec. 2010.

[29] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, Apr. 2010.

[30] T. Hey, S. Tansley, and K. M. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

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

[32] V. Jimenez, I. Gelado, L. Vilanova, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2009)*, January 2009.

[33] T. Kisuki, P. Knijnenburg, and M. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 237–246, 2000.

[34] D. Kuck. Computational capacity-based codesign of computer systems. *High-Performance Scientific Computing*, 2013.

[35] 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.

[36] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'09), colocated with HiPEAC'09 conference*, January 2009.

[37] 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.

[38] 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.

[39] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the International Conference on Supercomputing*, 2004.

[40] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.

[41] S. S. Shende and A. D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

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

[43] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 77–90, June 2003.

[44] M. Tartara and S. Crespi-Reghizzi. Continuous learning of compiler heuristics. *TACO*, 9(4):46, 2013.

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

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

[47] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[48] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. 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.