Collective Mind, Part II: technical report

Grigori Fursin

Collective Mind, Part II: technical report

Grigori Fursin et.al.*†,
*INRIA, France

Abstract—**This is a technical report about practical usages of our recent Collective Mind framework.**

I. INTRODUCTION AND RELATED WORK

There is an impressive and ever increasing number of diverse computer systems available nowadays on the market. They all vary in performance, size, power consumption, reliability, price and other characteristics depending on numerous available hardware features such as processor architecture, number of cores, availability of specialized hardware accelerators, working frequency, memory hierarchy and available storage. As a result, engineers often have to develop software that may end up running across different, heterogeneous and possibly virtualized hardware in multiple mobile devices, desktops, HPC servers, data centers and cloud services. Such a rising complexity of computer systems and limited development time usually force software engineers to rely almost exclusively on existing compilers, operating systems and run-time libraries in a hope to deliver the fastest, smallest, most power efficient, scalable and reliable executable code across all available hardware. Unfortunately, this complexity of ever changing hardware also made development of compilers very challenging. They nowadays include hundreds of optimizations and often fail to produce efficient code while wasting expensive resources and energy [69], [18], [20], [41], [39].

Numerous auto-tuning, run-time adaptation, genetic and machine learning techniques (including our own) have been introduced in the past two decades to help software engineers optimize their applications for rapidly evolving hardware [69], [18], [25], [61], [58], [28], [68], [49], [35], [67], [29], [50], [65], [54], [52], [34], [62], [45], [22], [48], [19], [57], [60], [63], [59], [56]. These techniques usually demonstrate that it is possible to improve various characteristics of existing software by automatically and empirically searching and predicting better combinations of optimizations. Such optimizations typically include compiler flags, optimization parameters and their orders, different algorithm implementations, run-time scheduling policies, working frequency among many other choices. The resulting 10% to 10x performance improvements of frequently executed programs can already reduce usage costs of large data centers and supercomputer operators by thousands of dollars per year. These improvements can also considerably improve overall performance, battery life and storage space in mobile devices.

Nevertheless, in spite of so many promising results and research advances in the past twenty years, we still witness the rising number of reports, complaints, public discussions and research articles showing that existing compilers are missing many optimization opportunities for popular software [10], [8], [14]. This is caused by a fundamental and yet unsolved problem - design and optimization spaces are already too large and continue growing. Therefore, exhaustive exploration of the whole space to find the best solution is simply impossible. Practically all recent long-term research visions acknowledge above problem [21], [30], [15], [41], [17]. They even advocate that some radically new approaches should be invented to be able to continue building faster, more power efficient and reliable software and hardware by 2020 while somehow decreasing all development, optimization and usage costs.
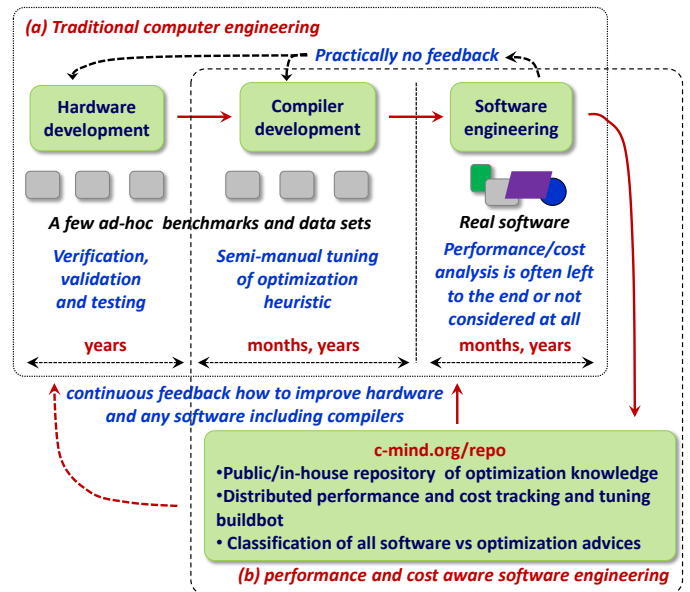

Fig. 1. (a) traditional computer engineering versus (b) our new collaborative performance and cost-aware software/hardware co-design as a web service.

Indeed, hardware and compiler designers can afford to explore only a tiny fraction of the whole optimization space together with a few ad-hoc benchmarks and s on a few architectures when tuning optimization and run-time adaptation heuristics or training predictive models. However, this simply can not anymore represent a vast amount of real-world software and hardware thus explaining why more and more optimization opportunities are so easily missed. Furthermore, static compilers often lack data set specific information and run-time adaptation mechanisms meaning that costly optimization process has to be restarted for practically every program input. Dynamic compilers and run-time auto-tuning techniques were not able to solve the problem either. Such just-in-time frameworks always have a very limited time budget to be able to react to varying execution contexts, and hence can not explore many aggressive optimizations. Worse, almost all vast practical optimization knowledge and experience from the software engineering community during their own attempts to analyze and tune program behavior is often wasted instead of helping to improve compilers and run-time systems as conceptually shown in Figure 1a.

Rather than waiting until 2020 in a hope that compiler and hardware designers will somehow find a "holy grail" solution, we propose an alternative and collaborative solution based on our interdisciplinary background in physics and AI. With the help of the software engineering community, we started gradually identifying real and frequently used software pieces ranging from the whole application to jut a few lines of code when more than one optimization (or implementation) choice is available. Performance and associated usage costs of such software pieces usually heavily depend on different optimizations for a given input (data set), utilized hardware, environment state, and possible end-user requirements (such as algorithm precision versus energy usage and performance).

We then propose to use our recent Collective Mind framework and Hadoop-based repository of knowledge (cM for short) [39], [4] to extract and share such open source software pieces together with various possible inputs and meta data at c-mind.org/repo. This meta data is gradually extended by the community via popular and human readable JSON format [12] currently describing how to build and run shared pieces together with all dependencies on the specific hardware and software including compilers, operating systems and run-time libraries. All these shared software pieces are then continuously and randomly optimized and executed with different data sets using distributed cM buildbot for Linux and Windows-based devices [4] or cM node for Android devices [3] across shared computational resources provided by volunteers. Such resources range from mobile phones, tablets, laptops and desktops to data centers, supercomputers and cloud services gradually covering all existing hardware configurations and environments. Furthermore, the community can use light weight cM wrappers around identified software pieces within a real and possibly proprietary applications to continuously monitor their behavior and interactions in the whole software project. Similar to nature and biological species, such approach treats all exposed and shared software pieces as *computational species* while continuously tracking and learning their behavior versus different optimizations across numerous hardware configurations, realistic software environments and run-time conditions. cM infrastructure then continuously records only the winning solutions (optimizations for a given data set and hardware) that minimize all or only monitored costs (execution time, power consumption, code size, failures, memory and storage footprint, and optimization time) of a given software piece on a Pareto frontier [51] in our public cM repository.

Software engineers can now assemble their projects from the cM plugins with continuously optimized computational species. Such software projects can continuously and collaboratively achieve better performance while reducing all costs across all hardware thus making software engineering performance- and cost-aware. Furthermore, software developers are now able to practically help compiler and hardware designers improve their technology as conceptually shown in Figure 1b. Indeed, our approach helps create the first to our knowledge public, realistic, large, diverse, distributed, evolving and continuously optimized benchmark with related optimization knowledge while gradually covering all possible software and hardware.

At the same time, we can also apply an extensible, top down methodology originating from physics when learning behavior of complex systems. The compiler community first learns and optimizes coarse grain behavior of large shared software pieces including whole applications, library functions, kernels and most time consuming loops versus global compiler optimization flags or other coarse-grain optimizations. After enough knowledge is collected, the community can gradually move to finer grain levels including just a few source lines or binary instructions versus all internal and individual compiler optimization decisions via our Interactive Compilation Interface. This plugin-based interface is already available in mainline GCC [38], and we plan to add it to LLVM in the future [39].

More importantly, our approach helps considerably improve existing methodology on optimization and run-time adaptation prediction using machine learning. Current methodology (used in most of the papers including ours and referenced at the beginning of this section) usually focuses on *showing that it is possible to predict* one or several optimizations to improve execution time, power consumption or some other characteristics using some off-the-shelf machine learning techniques such as SVM, (deep) neural networks or KNN [23], [43], [53] combined with a few ad-hoc program or architecture features. In contrast, our growing, large and diverse benchmark allows the community for the first time to apply methodology from sciences such as biology, medicine and AI based on big data predictive analytics [42]. For this purpose, cM infrastructure continuously classifies all winning species in terms of distinct optimizations and exposes them to the community in a unified and reproducible way through the public repository. This, in turn, allows our colleagues with interdisciplinary background to help the software engineering community *find best predictive models* for these optimization classes together *with relevant features from software species, hardware, data set and environment state* either manually or automatically. Such features (including extraction tool) and predictive models are continuously added to the species using cM wrappers and their meta-data thus practically enabling self-tuning software automatically adaptable to any hardware and environment.

Importantly, cM continues tracking unexpected behavior (abnormal variation of characteristics of species such as execution time, or mispredictions from current classification) in a reproducible way in order to allow the community improve predictive models and find missing features that can explain such behavior. Also, in contrast with using more and more complex and computationally intensive machine learning techniques to predict optimizations such as deep neural networks [23], [43], [53], we decided to provide a new manual option useful for compiler and hardware designers. This option allows the community to combine existing predictive techniques as a cheap way to quickly analyze large amount of data, with manually crafted human-readable, simple, compact and fast rules-based models (decision trees) that can explain and predict optimizations for a given computational species. Thus, we are collaboratively building a giant optimization advice web service that links together all shared software species, optimizations and hardware configurations while resembling Wikipedia, IBM Watson advice engine [33], Google knowledge graph [9] and a brain.

We understand, that the success of our approach will depend on the active involvement from the community. Therefore, we tried to make our approach as simple and transparent to use as possible. For example, our light-weight cM version for Android mobile systems [3] is a "one-button approach" allowing anyone to share their computational resources and tune shared computational species. At the same time, extraction of software pieces from large applications is still semi-manual and may incur some costs. Therefore we are gradually working on automating this process using plugin-based capabilities in GCC and LLVM. Furthermore, together with participating companies and volunteers, we already extracted, described and partially [1] shared 285 computational species together with around 500 input samples [2] from major benchmarks and software projects. We then validated our approach in STMicroelectronics during 3 months to help our colleagues tune their production GCC compiler and improve real customer software. During that time, we continuously optimized execution time, code size, compilation time and power consumption of all shared computational species using at least 5000 random combinations of compiler optimization flags on spare private cloud servers and mobile phones. We also managed to derive 79 distinct optimization optimization classes covering all shared species that we correlated with program semantic and dynamic features using SVM and other predictive analytics techniques. With the help of domain specialists (compiler engineers), we then analyzed predictive models for end-user software, found meaningless correlations, manually isolated problems [3], prepared and shared counter-example code sample, found missing program and input features to fix wrong classifications, and developed adaptive, self-tuning and statically compiled code. Finally, we managed to substitute ad-hoc benchmark used at the architecture verification department of our industrial partners with the minimal and realistic one based on derived optimization classes that helped to dramatically reduce development and testing time.

These positive outcomes demonstrate how our approach can help eventually involve the software engineering community into development and improvement of compilers and hardware. We also show how continuously growing collective knowledge repository accessible via unified web service can become an integral part of the practical software and hardware co-design of self-tuning computer systems while decreasing all development costs and time-to-market for new products. More importantly, the side effect of our approach to share code and data in a reproducible way help support recent international initiatives on reproducible research and sustainable software engineering [2].

Our paper is organized as follows. This section has introduced the problem, related work and our novel community-driven approach for performance- and cost-aware software

engineering. It is followed by Section II presenting our personal and real-life motivating software engineering example for neural networks with some of the encountered optimization issues during past 15 years. Next section III briefly introduces our recent open source Collective Mind infrastructure and repository to enable sharing of computational species and collaborative tracking and tuning of their performance together with all associated costs across voluntarily provided computer systems. Section IV demonstrates how we continuously systematize "big performance data" collected by cM, classify species, and predict optimizations while creating a realistic and representative benchmark. It is followed by Section V which demonstrates how to understand machine learning and improve optimization predictions. Section VI demonstrates how to find missing features to explain unexpected behavior of computational species and improve optimization predictions. It also shows how to build adaptive and self-tuning applications assembled from available computational species as plugins. Finally, we conclude paper and describe future research and development directions in Section VII.

II. Real-life motivating example

One of the authors has a hobby started more than twenty years ago developing and analyzing various artificial neural networks as a part of a possible non-traditional and brain-inspired computer. Such networks can mimic brain functions and are often used for machine learning and data mining [23]. For example, Figure 2 shows one of the oldest and well-known one-layer, fully interconnected, recurrent (with feedback connections) Hopfield neural network [44]. It is a popular choice for function modeling, pattern recognition and image filtering tasks including noise reduction. Implemented as a software, this neural network have a fairly simple and regular code where each neuron receives a weighted sum of all inputs of an image as well as outputs of all other neurons. This sum is then processed using some neuron activation function including sigmoid or linear ones to calculate the output value. The small and simple C kernel presented in above Figure 2 is one of many possible implementations of a threshold filter we used as a part of a linear activation function, i.e. switching neuron output from 0 to 1 when its input meets a given threshold.

Very simplistically, the quality of a neural network is usually determined by its processing speed as well as capacity (maximum amount of patterns or information that can be stored in such networks) and recognition accuracy (correct predictions versus failures). It heavily depends on the total number of neurons, connections and layers [46], and is primarily limited by the speed and resources of the available hardware including specialized accelerators. Hence, neural network software/hardware co-design process always involves careful balancing of performance versus all associated costs including storage size, memory footprint, energy consumption, development time and hardware price depending on usage scenarios and required time to market. Indeed, our research on improving neural networks requires many iterative runs of a slightly evolving modeling software with varying parameters to maximize prediction accuracy. In this case, our main concern is about minimizing compilation and execution time of each execution across available hardware. However, when the best found network is found and deployed in a large data center or cloud service (for example, for big data

---

[1] We can not share extracted pieces from proprietary software but we still use them internally for research purposes.

[2] We currently have more than 15000 input samples collected in our past projects for our shared computational species [37], [11], [27]. However since they require more than 17GB of storage, at the moment we decided to share only representative ones, i.e. which require distinct compiler optimization.

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

Image filters: *noise reduction*
B&W threshold

Classifiers: *pattern recognition*

3?
yes / no

$$y_i = \begin{cases} \text{`1'} & \text{if } \sum_j w_{ij} x_j > \theta_i \\ \text{`0'} & \text{otherwise} \end{cases}$$

```
v1 = *matrix_ptr1++;
for (i = 0; i < N * N; i++) {
    tmp1 = abs(v1);
    *matrix_ptr2++ =
        (tmp1 > T) ? 1 : 0;
    v1 = *matrix_ptr1++;
}
```

Neuron activation function

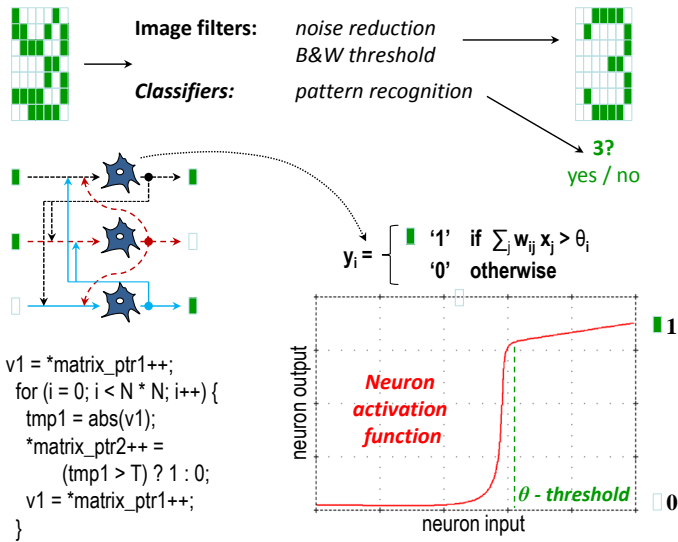neuron output

1

θ - threshold

0

neuron input

Fig. 2. Conceptual example of pattern recognition, image filtering and character noise reduction using Hopfield fully interconnected and recurrent neural network. Simple C kernel is a part of a neuron activation function processing thresholds for all neurons.

analysis), end users would like to minimize all additional costs including energy and storage consumption across all provided computer systems. Finally, when deploying neural networks in small, autonomic and possibly mass-produced devices such as surveillance cameras and mobile phones or future robots and Internet of Things objects, more strict requirements are placed on software and hardware size, memory footprint, real time processing, and the cost of the whole system.

Twenty years ago, our software engineering of neural networks was relatively straightforward. We did not have a choice but to simply select the latest hardware with the accompanying and highly tuned compiler to achieve nearly peak performance for our software including for the code shown in Figure 2. Therefore, in order to innovate and process more neurons and their configurations, we usually had to wait for more than a year until arrival of a new hardware. This hardware would likely double performance of our software and provide more memory and permanent storage but often at a cost of higher power consumption and thus dramatically rising electricity bill.

In contrast, we now have an impressive choice of hardware of all flavors which our software can be executed on. Each year, there are numerous variations of processors appearing on the market with different features (properties) including frequency, number of cores, cache size, ISA extensions, specialized hardware accelerators (such as GPU and even revived semiconductor neural networks), power consumption and price. Furthermore, we can now have an easy access to large-scale parallel resources from home via gaining popularity virtualized cloud services from Amazon, Google, Microsoft and others. Therefore, the number of experiments we can now run is mainly limited by the price we can afford to pay for computing services. At the same time, we also enjoy continuous community-driven improvements of operating systems together with numerous free or proprietary libraries and software development tools including popular optimizing compilers such as GCC and LLVM. One may expect that with so many advances

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

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

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

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

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

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

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

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

(a)

1) P1 O3 W2 X1 Y1 T2 D1
2) P1 O3 W2 X7 Y1 T2 D1
3) P1 O3 W2 X1 Y7 T2 D1
4) P1 O3 W2 X7 Y5 T2 D1
5) P1 O3 W2 X11 Y1 T2 D1
6) P1 O3 W2 X9 Y1 T2 D1
7) P1 O3 W2 X3 Y7 T2 D1
8) P1 O3 W2 X4 Y8 T2 D2
9) P1 O1 W1 X14 Y1 T3 D1
10) P1 O1 W1 X13 Y1 T2 D1
11) P1 O3 W2 X7 Y8 T2 D2

A) P3 O5 W1 X1 Y1 T4 D1
B) P3 O5 W1 X4 Y1 T4 D1
C) P3 O5 W1 X4 Y7 T4 D1
D) P3 O5 W1 X6 Y1 T4 D1
E) P3 O5 W1 X6 Y7 T4 D1
F) P3 O5 W1 X9 Y1 T4 D1

I) P2 O4 W1 X1 Y1 T4 D1
II) P2 O4 W1 X6 Y1 T4 D1

$) P4 O3 W1 X12 Y1 T1 D1

Program execution time (sec.)

Program binary size (bytes)

Available resource: P1, one core
Available resources: P1 and P3
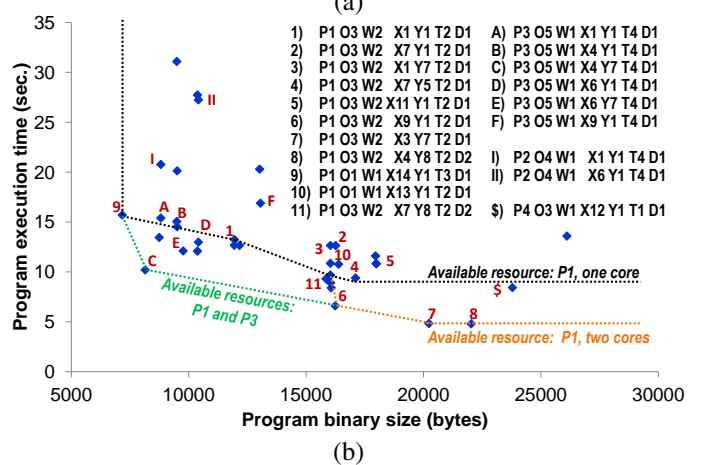*Available resource: P1, two cores*

(b)

Fig. 3. (a) A small subset of various hardware, software, development tools and optimizations used in our research on neural networks in the past 20 years (P - processors, W - processor mode, X - compiler, O - operating system, S - system, T - total number of processed pixels or neurons, D - software data set, Y - compiler optimization used) (b) 2D projection of the multidimensional space of characteristics together with winning solutions on the Pareto frontier (all data and interactive graphs are available at c-mind.org/nnet-tuning-motivation).

in the computer technology, practically any recent compiler would generate the fastest and most energy efficient code for such an old, simple, small and frequently used software piece shown in Figure 2 across existing hardware. Nevertheless, since we pay for our experiments, we eventually decided to validate their performance/cost efficiency.

For the sake of accountability and reproducibility, we started gradually collecting at c-mind.org/nnet-tuning-motivation various information about several computer systems we used including their price, cost, available operating systems, compilers and optimizations. Figure 3a shows a tiny subset of this multidimensional space of design and optimization choices. At the same time, whenever running real experiments, we also started recording their execution time [4] and all associated costs including compilation time,

---

[4]In this paper, for the sake of simplicity and without loosing generality, when speaking about performance tuning, we mean reducing execution time. However, on modern out-of-order processors with complex memory hierarchy, the dependency between performance (speed of execution) and total execution time may be non-linear. Thus, depending on user requirements, these characteristics have to be tuned separately.

code size, energy usage, software/hardware price and utility bill. At the same time, we decided to perform a simple and well-known optimization compiler flag auto-tuning [1], [38] with at least 100 iterations to see whether there is still room for improvement over the fastest default compiler optimization level (-O3). Figure 3b shows one of many possible 2D projections of the multidimensional space of characteristics (which we consider as costs of running our experiments or tasks). We then gradually track the winning solutions that maximize performance and at the same time minimize all costs using our experience in physics and electronics, namely by applying Pareto frontier filter [51].

We quickly realized that in contrast to the traditional wisdom, the latest technology is not necessarily the fastest or most energy efficient and further optimization is always required. For example, when moving from GCC 4.1.1 (released in 2006) to GCC 4.9.1 (released in 2014), we observed a modest 4% improvement [5] in single core execution time of our neural network and 2% degradation in a code size on Intel E6320 based system (released in 2008). However, 8 years old GCC 4.1.1 can achieve 27% improvement in execution time after auto-tuning (which comes at cost of 100 recompilations and executions as well as increasing binary size by 34%)! Interestingly, 8 years old PathScale 2.3.1 produces faster code than the latest version of GCC 4.9.1 and LLVM 3.4.2! Furthermore, when using internal parallelization, LLVM 3.4.2 beats GCC 4.9.1 by about 23% but has a sub-linear scaling versus number of threads. In contrast, 2 years old GCC 4.6.3 achieves the best result and linear scaling versus number of threads when using both parallelization and auto-tuning!
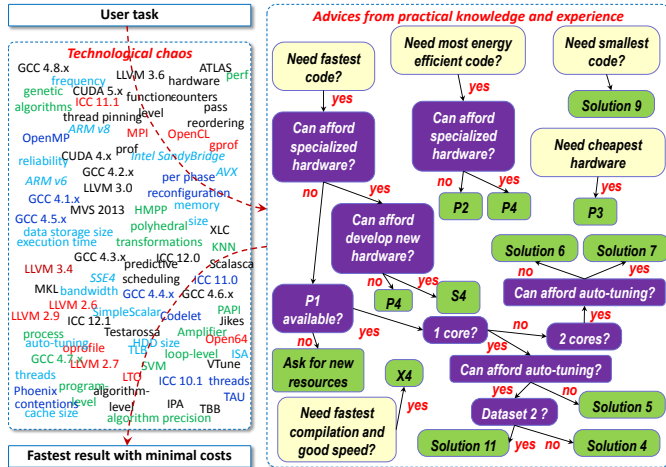

Fig. 4. Example of gradually and manually crafted advices as decision trees to deliver best performance and cost for our neural network depending on available resources, usage scenarios (requirements) and data sets.

When running the same code on cheap, commodity mobile phones with ARM architecture, the execution time increased dramatically by around 5 times! However, the power consumption dropped by about 10 times! When trying to use specialized hardware (GPUs or our semiconductor neural networks), we

could increase execution time by about tens to hundreds of times, but at a considerable development cost and time to market. Furthermore, with time, we discovered that the same best found optimization for one class of images can considerably degrade performance on another class of images (as we explain in Section VI). We also encountered problems with cache contentions on multi-core systems, sub-linear scaling on many core systems, unexpected frequency scaling, nondeterministic IO for large images, and many other problems that had to be addressed by new optimizations. These issues can not be easily solved by static compilers due to a fundamental problem of a lack of run-time information at compile time. Therefore, we even tried to move to dynamic and possibly adaptive languages including Java and Python but were not yet able to achieve similar performance while spending even more energy and storage during just-in-time compilation.

Sadly and similar to many other scientists and software engineers, we now have to waste considerable amount of our time on a tedious and ad-hoc navigation through the current technological chaos to find some good hardware, software and optimization solutions that can speed up our programs and reduce costs instead of innovating as conceptually summarized in Figure 4. Worse, software engineers are often not even aware of all available design and optimization choices they have to improve performance of their software and reduce development and usage costs. Therefore, we strongly believe that current performance- and cost-blind software engineering has be changed to improve productivity and boost innovation in science and technology.

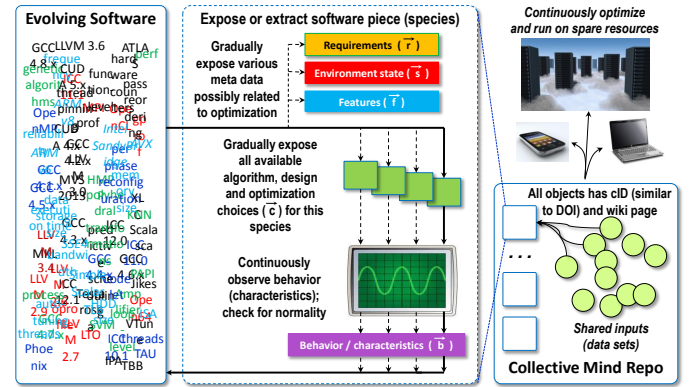## III. PUBLIC AND OPEN SOURCE COLLECTIVE MIND INFRASTRUCTURE AND REPOSITORY

Fig. 5. Collective Mind Framework and Repository (cM) help to decompose any complex software into pieces with light-weight wrappers that expose design and optimization choices, measured characteristics, features and environment state in a unified and mathematical way. It was developed to unify and systematize software auto-tuning, make it practical and reproducible, and distribute it among numerous computing resources such as mobile phones and data centers shared by volunteers [39], [4].

Eventually, we started searching for a possible solution that could liberate software developers from the tedious and not necessarily relevant job of continuous optimization and accounting while gradually making existing software performance- and cost-aware. At first, we tried to create a

---

[5]Similar to physics, we execute optimized code many times, check distribution of characteristics for normality [32], and report expected value if variation is less than 3%

simple database of optimizations and connect it to some existing benchmarking and auto-tuning tools to keep track of all optimizations [36], [38]. However, when trying to implement it within production environments of our industrial partners, we faced several severe problems including difficulty to expose all design and optimization choices from continuously evolving software, and difficulty to reproduce performance numbers collected from different machines. This eventually pushed us to develop a full-fledged repository of knowledge with unified web services (Collective Mind or cM for short) similar to ones that helped successfully systematize research and experimentation in biology, genomics and other natural sciences. Such repository should be able to keep the whole auto-tuning setups with all dependencies including optimized software, data sets and auto-tuning tools. This, in turn, should allow us to distribute the whole auto-tuning setups among many users to crowdsource software optimization (or any other experimentation) in a reproducible way while considerably reducing usage costs.

Briefly [6], cM helps decompose software into standalone pieces interconnected through cM wrappers. Such light-weight wrappers currently support major languages including C, C++, Fortran, Python, PHP and Java, and allow the community to gradually expose various design and optimization choices **c**, features **f**, dependencies on other software and hardware, monitored characteristics (costs) **b** and environment state **s** in a unified way through extensible JSON format [12]. This allowed us to formalize almost all existing auto-tuning techniques as finding a function of a behavior of a given software piece *B* running on a given computer system with a given data set, selected hardware design and software optimization choices **c**, and a system state **s** ([39]):

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

Furthermore, software pieces can be extracted and then shared together with their wrappers and data sets samples in the Hadoop-enabled [7] cM repository. For example, with the help of our colleagues and supporters, we already gradually and semi-automatically extracted and shared 285 software pieces together with several thousand data set pairs from several real software projects as well as 8 popular benchmark suits including NAS, MiBench, SPEC2000, SPEC2006, Powerstone, UTDSP and SNU-RT. This can liberate software engineers from developing their own ad-hoc and complex tuning setups in favor of implementing common auto-tuning pipelines consisting of shared software pieces, data sets, tools and optimization space exploration modules. Such pipelines can then be easily shared and distributed across a large number of diverse computer systems either using open source *cM buildbot* or a small *cM node* that can deploy experiments on Android-based devices [3]. cM will then continuously "crawl" for better optimizations for all shared software pieces, data sets and compilers, while recording experiments in a reproducible way in the public cM repository c-mind.org/repo.

As requested by our industrial partners, we first used cM to implement compiler optimization flag auto-tuning pipeline. Though this pipeline continues growing and now

-O3 -fif-conversion *-fno-ALL*
-O3 --param max-inline-insns-auto=88 -finline-functions *-fno-ALL*
-O3 -fregmove -ftree-vrp *-fno-ALL*
-O3 -fomit-frame-pointer -fpeel-loops -ftree-fre *-fno-ALL*
-O3 -falign-functions -fomit-frame-pointer -ftree-ch *-fno-ALL*
-O3 -ftree-dominator-opts -ftree-loop-optimize -funswitch-loops *-fno-ALL*
-O3 -ftree-ccp -ftree-forwprop -ftree-fre -ftree-loop-optimize *-fno-ALL*
-O3 -finline-functions -fivopts -fprefetch-loop-arrays -ftree-loop-optimize -ftree-vrp *-fno-ALL*
-O3 -fdce -fgcse -fomit-frame-pointer -freorder-blocks-and-partition -ftree-reassoc -funroll-all-loops *-fno-ALL*

TABLE I.   EXAMPLE OF A FEW TOP RANKED OPTIMIZATION CLASSES FOR GCC 4.6.3 ACROSS INTEL E5520 BASED COMPUTER SYSTEMS FROM THE FRENCH GRID5000 PUBLIC DATA CENTER [40] USING CM BUILDBOT WITH ALL SHARED SOFTWARE PIECES AND DATA SETS.

supports other optimizations including polyhedral source-to-source transformations, MPI/OpenMP tuning and fine-grain compiler transformations as can be seen at c-mind.org/ctuning-pipeline, compiler flag optimization problem remains critical for new hardware and unsolved for several decades due to growing optimization spaces. Indeed, latest GCC, LLVM and Intel compilers feature hundreds of optimizations which have to be carefully orchestrated for all existing software and hardware. At the same time, it should be simple enough to demonstrate the concepts of our approach which follows top-down optimization methodology. Therefore, we start from coarse-grain optimizations and gradually move to finer-grain levels motivated by our experience in physics when gradually moving from three Newton laws to quantum mechanics.

Our auto-tuning pipeline randomly selects a software piece from all shared ones, builds it with randomly generated combinations of compiler optimization flags of format *-O3 -f(no-)optimization_flag --parameter param = random_number_from_range*, and runs it on a randomly selected spare computer system with a randomly selected shared data set. Measured characteristics (costs) are then processed using Pareto frontier filter to record or update the winning solution for a given software piece (optimization versus hardware, compiler and data set) in the cM repository. For example, table I presents some of the winning combinations of flags for GCC 4.6.3 used in the production environment by one of our industrial partners. Note that meta flag *-fno-ALL* means that all other optimization flags have been gradually switched off to leave only the influential ones and thus help compiler designers understand missed optimization opportunities. This allows developers to continuously monitor and reuse best solutions for shared software pieces through unified cM web services depending on their further intended usage. For example, the fastest and energy efficient solution is often used for HPC systems, the smallest - for computer systems 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 phones, tablets and other mobile devices.

IV.   CLASSIFYING COMPUTATIONAL SPECIES

Though cM helped simplify, unify, automate, speed up and preserve a tedious process of hardware benchmarking and software auto-tuning, it also encountered a new serious problem. Even with a modest number of 400 volunteers

participated in our project since the beginning of 2014 to crowdsource compiler auto-tuning using our Android-based cM node [3], we immediately faced a big data problem - too much collected statistics that our repository and infrastructure can process. However, we have already faced a similar problem in natural sciences and AI for many years and managed to effectively tackle it using predictive analytics (statistical analysis, data mining and machine learning) [23], [43], [53], [42]. For example, we used to classify numerous physical objects or biological species in terms of behavior and features while leaving only representative classes and thus considerably reducing analysis time and required storage.

We propose a similar approach for software engineering. Collective Mind Framework, Repository and mathematical formalization allows us to treat all shared software pieces as computational species running across numerous hardware configurations. Naturally, these species will behave differently depending on their data sets, hardware used and environment state (which includes the state of the species, Operating System and hardware as well as interaction with other species). Since our main goal is to help software engineers find better optimizations that improve overall behavior of their software pieces under all usage conditions (reduce execution time and all other costs), we implemented continuous clustering of all available species that share the same top performing optimizations such as ones shown in Table I. Our expectation is that software pieces belonging to the same cluster, i.e. share similar optimizations, will also share some features describing their semantics, data set and run-time behavior. This, in turn, would allow us to automatically classify previously unseen computational species from the community and relate them to existing optimization cluster based on their features thus effecitvely predicting optimizations and dramatically reducing tuning time and storage size.

However, the major challenge with such approaches is to assemble a large and diverse enough training set, and to find meaningful features that can correctly separate optimization clusters. Collective Mind Repository helps solve the first problem by collecting a large number of real and diverse software pieces from the community. At the same time, from our practical experience in using machine learning for program optimization and hardware designs [38], [31], we believe that it is not currently possible to fully automate this process due to a practically infinite feature space. Instead, we propose to use a collaborative approach similar to ones currently used in many natural sciences. We expose optimization clusters together with software species and data sets through our public cM repository in a reproducible way to let the interdisciplinary community find and share meaningful features and correlations similar to ones shown in Figure 4. This approach contrasts with some existing works on machine learning for compilation and architecture. They were referenced in the related work section and mainly focus on showing that machine learning can be used to predict optimizations while using just a few benchmarks, data sets and ad-hoc features.

Our approach helped our industrial partners tune and cluster all shared software species across several production compilers and architectures. Next, we present a small subset of our experimental results for GCC 4.6.3 and Intel E5520. All other results are continuously updated at our public cM
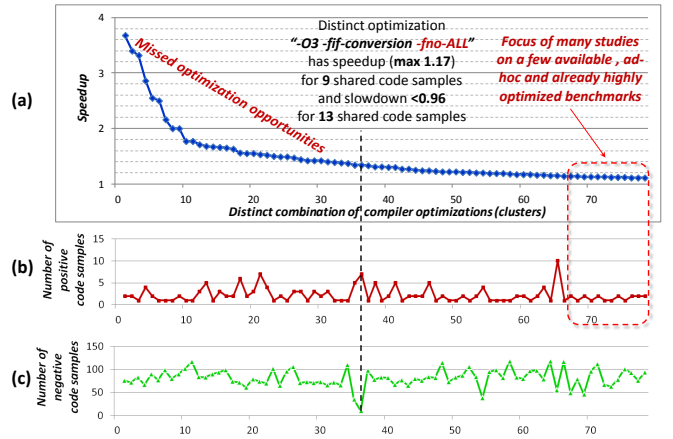

Fig. 6. (a) 79 distinct combinations of optimizations (optimization clusters) covering all 285 shared software species and data set samples on Intel E5520, GCC 4.6.3 and at least 5000 random combinations of optimization flags together with maximum speedup achieved within each optimization cluster; (b) number of benchmarks with speedup at least more than 1.1 for a given cluster; (c) number of benchmarks with speedup less than 0.96 (slowdown) for a given cluster.

Number of species	Prediction accuracy
12 from prior work [38]	87%
285 from current work	56%

TABLE II. PREDICTION ACCURACY WHEN USING OPTIMIZED SVM WITH FULL CROSS-VALIDATION FOR 12 AND 285 SOFTWARE SPECIES TOGETHER WITH ALL AVAILABLE SEMANTIC AND DYNAMIC FEATURES FROM MILEPOST GCC AND HARDWARE COUNTERS RESPECTIVELY.

repository at c-mind.org/repo. By now, cM has found a pool of 79 distinct combinations of optimization flags (clusters) for this compiler covering all shared software pieces and data set samples. Figure 6 shows maximum speedups achieved for each top performing and representative optimization across all benchmarks together with the number of benchmarks which achieve highest speedup using this optimization (or at least more than 1.1) and the number of benchmarks with speedups less than 0.96 (slowdown) for the same optimization. For example, distinct combination of optimizations -*O3 -fif-conversion -fno-ALL* achieved maximum speedup on 9 benchmarks (including 1.17 speedup on at least one of these benchmarks) and slowdowns for 13 benchmarks. In case some optimization cluster does not have slowdowns across all shared software pieces, it can help automatically substitute a best and often manually crafted default compiler optimization level such as -O3, thus practically helping to improve existing compilers.

V. UNDERSTANDING AND IMPROVING MACHINE LEARNING TO PREDICT OPTIMIZATIONS

As the first approximation, we decided to reuse existing machine learning techniques and build a predictive model (classifier) that can associate any previously unseen species with a unique optimization cluster based on some software features. We also decided to reuse and validate already available semantic and dynamic features from our previous work on machine learning based compiler [38]. For this purpose, we generated and shared in the cM repository a feature vector **f**

for each software species using 56 semantic program features from the MILEPOST GCC [38] (extracted during compilation at *-O1* optimization level after *pre* pass) and 30 dynamic features (hardware counters collected by default using standard performance monitoring tool *perf* when running unoptimized software piece on a shared computing resource with Linux-based OS). We then passed 79 optimization clusters, 86 features and either all 285 shared species or a small subset of 12 ones used in some of our past works through a standard SVM classifier from R package [23], [16] with full cross-validation as described in [38], [26].
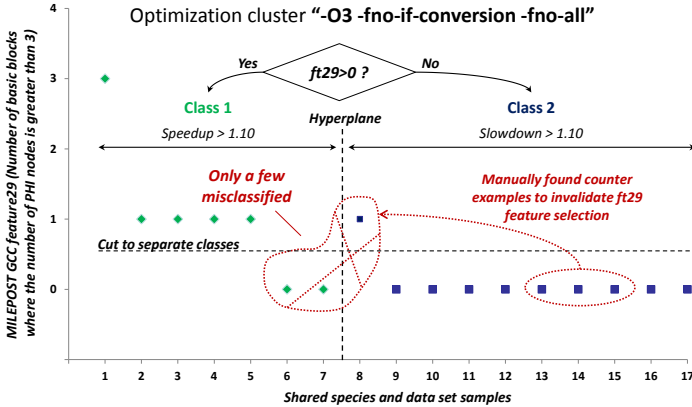

Fig. 7. Visualization of the predictive model built by SVM to predict optimization cluster "-O3 -fno-if-conversion -fno-ALL" using MILEPOST GCC features. We provided several counter examples to show that automatically found correlation can be totally misleading.

Results shown in Table II demonstrate that we can obtain a relatively high prediction accuracy when using just a few software species thus supporting findings from similar works on machine learning for auto-tuning, i.e. that it is possible to predict optimizations using machine learning. However, with a new opportunity provided by cM to use a much larger training set of shared software species from the community, prediction accuracy dramatically dropped exhibiting close to random behavior (50%), i.e. with no meaningful predictions. At the same time, Collective Mind approach can now help understand such a variation in prediction accuracy by exposing all optimization and modeling results in a reproducible way to experts. Our colleagues and compiler specialists from STMicroelectronics analyzed one of the simplest optimization clusters *-O3 -fif-conversion -fno-ALL* which has a relatively high prediction rate when using a few or all shared programs, i.e. only 3 mispredictions out of a pool of 17 optimization clusters (7 positive speedups and 10 negative ones) as shown in Figure 6. We then incrementally removed all unrelated features that did not influence predictions leaving only one semantic feature from MILEPOST GCC (ft29) that counts the number of basic blocks where the number of phi-nodes is greater than 3. Visualization of the predictive model at Figure 7 shows that SVM derived a decision ft29 $\rangle$ 0 to effectively separate two classes with only 3 mispredictions out of 17.

At this stage, many existing academic works will conclude that relevant feature is found and it is possible to use machine learning to predict optimization. However, compiler or hardware designers also need to understand whether this feature makes sense in order to improve their technology. Since our

colleagues did not manage to explain this feature, they decided to try to find a counter example to invalidate this correlation. Therefore, they selected a simple *blocksort function* from the shared *bzip2* species that has 0 phi-nodes and tried to manually add phi-nodes by converting source code as following (added lines are highlighted):

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

This manual transformation added 3 PHI nodes to the code changing MILEPOST feature *ft29* from 0 to 1, but without any effect on the speedup. We then performed similar transformation in a few other species that did not influence the original speedup while changing *ft29* to non-zero value and thus invalidating original decision, i.e. showing that *ft29* is not relevant to a given optimization cluster.

When exposing this problem to machine learning specialists, we realized that a high prediction accuracy can be explained by finding meaningless correlations in a very large and sparse feature space for just a few species that are invalidated on a very large and diverse training set. Though relatively naive, this example highlights the importance of creating a common repository of software species together with their features. Furthermore, this example shows that negative results (mispredictions or unexpected behavior), usually overlooked by our community, are very important in practice and should also be reported, shared and published to improve machine learning techniques. For example, our colleagues from STMicroelectronics shared their counter examples as new software species thus contributing to a public and realistic benchmark.

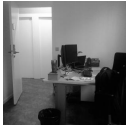## VI. LEARNING DATA SET FEATURES TO ENABLE ADAPTIVE SOFTWARE

Dataset Class	Optimization class	
	Class 1	Class 2
Shared data set sample₁	**+21.5%** ± 1.5% improvement in execution time	**-8.2%** ± 1.5% degradation in execution time
Shared data set sample₂	**-11.9%** ± 1.5% degradation in execution time	**+17.3%** ± 1.5% ı mprovement in execution time

Fig. 8. Detecting missing data set feature "time of the day" with the help of the community. Such feature enables adaptive software species that performs well across all inputs.

Though we demonstrated how our approach and methodology can help automate classification of shared software species to improve optimization predictions, it still did not solve another fundamental problem of static compilation - lack of run-time information. On the other hand, since cM continuously

records unexpected behavior, it helped to automatically detect that one of the real customer's software species (image B&W threshold filter from a surveillance camera application similar to one shown in Figure 2) requires two distinct optimizations with around 20% improvement in execution time on Intel Core i5-2540M across all shared images (data set samples) as shown in Figure 8.

In order to understand such behavior, we can now reuse the same clustering methodology to classify available data sets and detect their features that can explain such behavior and separate optimization classes. Compiler designers again helped us analyze this software species and gradually identified a suspicious "sub-species" causing unusual behavior: *(temp1 〉 T) ? 255 : 0*. One optimization class included "if conversion" transformation which added several predicated statements that may degrade performance if additional branches are rarely taken due to a few additional useless cycles to check branch condition. At this stage, compiler designers concluded that it is a well-known run-time dependency which is difficult or even impossible to solve in static compilers. Nevertheless, one of the volunteers noticed that some images shown in Figure 8 where taken during the day and some during the night. This helped us find new, simple and relevant feature related to both data set and the environment state *"time of the day"* that effectively separated two optimization classes.
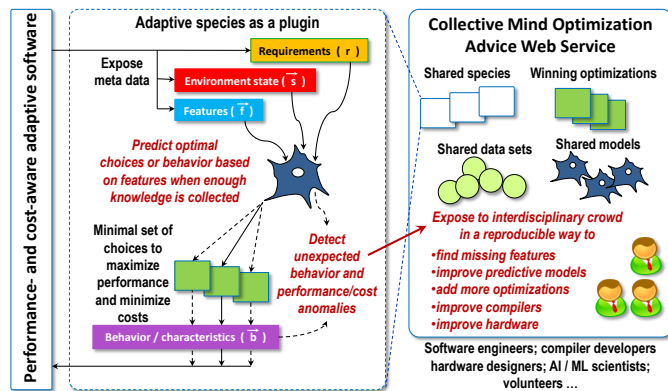

Fig. 9.   Concept of performance- and cost-aware self-tuning software assembled from cM plugins.

This real example demonstrates how our approach can help *collaboratively find missing and nontrivial features that may not be even exist and have to be exposed* to improve optimization prediction. Furthermore, our approach helped substitute the threshold filter in the customer's real software by a shared cM plugin consisting of two differently optimized clones of this filter and a compact decision tree. This decision tree selects an appropriate clone at run-time based on features of a used data set, hardware and environment state. Therefore, our Collective Mind approach can also help make statically compiled software easily adaptable to different contexts as conceptually shown in Figure 9. Moreover, such software will be continuously optimized with the help of the community while maximizing its performance, minimizing development costs, improving productivity of software engineers and reducing time to market.

We believe that presented approach can eventually en-

able performance- and cost-aware software engineering. We envision that instead of struggling integrating various ad-hoc optimization heuristics to their software projects similar to one shown in Figure 4, engineers will simply need to expose various features from data sets, software, hardware and environment state for their software pieces. These features will be then correlated with the winning optimizations either automatically or with the help of the community to gradually minimize execution time, power consumption, code size, compilation time, faults, and other costs.

VII.   CONCLUSIONS AND FUTURE WORK

The computer engineering community has been desperately trying to find some practical ways to automatically improve software performance while reducing power consumption and other usage costs across numerous and rapidly evolving computer systems for several decades [21], [30], [15], [41], [17]. In this paper, we presented a novel and practical approach inspired by natural sciences and Wikipedia that may help collaboratively solve this problem while improving productivity of software developers. The biggest challenge in this approach is to connect together, systematize and make practical various techniques and tools from different interdisciplinary domains often overlooked by our community into a coherent, extensible and top-down optimization and classification methodology.

The backbone of our approach is a public repository of optimization knowledge at c-mind.org/repo. It allows the software engineering community to gradually share their most frequently used software pieces (computational species) together with various possible inputs and features. All shared species are then continuously and randomly optimized and executed with randomly selected inputs either as standalone pieces or within real software across numerous mobile phones, laptops and data centers provided by volunteers using our recent Collective Mind framework (cM). In contrast with a very few existing public repositories, notably SPEC and Phoronix benchmarking platforms [66], [13], cM also continuously classifies best found optimizations while exposing unexpected behavior in a reproducible way. This, in turn, allows the interdisciplinary community to collaboratively correlate found classes with gradually exposed features from the software, hardware, data sets and environment state either manually or using popular big data predictive analytics [23], [42]. Resulting predictive models are then integrated into cM plugins together with several pre-optimized (specialized) versions of a given species that maximize performance and minimize costs across as many inputs, hardware and environment states as possible.

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

our approach will also help simplify compilers and convert them into generic libraries of code analysis, optimization and generation routines orchestrated by cM-like frameworks.

To avoid the fate of many projects that vanish shortly after publication, we agreed with our partners to share most of the related code and data at our public optimization repository to continue further community-driven developments. For example, with the help of our supporters, we already shared around 300 software species and collected around 15000 possible data sets. At the same time, we also shared various features as cM meta-data from our past research on machine learning based optimization including MILEPOST semantic code properties [38], code patterns and control flow graph extracted by our GCC/LLVM Alchemist plugin [39], image and matrix dimensions together with data set sizes from [55], OS parameters, system descriptions, hardware performance counters, CPU frequency and many other.

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

As a part of the future work, we plan to simplify as much as possible the experience of software engineers and volunteers wishing to participate in our project. Therefore, we are currently extending our cM framework to automate identification, extraction and sharing of the frequently used and most time consuming software pieces and their features in real programs. For this purpose, we plan to use and extend our Interactive Compilation Interface for GCC and LLVM while connecting cM framework with Eclipse IDE [6] to simplify integration of our cM wrappers and performance/cost monitoring plugins with real applications, with Docker [5] and CARE [47] to automatically detect all software dependencies for sharing, and with Phoronix open benchmarking infrastructure [13] to add even more realistic software pieces to our repository.

At the same time, our top-down methodology originating from physics allows the software engineering community benefit from all existing optimizations including powerful polyhedral source-to-source code restructuring and parallelization [24] by gradually adding them to our cM performance tracking and tuning framework [39]. Furthermore, researchers now have an opportunity to immediately validate their novel or existing optimization techniques across a realistic benchmark and a large number of participating computer systems. Unified big data repository of optimization knowledge also helped us initiate collaboration with AI and physics departments to gradually characterize complex interactions between shared software pieces inside large applications using agent based techniques [64]. We are also actively working with the academic and industrial community through the cTuning foundation to continue adding more coarse-grain and fine-grain species to our repository to gradually cover all possible and frequently used software. We hope, that in a longer term, our approach will help software and hardware engineers boost innovation and focus on implementing new ideas and functionality rather than worrying about performance regressions and costs. Finally, we plan to use our public repository to promote and support recent initiatives on reproducible and sustainable software engineering.

References

[1] ACOVEA: Using Natural Selection to Investigate Software Complexities. http://www.coyotegulch.com/products/acovea.

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

[3] 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.

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

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

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

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

[8] Gcc developers mailing list. https://gcc.gnu.org/ml/gcc.

[9] Google knowledge graph. http://en.wikipedia.org/wiki/Knowledge_Graph.

[10] Llvm developers mailing list. http://lists.cs.uiuc.edu/pipermail/llvmdev.

[11] MiDataSets: multiple datasets for cBench/MiBench benchmark. http://cTuning.org/cbench.

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

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

[14] Openbenchmarking.org blog. http://openbenchmarking.org/blog.

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

[16] The r project for statistical computing. http://www.r-project.org.

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

[18] B. Aarts and 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.

[19] 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.

[20] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine 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, December 2006.

[21] 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.

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

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

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

[34] 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.

[35] 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.

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

[37] 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.

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

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

[40] GRID5000. A nationwide infrastructure for large scale parallel and distributed computing research. http://www.grid5000.fr.

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

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

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

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

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

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

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] Irene Manotas, Lori Pollock, and James Clause. Seeds: A software engineer's energy-optimization decision support framework. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 503–514, New York, NY, USA, 2014. ACM.

[57] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: Online contention detection and response. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 257–265, New York, NY, USA, 2010. ACM.

[58] 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.

[59] 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.

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

[61] A. Nisbet. Iterative feedback directed parallelisation using genetic algorithms. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation in conjunction with International Conference on Parallel Architectures and Compilation Technique (PACT)*, 1998.

[62] 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.

[63] 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.

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

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

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

[67] 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.

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

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