Collective Knowledge: towards R&D sustainability

Grigori Fursin
cTuning foundation (France) / dividiti (UK)
E-mail: Grigori.Fursin@cTuning.org

Anton Lokhmotov
dividiti (UK)
E-mail: anton@dividiti.com

Ed Plowman
ARM (UK)
E-mail: Ed.Plowman@arm.com

***Abstract*—Research funding bodies strongly encourage research projects to disseminate discovered knowledge and transfer developed technology to industry. Unfortunately, capturing, sharing, reproducing and building upon experimental results has become close to impossible in computer systems' R&D. The main challenges include the ever changing hardware and software technologies, lack of standard experimental methodology and lack of robust knowledge exchange mechanisms apart from publications where reproducibility is still rarely considered.**

Supported by the EU FP7 TETRACOM Coordination Action, we have developed Collective Knowledge (CK), an open-source framework and methodology that involves the R&D community to solve the above problems collaboratively. CK helps researchers gradually convert their code and data into reusable components and share them via repositories such as GitHub, design and evolve over time experimental scenarios, replay experiments under the same or similar conditions, apply state-of-the-art statistical techniques, crowdsource experiments across different platforms, and enable interactive publications. Importantly, CK encourages the continuity and sustainability of R&D efforts: researchers and engineers can build upon the work of others and make their own work available for others to build upon. We believe that R&D sustainability will lead to better research and faster commercialization, thus increasing return-on-investment.

I. Introduction

Research funding bodies, including the European Commission, base their policies to maximize in the long term return on investment of public money. Calls for proposals are replete with requests for evaluating potential impact, disseminating results and facilitating technology transfer [1], [2]. Yet, more often than not, even promising results are not shared in a way that others (including members of the same consortium) can easily review, reuse and build upon. This leads to the problem of R&D sustainability that can be formulated as follows: how to ensure that valuable (mostly human) resources are used responsibly towards creating reliable and reusable knowledge?

Several practical tools have emerged to address R&D sustainability, most notably, Virtual Machines [3] and Docker [4]. These tools have quickly gained popularity in computational science to make snapshots of the whole software environment of an experiment, capturing all code and data used during its execution. Unfortunately, such monolithic snapshots are of limited use in computer systems' R&D, where researchers need to validate and build upon previous work using the latest available hardware, software, tools and workloads. Importantly, software snapshots do not capture well run-time state information critical for this domain [5], [6].

Supported by a grant from the EU FP7 609491 TETRACOM Coordination Action,[1] we have developed Collective Knowledge (CK), an open framework and methodology to address the needs of computer systems' R&D. To ensure its wide adoption, we have released CK under a permissive license and provided extensive documentation.[2]

We describe how CK has already enabled several research projects to share their research artifacts as reproducible and reusable components via our live repository at http://cknowledge.org/repo. We humbly hope that CK will contribute to R&D sustainability, which will in turn lead to better research and faster commercialization.

II. Collective Knowledge: a framework for reproducible and collaborative R&D

Collective Knowledge is a simple, portable and extensible framework for reproducible and collaborative R&D. Using CK, researchers can create and share entire experimental workflows involving components such as programs (*e.g.* benchmarks), data sets, tools (*e.g.* compilers and libraries), experimental results, predictive models, articles, *etc.*. In addition, CK components can abstract away access to hardware, monitor run-time state, apply predictive analytics, *etc.*

As shown in Fig. 1a, each CK component has a class. Classes are implemented as Python modules, with a JSON[3] meta description, JSON-based API, and unified command line interface. New classes can be defined as needed.

Each CK component has a DOI-style unique identifier (UID). CK components can be referenced and searched through via their UIDs using Hadoop-based Elasticsearch.[4] CK components can be flexibly combined into experimental workflows, similar to playing with LEGO® bricks.

Researchers can share CK workflows complete with all their components via repositories such as GitHub. Other researchers can reproduce an experiment under the same or similar conditions using a single CK command. Importantly, if the other researchers are unable to reproduce an experiment due to uncaptured dependencies (*e.g.* on run-time state), they can "debug" the workflow and share the "fixed" workflow back (possibly with new extensions, experiments, models, *etc.*).

[1] http://tetracom.eu
[2] https://github.com/ctuning/ck, https://github.com/ctuning/ck/wiki
[3] JavaScript Object Notation: http://json.org
[4] Open-source distributed real-time search and analytics: http://elastic.co

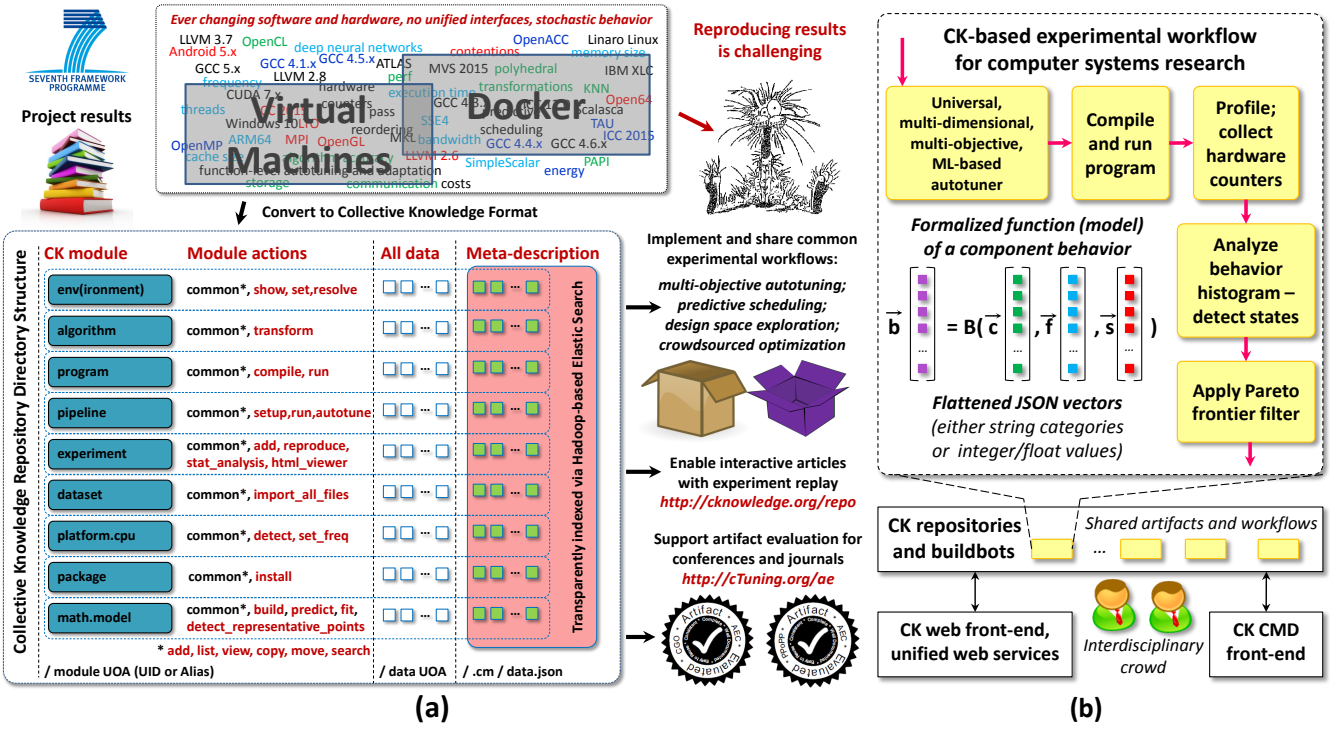

Fig. 1. (a) Tackling the reproducibility and reusability challenges in computer systems' R&D with the Collective Knowledge framework; (b) assembling experimental workflows from shared components similar to playing with LEGO® bricks.

The community is thus able to gradually expose in a unified way multi-dimensional design and optimization choices **c** of all components, their features **f**, dependencies on other components, run-time state **s** and observed behavior **b**, as shown in Fig. 1b and described in detail in [6], [7].

This, in turn, allows the community to address the most essential question of computer systems' R&D: how to optimize any given computation in terms of performance, power consumption, resource usage, accuracy, resiliency and cost; in other words, how to learn and optimize the behavior function B:

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

We believe our approach can help change researchers' mentality making them think about how to make their workflows reusable and extensible by the community. For example, the reviewers can validate presented techniques at submission time; the authors can fix problems at rebuttal time; the wider community can reuse and maintain research artifacts after publication. Eventually, our approach should also enable open computer systems' research, similar to the open source software movement [7], [8], [9].

III. Practical use cases

We demonstrate the Collective Knowledge approach using two realistic programs: HOG from the Realeyes image processing benchmark [10] developed in the EU CARP project[5] and KFusion from the SLAMBench 3D scene understanding benchmark [11] developed in the UK PAMELA project.[6] Our approach formalizes and supports many techniques commonly used in computer systems' R&D such as systematic experimentation (§III-A), run-time adaptation (§III-B), and design space exploration (§III-C). See [12] for a live report with further information, shared artifacts and interactive graphs.

A. Systematic experimentation

Performance evaluation (colloquially called "benchmarking") is one of the most important activities in computer engineering. Getting it right, however, is hard. For example, on mobile devices, unexpected performance variation can often be attributed to dynamic voltage and frequency scaling (DVFS). Mobile devices have power and temperature limits to prevent device damage; in addition, when a workload's computational requirements can still be met at a lower frequency, lowering the frequency conserves energy. Further complications arise when benchmarking on heterogeneous multicore systems such as ARM big.LITTLE: in a short time, a workload can migrate between cores having different microarchitectures, as well as running at different frequencies. Controlling for such factors (or at least accounting for them with elementary statistics) is key to meaningful performance evaluation on mobile devices.

Fig. 2 shows a performance surface plot for the HOG program (4×4 cells; 64×64 blocks; one image) on a Samsung Chromebook 2 with DVFS disabled and the processors' frequencies controlled for. The X and Y axis show the CPU

[5]http://carpproject.eu

[6]http://apt.cs.manchester.ac.uk/projects/PAMELA

and the GPU frequencies; the Z axis shows the GPU *kernel* execution time. As expected, the execution time is roughly inversely proportional to the GPU frequency, but does not depend on the CPU frequency.

Fig. 3 is similar but the Z axis shows the *total* GPU execution time calculated as the *sum* of the kernel execution time and the overhead of transferring the data to the GPU and back. At higher CPU frequencies (roughly greater than 800 MHz), this plot looks just like a raised version of the plot in Fig. 2. At lower CPU frequencies, however, the data copy overhead becomes more pronounced than at higher CPU frequencies. In other words, raising the CPU frequency up to 800 MHz improves the performance, but raising it higher than 800 MHz results in no further performance improvement.

An often voiced objection that on a real platform DVFS would be enabled does not invalidate the need for reproducible systematic experimentation. When software developers are aware of their program's performance profile, they can make right optimization decisions such as optimizing for typical conditions (*e.g.* when the frequencies are set at half of the peak) or the worst case.

B. Run-time adaptation with active learning

Systematically collecting performance data that can be trusted is essential but does not by itself produce insights. Collective Knowledge enables applying state-of-the-art statistical techniques to "raw data" to deliver "useful insights". For example, by amassing performance data on executing the HOG program with different images, we can confirm or reject with the desired confidence the hypothesis that its performance does not depend significantly on the image size. Perhaps we can even conclude that the performance does not depend on the image shape (dimensions). But what *does* the performance depend on (apart from the processors' frequencies)? Moreover, can we quantify the dependence? Sometimes, we will not have enough data to confidently answer certain questions. Sometimes, we will have just the opposite problem: how to find answers in the "big data" enthusiastically collected when experimentation is cheap.

Consider optimizing the execution time of a parallel workload running on a heterogeneous platform comprised of a CPU and a GPU. Running the workload on the GPU as a data parallel kernel is typically faster than on the CPU. The *total* GPU execution time, however, sometimes exceeds the CPU execution time. In fact, we have just described the behaviour of the HOG program. Fig. 4 shows on the Z axis the CPU execution time divided by the total GPU execution time. When this ratio is greater than 1 (the light pink to bright red areas), using the GPU is faster than using the CPU, despite the data copy overhead. A sensible scheduling decision, therefore, is to schedule the workload on the GPU.

While this plot suggests when the GPU should be used for this particular experiment, will it still be possible to make sensible scheduling decisions if the performance also depends on factors other than the processors' frequencies?

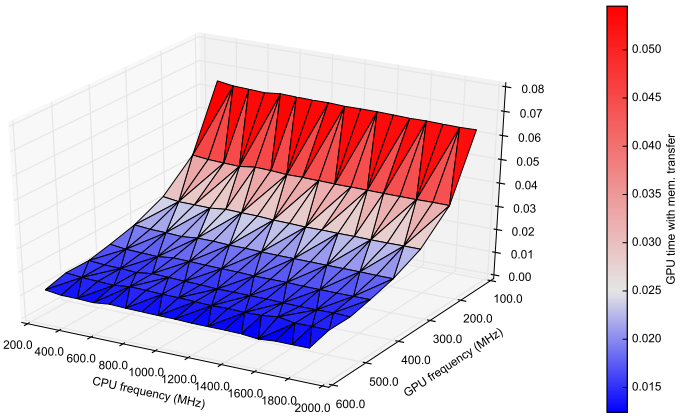

Fig. 2. Z axis: GPU [kernel only] execution time (seconds).

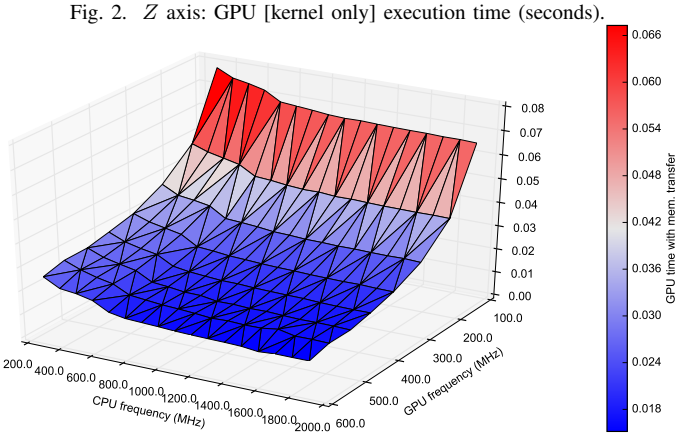

Fig. 3. Z axis: GPU [kernel + data copy] execution time (seconds).

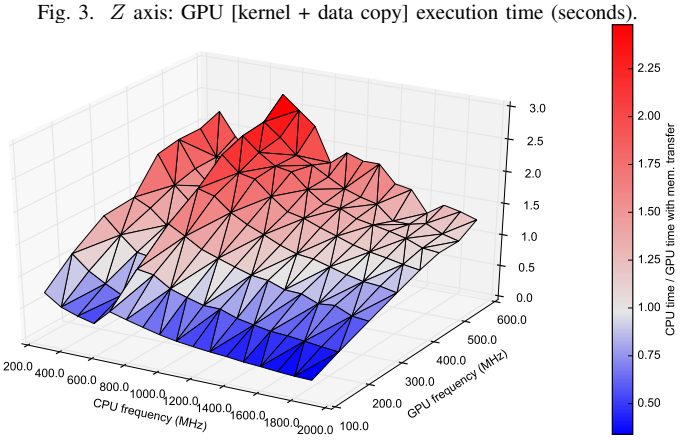

Fig. 4. Z axis: CPU execution time divided by GPU [kernel + data copy] execution time ($\times$).

To answer this question, we conducted multiple experiments with HOG (1×1 cells) on a Samsung Chromebook 1 (dual-core ARM Cortex-A15 CPU, quad-core ARM Mali-T604 GPU).[7] The experiments covered the Cartesian product of: 2 CPU frequencies (800 MHz, 1600 MHz); 2 GPU frequencies (266 MHz, 533 MHz); 3 block sizes (16, 64, 128); 23 images (in different shapes and sizes); in total, 276 experiments (with 5 repetitions each).

[7]See [12] for experiments on a Samsung Chromebook 2.

To analyze the collected experimental data, we use decision trees, a popular supervised learning method for classification and regression.[8] We build decision trees using a Collective Knowledge interface to the Python `scikit-learn` package.[9] We thus obtain a predictive model that tells us if it is faster to execute HOG on the GPU than on the CPU by looking at several features of a sample (experiment). In other words, the model assigns to a sample one of the two labels: "YES" means the GPU should be used; "NO" means the CPU should be used. We train the model on the experimental data, by labeling a sample with "YES" if the CPU execution time exceeds the GPU execution time by at least 7% (to account for variability), and with "NO" otherwise.

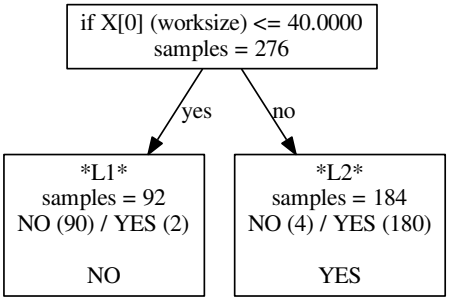

Fig. 5. Question: Is the GPU [kernel + data copy] faster than the CPU? Model: feature set: 1; depth: 1.

Fig. 5 shows a decision tree of depth 1 built using a single feature: the algorithm block size (designated as 'worksize'), which, informally, determines the computational intensity of HOG. The root node divides the training set of 276 samples into two subsets. For 92 samples in the first subset, represented by the left leaf node ("L1"), the worksize is less than or equal to 40 (i.e. 16). For 184 samples in the second subset, represented by the right leaf node ("L2"), the worksize is greater than 40 (i.e. 64 and 128).

In the first subset, 90 samples are labeled with "NO" and 2 samples are labeled with "YES". Since the majority of the samples are labeled with "NO", the tree predicts that the workload for which the worksize is less than or equal to 40 should be executed on the CPU. Similarly, the workload for which the worksize is greater than 40 should be executed on the GPU. Intuitively, this makes sense: the workload with a higher computational intensity (a higher worksize value) should be executed on the GPU, despite the data copy overhead.

For 6 samples out of 276, the model in Fig. 5 mispredicts the correct scheduling decision. (We say that the rate of correct predictions is $270/276$ or 97.8%.) For example, for the 2 samples out of 92 in the subset for which the worksize is 16 ("L1"), the GPU was still faster than the CPU. Yet, based on labeling of the majority of the samples in this subset, the model mispredicts that the workload should be executed on the CPU.

[8]https://en.wikipedia.org/wiki/Decision_tree_learning
[9]http://scikit-learn.org

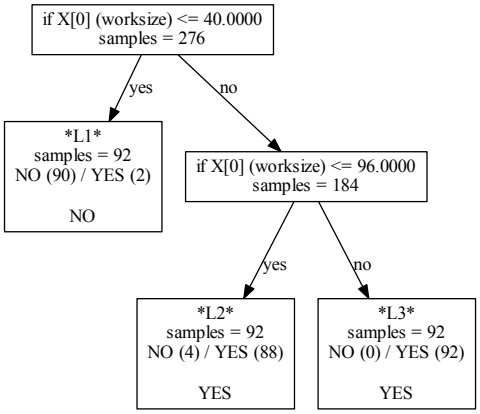

Fig. 6. Question: Is the GPU [kernel + data copy] faster than the CPU? Model: feature set: 1; depth: 2.

Fig. 6 shows a decision tree of depth 2 using the same worksize feature. The right child of the root now has two children of its own. All the samples in the rightmost leaf ("L3") for which the worksize is greater than 96 (i.e. 128) are labeled with "YES". This means that at the highest computational intensity, the GPU was always faster than the CPU, thus confirming our intuition. However, the model in Fig. 6 still makes 6 mispredictions. To improve the prediction rate, we build models using more features, as well as having more levels. We consider 2 more sets of features as in Fig. 7.

Id	**Features**
FS1	worksize [block size]
FS2	all features from FS1, CPU frequency, GPU frequency, image rows (m), image columns (n), image size ($m \times n$), (GWS0, GWS1, GWS2) [OpenCL global work size]
FS3	all features from FS2, image size / CPU frequency, image size / GPU frequency, CPU frequency / GPU frequency

Fig. 7. Feature sets: simple (FS1); natural (FS2); designed (FS3).

The "natural" set is constructed from the features that we expected would impact the scheduling. Fig. 8 shows a decision tree of depth 4 built using this set. This model uses 4 additional features (the GPU frequency, the CPU frequency, the number of image columns, the number of image rows) and has 8 leaf nodes. This model makes the same decision on the worksize at the top level, but better fits the training data at lower levels. Still, it results in 2 mispredictions ("L7"), achieving the prediction rate of 99.3%. However, this model is more difficult to grasp intuitively and may not fit new data well.

The "designed" set can be used to build models achieving the 100.0% prediction rate. A decision tree of depth 5 (not shown) uses all the new features from the designed set. With 12 leaf nodes, however, this model is even less intuitive and exhibits even more overfitting than the one in Fig. 8.

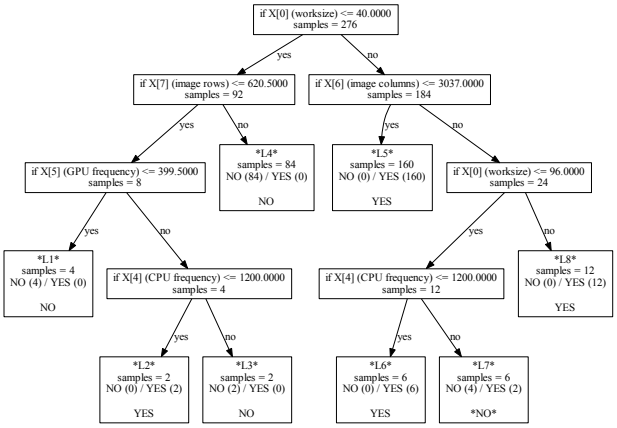

Fig. 8. Question: Is the GPU [kernel + data copy] faster than the CPU? Model: feature set: 2; depth: 4.

C. Design space exploration

The architecture of CK with unified JSON interfaces makes it easy not only to replay shared workflows, but also to augment the workflows during replay. This enables performing automatic design space exploration (autotuning) of exposed optimization choices.

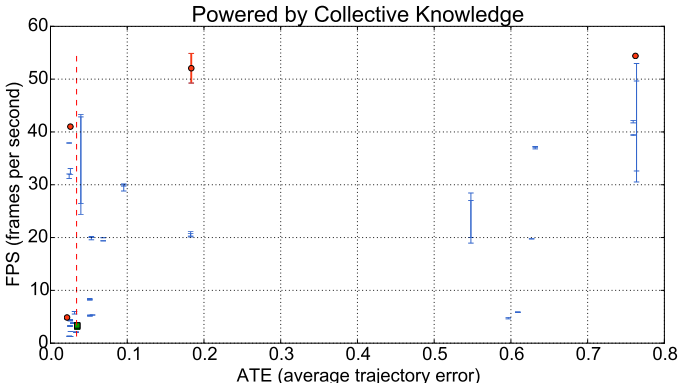

Fig. 9. SLAMbench design space exploration on an Odroid XU3 platform (see [12] for an interactive graph).

Fig. 9 shows results of such design space exploration via random sampling for the KFusion program. The Y axis shows the performance in frames per second (with error bars); the X axis shows the average trajectory error (tracking accuracy) in meters. The green dot represents the default configuration. The red dots represent a Pareto frontier. Configurations to the left of the dashed line have better accuracy than the default. One configuration is both roughly 10 times faster and more accurate than the default configuration.

We have similarly used CK to tune parameterized OpenCL programs across multiple devices and also found order-of-magnitude improvements over reasonable defaults. We thus believe using CK can improve performance portability of OpenCL programs and enable fair comparisons between devices.

IV. Conclusion and Outlook

We have presented Collective Knowledge, the first to our knowledge open framework that involves the community to gradually grow a common methodology for reproducible computer systems' R&D. Our approach allows to crowdsource experimentation across multiple platforms, programs, datasets, *etc.* Moreover, it opens up opportunities to grow community knowledge in forms of predictive models where mispredictions can be detected and addressed by the experts as more data becomes available. We outline our vision for community-based R&D in several areas.

A. Community-sourced benchmarks

Two issues often affect validity and conclusions of performance analysis [13]:

1) Evaluation is not conducted in a rigorous way.
2) Workloads selected for evaluation are not representative.

Our approach aims to address the first issue by using feedback and contributions from the expert community to avoid common pitfalls.

The second issue is more insidious. Several companies devise and license benchmark suites based on their guesses of what representative workloads might be in the near future. Since benchmarking is their primary business, their programs, datasets and methodology often go unchallenged, with the benchmarking scores driving the purchasing decisions both of OEMs (*e.g.* phone manufacturers) and consumers (*e.g.* phone users). When stakes are that high, the vendors have no choice but to optimize their products for the commercial benchmarks. When those turn out to have no close resemblance to real workloads, the products underperform.

We believe we can collectively tackle the second issue as well as the first one. The community will both provide representative workloads and rank them according to established quality criteria. From time to time (say, every 6 months), a selection of workloads (say, the top 20) can be further ranked by a panel of recognized experts to provide a valuable complement to today's commercial benchmark suites.

The success will depend on establishing the right incentives for the community. Leading academics have long recognized the need for representative workloads to drive research in hardware design and software tools. For example, the UK PAMELA project promotes 3D scene understanding as a key use case.[10] With funding agencies increasingly requiring academics to demonstrate impact of their research and to share their code and data, the time is ripe for excellent individual research to make a wide community impact.

Incentives to share representative workloads may be somewhat different for industry. As the example of Realeyes shows, even when commercial sensitivity prevents a company from releasing their full application under an open-source license, it may still be possible to distill a performance-sensitive portion of it into a standalone benchmark. The community can help the company to optimize their benchmark (for free or for fee), thus

[10]Google agrees: https://www.google.com/atap/project-tango

improving the overall performance of their full application.[11] Some researchers and software developers will just want to see their benchmark appear in the ranked selection of workloads, highlighting their skill and expertise (similar to "kudos" for open-source contributions).

B. Community-sourced performance data

We see the potential of creating open performance data even from completely closed-source software. For example, a software company wishing to improve performance of their product may turn to services of a specialized autotuning provider. A *successful* provider would perform astounding numbers of performance experiments every day. Rather than throwing the experimental data away, a *responsible* autotuning provider would include retention of anonymized data in the terms and conditions of their service. A similar case can be made for providers of device farms such as Amazon Device Farm and Xamarin Test Cloud.[12] An even more intriguing case can be made for hardware vendors to share their performance data from internal benchmarking. The objection we hear most often from hardware vendors is that performance data is highly sensitive. Give too much information, the sceptics argue, and you will expose your weaknesses to competitors, customers, press and even patent trolls. (Give too little, we argue, and software developers will be unable to use your hardware effectively.)

The CK approach can support a variety of mechanisms and models for knowledge sharing. Each party will decide for themselves how much, if anything, they share. Some vendors will only share data about publicly available devices. (This kind of data the community could actually gather without vendors.) Some vendors will share data about their products in development but only under an NDA to selected parties. Some vendors will not share raw data but will be happy for software developers to have access to performance models built from such data. Some test farms providers will license raw data or data-mined insights to software developers, tool developers or even back to hardware vendors. We envision that eventually the pros and cons of "open performance" will be understood and accepted as they are understood and accepted today for "open software".

C. Community-grown programming tools

Developing programming tools is challenging but not particularly profitable. For example, independent compiler companies often struggle to make the ends meet, change owners or completely disappear. Lack of funding also means that even when academic research is close to industry needs, it rarely gets successfully transferred.

The community can encourage better research and spur technology transfer of programming tools. First, having access to community-sourced realistic workloads will be a boon to programming tool developers. Second, rigorous evaluation using the CK approach can help the community to separate leaders from also-rans. Third, the community can provide valuable feedback on programming tools, making sure that promising tools receive the attention and funding they deserve. All of this will lead to more innovation in tools and consequently greatly benefit the software development ecosystem.

D. Exciting opportunities on the horizon!

We view Collective Knowledge as a catalyst for stimulating flows of reproducible insights across largely divided hardware/software and industry/academia communities. Better flows will lead to breakthroughs in energy efficiency, performance and reliability of computer systems. Effective knowledge sharing and open innovation will enable new exciting applications in consumer electronics, robotics, automotive and healthcare—at better quality, lower cost and faster time-to-market.

[11] The original HOG paper [14] has over 12500 citations. Just imagine all this community combining their efforts to squeeze out every gram of HOG performance across different configurations, data sets, platforms, etc.

[12] https://aws.amazon.com/device-farm, https://xamarin.com/test-cloud

ACKNOWLEDGEMENTS

We thank the EU FP7 TETRACOM Coordination Action and the CK community for their feedback, discussions and contributions.

REFERENCES

[1] "Scientific data: open access to research results will boost Europe's innovation capacity." http://europa.eu/rapid/press-release_IP-12-790_en.htm.

[2] "EU Open Science and Open Access Policies." http://ec.europa.eu/research/swafs/index.cfm?pg=policy&lib=science.

[3] J. E. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, pp. 32–38, May 2005.

[4] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[5] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pp. 257–265, 2010.

[6] G. Fursin, R. Miceli, A. Lokhmotov, M. Gerndt, M. Baboulin, D. Malony, Allen, Z. Chamski, D. Novillo, and D. D. Vento, "Collective Mind: Towards practical and collaborative auto-tuning," *Scientific Programming*, vol. 22, pp. 309–329, July 2014.

[7] G. Fursin, A. Memon, C. Guillon, and A. Lokhmotov, "Collective Mind, Part II: Towards performance- and cost-aware software engineering as a natural science," in *18th International Workshop on Compilers for Parallel Computing (CPC'15)*, January 2015.

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

[9] "Community-driven Artifact Evaluation Initiative for PPoPP and CGO conferences." http://cTuning.org/ae.

[10] E. Hajiyev, R. Dávid, L. Marák, and R. Baghdadi, "Realeyes image processing benchmark." https://github.com/Realeyes/pencil-benchmarks-imageproc, 2011–2015.

[11] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham, and S. Furber, "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM," in *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167.

[12] G. Fursin and A. Lokhmotov, "Live report with shared artifacts and interactive graphs." http://cknowledge.org/repo/web.php?wcid=report:b0779e2a64c22907.

[13] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, May 1991.

[14] N. Dalal and T. Bill, "Histograms of oriented gradients for human detection," 2005.