A Collective Knowledge workflow for collaborative research into multi-objective autotuning and machine learning techniques

Grigori Fursin[1,2], Dmitry Savenko[3], Anton Lokhmotov[1], and Eben Upton[4,5]

[1] dividiti, UK ; [2] cTuning foundation, France ; [3] Xored, Russia
[4] Raspberry Pi foundation, UK ; [5] Broadcom, UK

August 15, 2017 (first draft)

Abstract

Developing efficient software and hardware has never been harder whether it is for a tiny IoT device or an Exascale supercomputer. Apart from the ever growing design and optimization complexity, there exist even more fundamental problems such as lack of interdisciplinary knowledge required for effective software/hardware co-design, and a growing technology transfer gap between academia and industry.

We introduce our new educational initiative to tackle these problems by developing Collective Knowledge (CK), a unified experimental framework for computer systems research and development. We use CK to teach the community how to make their research artifacts and experimental workflows portable, reproducible, customizable and reusable while enabling sustainable R&D and facilitating technology transfer. We also demonstrate how to redesign multi-objective autotuning and machine learning as a portable and extensible CK workflow. Such workflows enable researchers to experiment with different applications, data sets and tools; crowdsource experimentation across diverse platforms; share experimental results, models, visualizations; gradually expose more design and optimization choices using a simple JSON API; and ultimately build upon each other's findings.

As the first practical step, we have implemented customizable compiler autotuning, crowdsourced optimization of diverse workloads across Raspberry Pi 3 devices, reduced the execution time and code size by up to 40%, and applied machine learning to predict optimizations. We hope such approach will help teach students how to build upon each others' work to enable efficient and self-optimizing software/hardware/model stack for emerging workloads.

Keywords: *collective knowledge, customizable research workflows, portable package manager, reusable artifacts, collaborative optimization, customizable autotuning, machine learning, crowd-fuzzing, software/hardware co-design competitions, technology transfer, Raspberry Pi*

Live CK repository:
github.com/ctuning/ck-rpi-optimization-results (0.9GB)
Interactive report:
cKnowledge.org/rpi-crowd-tuning
Archives of CK repositories at FigShare:
doi.org/10.6084/m9.figshare.5789007.v2

1 Introduction

Many recent international roadmaps for computer systems research appeal to reinvent computing [29, 52, 9]. Indeed, developing, benchmarking, optimizing and co-designing hardware and software has never been harder, no matter if it is for embedded and IoT devices, or data centers and Exascale supercomputers. This is caused by both physical limitations of existing technologies and an unmanageable complexity of continuously changing computer systems which already have too many design and optimization choices and objectives to consider at all software and hardware levels [65], as conceptually shown in Figure 1. That is why most of these roadmaps now agree with our vision that such problems should be solved in a close collaboration between industry, academia and end-users [58, 64].

However, after we initiated artifact evaluation (AE) [20, 48] at several premier ACM and IEEE conferences to reproduce and validate experimental results from published papers, we noticed an even more fundamental problem: a growing technology transfer gap between academic research and industrial development. After evaluating more than 100 artifacts from the leading computer systems conferences in the past 4 years, we noticed that only a small fraction of research artifacts could be easily customized, ported to other environments and hardware, reused, and built upon. We have grown to believe that this due to a lack of a common workflow framework that could simplify implementation and sharing of artifacts and workflows as portable, customizable and reusable components with

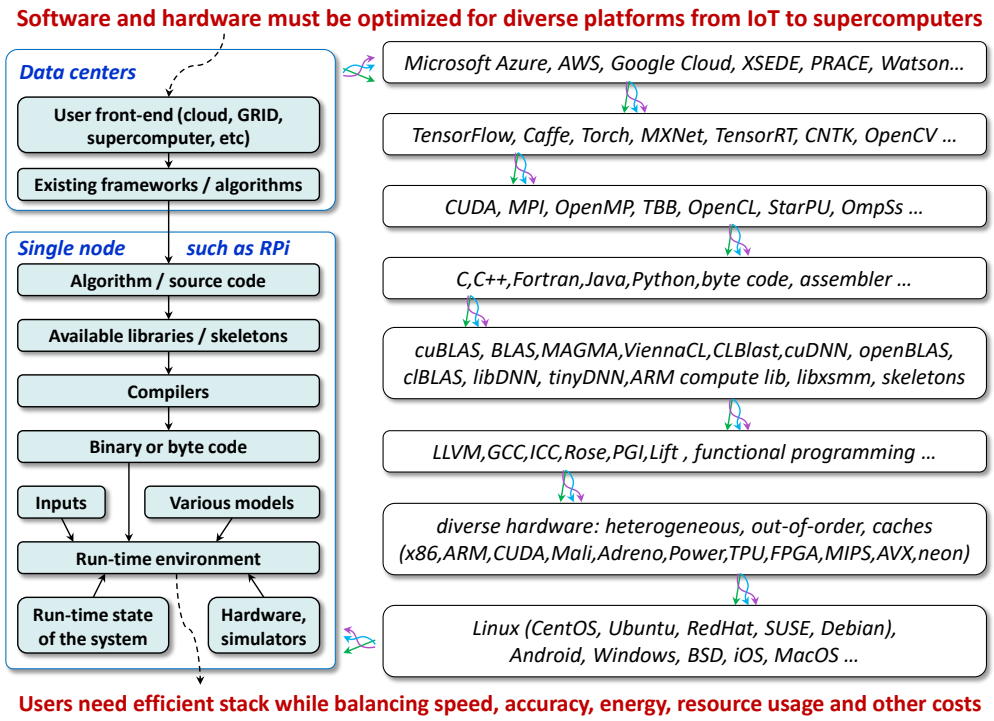


Figure 1: Too many design and optimization choices at all levels of the continuously changing software and hardware stack make it extremely challenging and time consuming to design efficient computer systems for realistic workloads.

some common API and meta information vital for open science [60].

At the same time, companies are always under pressure and rarely have time to dig into numerous academic artifacts shared as CSV/Excel files and "black box" VM and Docker images, or adapt numerous ad-hoc scripts to realistic and ever changing workloads, software and hardware. That is why promising techniques may remain in academia for decades while just being incrementally improved, put on the shelf when leading students graduate, and "reinvented" from time to time.

Autotuning is one such example: this very popular technique has been actively researched since the 1990s to automatically explore large optimization spaces and improve efficiency of computer systems [113, 89, 49, 112, 66, 50, 78, 105, 84, 80, 96, 104, 44, 72, 61, 110, 30, 100, 69, 70, 76, 39, 111, 34]. Every year, dozens of autotuning papers get published to optimize some computer system components, improve and speed up exploration and co-design strategies, and enable run-time adaptation. Yet, when trying to make autotuning practical (in particular, by applying machine learning) we faced numerous challenges with integrating such published techniques into real, complex and continuously evolving software and hardware stack [58, 65, 64, 60].

Eventually, these problems motivated us to develop a common experimental framework and methodology similar to physics and other natural sciences to collaboratively improve autotuning and other techniques. As part of this educational initiative, we implemented an extensible, portable and technology-agnostic workflow for autotuning using the open-source Collective Knowledge framework (CK) [25, 62]. Such workflows help researchers to reuse already shared applications, kernels, data sets and tools, or add their own ones using a common JSON API and meta-description [6]. Moreover, such workflows can automatically adapt compilation and execution to a given environment on a given device using integrated cross-platform package manager.

Our approach takes advantage of a powerful and holistic top-down methodology successfully used in physics and other sciences when learning complex systems. The key idea is to let novice researchers first master simple compiler flag autotuning scenarios while learning interdisciplinary techniques including machine learning and statistical analysis. Researchers can then gradually increase complexity to enable automatic and collaborative co-design of the whole SW/HW stack by exposing more design and optimization choices, multiple optimization objectives (execution time, code size, power consumption, memory usage, platform cost, accuracy, etc.), crowdsource autotuning across diverse devices provided by volunteers similar to SETI@home [37], continuously exchange and discuss optimization results, and eventually build upon each other's results.

We use our approach to optimize diverse kernels and real workloads such as `zlib` in terms of speed and code size by crowdsourcing compiler flag autotuning across Raspberry Pi3 devices using the default GCC 4.9.2 and

the latest GCC 7.1.0 compilers. We have been able to achieve up to 50% reductions in code size and from 15% to 8 times speed ups across different workloads over the "-O3" baseline. Our CK workflow and all related artifacts are available at GitHub to allow researchers to compare and improve various exploration strategies (particularly based on machine learning algorithms such as KNN, GA, SVM, deep learning, though further documentation of APIs is still required) [61, 64]. We have also shared all experimental results in our open repository of optimization knowledge [8, 10] to be validated and reproduced by the community.

We hope that our approach will serve as a practical foundation for open, reproducible and sustainable computer systems research by connecting students, scientists, end-users, hardware designers and software developers to learn together how to co-design next generation of efficient and self-optimizing computer systems, particularly via reproducible competitions such as ReQuEST [31].

This technical report is organized as follows. Section 2 introduces the Collective Knowledge framework (CK) and the concept of sharing artifacts as portable, customizable and reusable components. Section 3 describes how to implement a customizable, multi-dimensional and multi-objective autotuning as a CK workflow. Section 4 shows how to optimize compiler flags using our universal CK autotuner. Section 5 presents a snapshot of the latest optimization results from collaborative tuning of GCC flags for numerous shared workloads across Raspberry Pi3 devices. Section 6 shows optimization results of zlib and other realistic workloads for GCC 4.9.2 and GCC 7.1.0 across Raspberry Pi3 devices. Section 7 describes how implement and crowdsource fuzzing of compilers and systems for various bugs using our customizable CK autotuning workflow. Section 8 shows how to predict optimizations via CK for previously unseen programs using machine learning. Section 9 demonstrates how to select and autotune models and features to improve optimization predictions while reducing complexity. Section 10 shows how to enable efficient, input-aware and adaptive libraries and programs via CK. Section 11 presents CK as an open platform to support reproducible and Pareto-efficient co-design competitions of the whole software/hardware/model stack for emerging workloads such as deep learning and quantum computing. We present future work in Section 12. We also included Artifact Appendix to allow students try our framework, participate in collaborative autotuning, gradually document APIs and improve experimental workflows.

2 Converting ad-hoc artifacts to portable and reusable components with JSON API

Artifact sharing and reproducible experimentation are key for our collaborative approach to machine-learning based optimization and co-design of computer systems, which was first prototyped during the EU-funded MILEPOST project [58, 7, 61]. Indeed, it is difficult, if not impossible, and time consuming to build useful predictive models without large and diverse training sets (programs, data sets), and without crowdsourcing design and optimization space exploration across diverse hardware [60, 64].

While we have been actively promoting artifact sharing for the past 10 years since the MILEPOST project [58, 4], it is still relatively rare in the community systems community. We have begun to understand possible reasons for that through our Artifact Evaluation initiative [20, 48] at PPoPP, CGO, PACT, SuperComputing and other leading ACM and IEEE conferences which has attracted over a hundred of artifacts in the past few years.

Unfortunately, nearly all the artifacts have been shared simply as zip archives, GitHub/GitLab/Bitbucket repositories, or VM/Docker images, with many ad-hoc scripts to prepare, run and visualize experiments, as shown in Figure 2a. While a good step towards reproducibility, such ad-hoc artifacts are hard to reuse and customize as they do not provide a common API and meta information.

Some popular and useful services such as Zenodo [17] and FigShare [16] allow researchers to upload individual artifacts to specific websites while assigning DOI [13] and providing some meta information. This helps the community to discover the artifacts, but does not necessarily make them easy to reuse.

After an ACM workshop on reproducible research methodologies (TRUST'14) [18] and a Dagstuhl Perspective workshop on Artifact Evaluation [48], we concluded that compute systems research lacked a common experimental framework in contrast with other sciences [95].

Together with our fellow researchers, we also assembled the following wish-list for such a framework:

- it should be able to help researchers quickly organize their local code and data into discoverable and reusable components with a unique ID, common API and unified meta information, rather then being forced to upload them to the web from the start;
- it should be open-source with a permissive license to simplify technology transfer;
- it should be portable, simple to install and use from the command line;

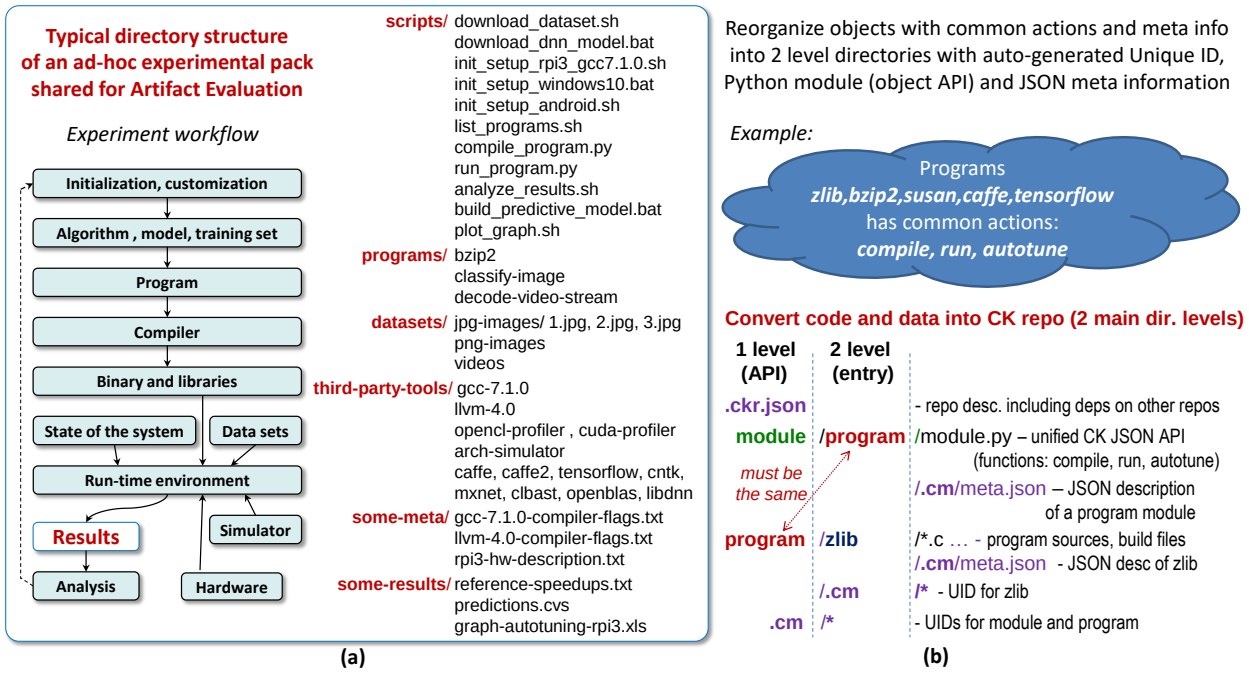


Figure 2: Reorganizing ad-hoc experimental packs into reusable, customizable and discoverable components with JSON API and meta information using the Collective Knowledge framework.

- it should allow to assemble experimental workflows by simply plugging in shared components;
- it should support native non-virtualized execution of such workflows, i.e. not only via Virtual Machine [106] and Docker [5], critical for empirical program optimization and hardware co-design experiments;
- it should be able to adapt to continuously evolving software environments and support different versions of tools such as rapidly evolving compilers and libraries;
- it should include a local web server to simplify crowdsourcing of experiments and visualization of results in workgroups.

Since there was no available open-source framework with all these features, we decided to develop such a framework, Collective Knowledge (CK) [25, 62], from scratch with initial support from the EU-funded TETRACOM project [19]. CK is implemented as a small and portable Python module with a command line front-end to assist users in converting their local objects (code and data) into searchable, reusable and shareable directory entries with user-friendly aliases and auto-generated Unique ID, JSON API and JSON meta information [6], as described in [62, 2] and conceptually shown in Figure 2b.

The user first creates a new local CK repository as follows:

```
$ ck add repo:new-ck-repo
```

Initially, it is just an empty directory:

```
$ ck find repo:new-ck-repo
$ ls `ck find repo:new-ck-repo`
```

Now, the user starts adding research artifacts as CK components with extensible APIs. For example, after noticing that we always perform 3 common actions on all our benchmarks during our experiments, "compile", "run" and "autotune", we want to provide a common API for these actions and benchmarks, rather than writing ad-hoc scripts. The user can provide such an API with actions by adding a new CK module to a CK repository as follows:

```
$ ck add new-ck-repo:module:program
```

CK will then create two levels of directories *module* and *program* in the *new-ck-repo* and will add a dummy *module.py* where common object actions can be implemented later. CK will also create a sub-directory *.cm* (collective meta) with an automatically generated Unique ID of this module and various pre-defined descriptions in JSON format (date and time of module creation, author, license, etc) to document provenance of the CK artifacts.

Users can now create holders (directories) for such objects sharing common CK module and an API as follows:

```
$ ck add new-ck-repo:program:new-benchmark
```

CK will again create two levels of directories: the first one specifying used CK module (*program*) and the second one with alias *new-benchmark* to keep objects. CK will also create three files in an internal *.cm* directory:

- **meta.json** - an empty JSON file which can be gradually extended to describe a given object (such as added program in our example);
- **info.json** - a JSON file with the date and time of the last modification as well as license, copyright and author information to keep attribution of all updates for open research;
- **desc.json** - an empty JSON file to describe types of keys in **meta.json** (useful for automatic type checking) and their value ranges (useful for autotuning as we will show later in this report).

Users can then find a path to a newly created object holder (CK entry) using the *ck find program:new-benchmark* command and then copy all files and sub-directories related to the given object using standard OS shell commands.

This allows to get rid of ad-hoc scripts by implementing actions inside reusable CK Python modules as shown in Figure 3. For example, the user can add an action to a given module such as *compile program* as follows:

```
$ ck add_action module:program --func=compile
```

CK will create a dummy function body with an input dictionary *i* inside *module.py* in the CK *module:program* entry. Whenever this function is invoked via CK using the following format:

```
$ ck compile program:some_entry --param1=val1
```

the command line will be converted to *i* dictionary and printed to the console to help novice users understand the CK API. The user can now substitute this dummy function with a specific action on a specific entry (some program in our example based on its meta information) as conceptually shown in Figure 3. The above example shows how to call CK functions from Python modules rather than from the command line using the *ck.access* function. It also demonstrates how to find a path to a given *program* entry, load its meta information and unique ID. For the reader's convenience, Figure 4 lists several important CK commands.

This functionality should be enough to start implementing unified compilation and execution of shared programs. For example, the *program* module can read instructions about how to compile and run a given program from the JSON meta data of related entries, prepare and execute portable sub-scripts, collect various statistics, and embed them to the output dictionary in a unified way. This can be also gradually extended to include extra tools into compilation and execution workflow such as code instrumentation and profiling.

Here we immediately face another problem common for computer systems research: how to support multiple versions of various and continuously evolving tools and libraries? However, since we no longer hardwire calls to specific tools directly in scripts but invoke them from higher-level CK modules, we can detect all required tools and set up their environment before execution. To support this concept even better, we have developed a cross-platform package manager as a ck-env repository [22] with several CK modules including *soft*, *env*, *package*, *os* and *platform*. These modules allow the community to describe various operating systems (Linux, Windows, MacOS, Android); detect platform features (*ck detect platform*); detect multiple-versions of already installed software (*ck detect soft:compiler.gcc*); prepare CK entries with their environments for a given OS and platform using *env* module (*ck show env*) thus allowing easy co-existence of multiple versions of a given tool; install missing software using *package* modules; describe software dependencies using simple tags in a program meta description (such as *compiler,gcc* or *lib,caffe*), and ask the user to select an appropriate version during program compilation when multiple software versions are registered in the CK as shown in Figure 5.

Such approach extends the concept of package managers including Spack [68] and EasyBuild [74] by integrating them directly with experimental CK workflows while using unified CK API, supporting any OS and platform, and allowing the community to gradually extend existing detection or installation procedures via CK Python scripts and CK meta data.

Note that this CK approach encourages reuse of all such existing CK modules from shared CK repositories rather then writing numerous ad-hoc scripts. It should indeed be possible to substitute most of ad-hoc scripts from public research projects (Figure 2) with just a few above modules and entries (Figure 6), and then collaboratively extend them, thus dramatically improving research productivity. For this reason, we keep track of all publicly shared modules and their repositories in this wiki page. The user will just need to add/update a *.ckr.json* file in the root directory of a given CK repository to describe a dependency on other existing CK repositories with required modules or entries. Since it is possible to uniquely reference any CK entry by two Unique IDs (*module UID:object UID*), we also plan to develop a simple web service to automatically index and discover all modules similar to DOI.

The open, file-based format of CK repositories allows researchers to continue editing entries and their meta directly using their favourite editors It also simplifies exchange of these entries using Git repositories, zip archives, Docker images and any other popular tool. At the same time, schema-free and human

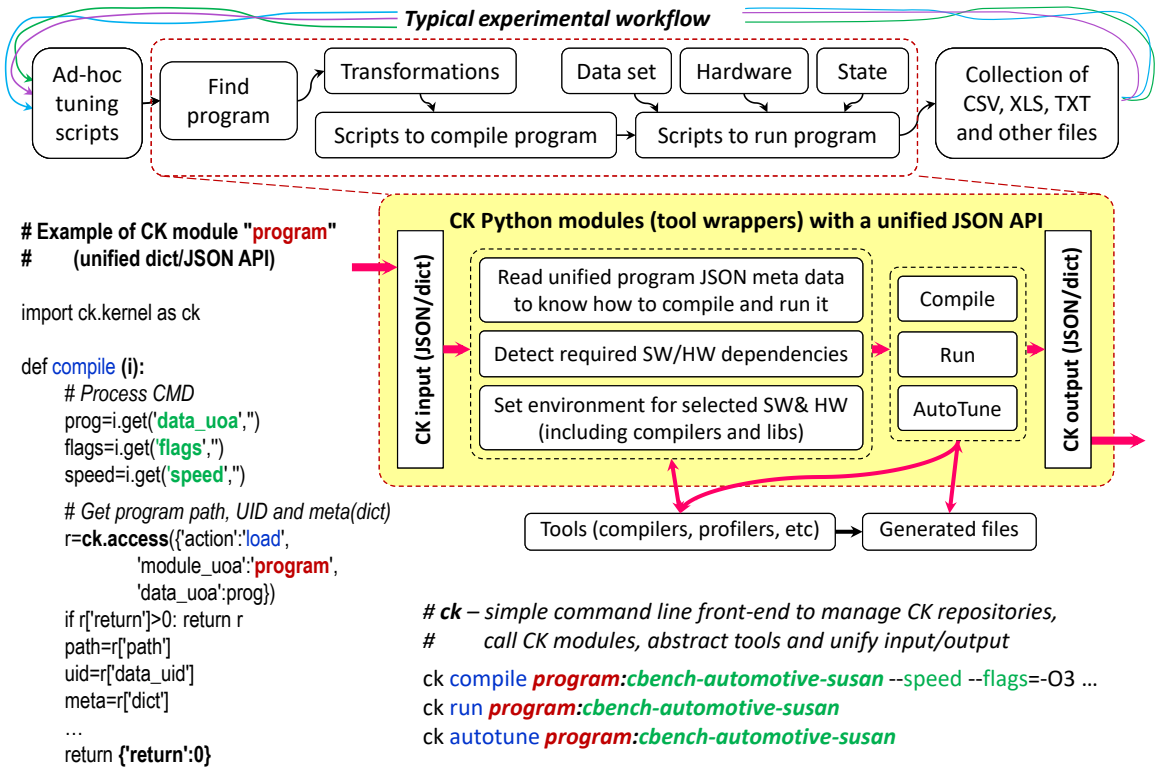


Figure 3: Converting ad-hoc scripts, tools and workflows to CK Python modules and standardized directories with actions, unified JSON API, and JSON meta information.

Create new CK repository:	**ck add repo:my_new_project**
Find CK repository:	**ck find repo:my_new_project**
List all CK repositories:	**ck list repo**
Add new module:	**ck add my_new_project:module:my_module**
Add dummy function to module:	**ck add_action my_module -- func=my_func**
Test dummy function:	**ck my_func my_module --param1=var1 --param2 -param3**
Add new entry for this module:	**ck add my_new_project:my_module:my_data @@dict** Enter *{"tags":"cool","data"}*
Add new entry for this module:	**ck add my_new_project:my_module:my_data2**
List my_module entries:	**ck list my_module**
Find entries by tags:	**ck search my_module –tags=cool**
Find entry path:	**ck find my_module:my_data**
Obtain entry info (UIDs):	**ck info my_module:my_data**
Rename entry:	**ck ren my_module:my_data2 :my_data3**
Delete entry:	**ck rm my_module:my_data3**
Pack (archive) repository:	**ck zip repo:my_new_project**
Import CK zip repository:	**ck add repo:my_new_project –zip=my_new_project.zip**
Pull existing repo from GitHub:	**ck pull repo:ck-autotuning**
Update all installed CK repos:	**ck pull all**
List modules from this repo:	**ck list ck-autotuning:module:***
Compile program:	**ck compile program:cbench-automotive-susan --speed**
Run program:	**ck run program:cbench-automotive-susan**
Autotune program:	**ck autotune program:cbench-automotive-susan**
Start CK internal web server:	**ck start web**
Start CK web front-end:	**ck browser**

Figure 4: Main CK commands to create new or pull existing repositories, add modules, manage entries, perform actions, and use a local web server.

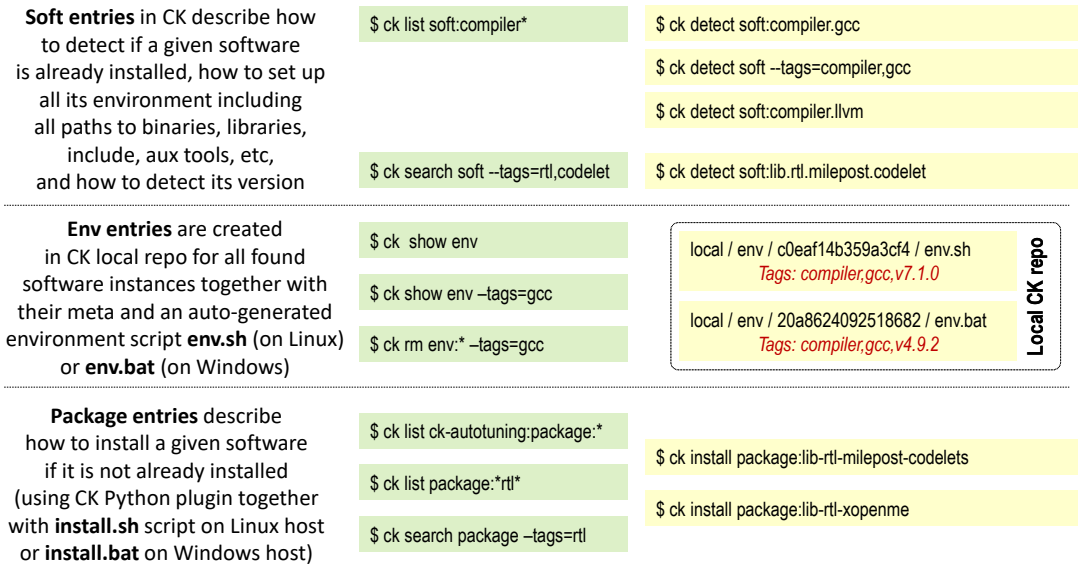


Figure 5: CK modules implementing portable package manager with JSON API to enable cross-platform CK workflows. The community shares CK entries with Python scripts and JSON meta information via Git repositories to describe how to detect, build and install any software. This approach also simplify co-existence of multiple versions of the same tool.

Unified experimental pack in the CK format

.ckr.json	- CK repo name, UID and deps on other CK repos
module/	**program** / module.py – unified CK JSON API (functions: compile, run, autotune)
	dataset
	package
	result
	jnotebook
.cm/	- UIDs for each CK module
program/	zlib
	zlib.cm/meta.json - JSON meta for all CK entries
	zlib/ *.c – program sources
	classify-image
	decode-video-stream
	.cm - UIDs for each CK entry (similar to DOI)
dataset/	image-jpeg-0001
	video-frame-0001
package/	compiler-gcc-7.1.0
	compiler-llvm-4.0
	plugin-llvm-sw-prefetch-pass
	lib-caffe-master-cpu
	lib-tensorflow-master-opencl
result/	cgo2017-paper
	zlib-autotuning-rpi3
jnotebook/	cgo2017-workflow
	cgo2017-graph
	rpi3-gcc-autotuning

Figure 6: Typical experiment pack with reusable and discoverable components shared in the CK format with two level directory structure (module and data).

readable Python dictionaries and JSON files allows users to collaboratively extend actions, API and meta information while keeping backward compatibility. Such approach should allow the community to gradually and collaboratively convert and cross-link all existing ad-hoc code and data into unified components with extensible API and meta information. This, in turn, allows users organize their own research while reusing existing artifacts, building upon them, improving them and continuously contributing back to Collective Knowledge similar to Wikipedia.

We also noticed that CK can help students reduce preparation time for Artifact Evaluation [20] at conferences while automating preparation and validation of experiments since all artifacts, workflows and repositories are immediatelly ready to be shared, ported and plugged in to research workflows.

For example, the highest ranked artifact from the CGO'17 article [36] was implemented and shared using the CK framework [28]. That is why CK is now used and publicly extended by leading companies [62], universities [36] and organizations [114] to encourage, support and simplify technology transfer between academia and industry.

3 Assembling portable and customizable autotuning workflow

Autotuning combined with various run-time adaptation, genetic and machine learning techniques is a popular

approach in computer systems research to automatically explore multi-dimensional design and optimization spaces [113, 32, 46, 94, 89, 49, 112, 77, 66, 108, 50, 78, 105, 84, 80, 56, 96, 73, 43, 75, 38, 88, 92, 103, 70, 90, 86, 42].

CK allows to unify such techniques by developing a common, universal, portable, customizable, multi-dimensional and multi-objective autotuning workflow as a CK module (*pipeline*[1] from the public ck-autotuning repository with the *autotune* function). This allows us to abstract autotuning by decoupling it from the autotuned objects such as *"program"*. Users just need to provide a compatible function *"pipeline"* in a CK module which they want to be autotuned with a specific API including the following keys in both input and output:

- **dependencies** to describe software dependencies via portable package manager from the CK;
- **choices** to expose various design and optimization knobs **c** such as algorithmic parameters, model topology, source-to-source transformations, compiler flags, hardware configurations, etc.;
- **characteristics** to monitor optimized behavior **b** such as execution time, code size, compilation time, energy, memory usage, accuracy, resiliency, costs, etc.;
- **features** to expose various object features **f** such as semantic program and data set features, hardware counters, platform properties, etc.;
- **state** to define run-time system state **s** such as hardware frequencies, network status, cache state, etc.

Autotuning can now be implemented as a universal and extensible workflow applied to any object with a matching JSON API by chaining together related CK modules with various exploration strategies, program transformation tools, compilers, program compilation and execution pipeline, architecture simulators, statistical analysis, Pareto frontier filter and other components, as conceptually shown in Figure 7. Researchers can also use unified machine learning CK modules (wrappers to R and scikit-learn [98]) to model the relationship between **c**, **f**, **s** and the observed behavior **b**, increase coverage, speed up (focus) exploration, and predict efficient optimizations [65, 64]. They can also take take advantage of a universal complexity reduction module which can automatically simplify found solutions without changing their behavior, reduce models and features without sacrificing accuracy, localize performance issues via differential analysis [67], reduce programs to localize bugs, and so on.

[1] We use the term *pipeline* similar to experiments in physics and electronics where an output of one object is chained to an input of another one.

Even more importantly, our concept of a universal autotuning workflow, knowledge sharing and artifact reuse can help teach students how to apply a well-established holistic and top-down experimental methodology from natural sciences to continuously learn and improve the behavior of complex computer systems [62, 65]. Researchers can continue exposing more design and optimization knobs **c**, behavioral characteristics **b**, static and dynamic features **f**, and run-time state *state* to optimize and model behavior of various interconnected objects from the workflow depending on their research interests and autotuning scenarios.

Such scenarios are also implemented as CK modules and describe which sets of choices to select, how to autotune them and which multiple characteristics to trade off. For example, existing scenarios include "autotuning OpenCL parameters to improve execution time", "autotuning GCC flags to balance execution time and code size", "autotune LLVM flags to reduce execution time", "automatically fuzzing compilers to detect bugs", "exploring CPU and GPU frequency in terms of execution time and power consumption", "autotuning deep learning algorithms in terms of speed, accuracy, energy, memory usage and costs", and so on.

You can see some of the autotuning scenarios using the following commands:

```
$ ck pull repo:ck-crowdtuning
$ ck search module --tags="program
optimization"
$ ck list program
```

They can then be invoked from the command line as follows:

```
$ ck autotune program:[CK program alias]
--scenario=[above CK scenario alias]
```

4 Implementing universal compiler flag autotuning

In this section we would like to show how to customize our universal autotuning workflow to tackle an old but yet unsolved problem of finding the most efficient selection of compiler flag which minimizes program size and execution time.

Indeed, the raising complexity of ever changing hardware made development of compilers very challenging. Popular GCC and LLVM compilers nowadays include hundreds of optimizations (Figure 8) and often fail to produce efficient code (execution time and code size) on realistic workloads within a reasonable compilation time [113, 32, 40, 71, 65]. Such large

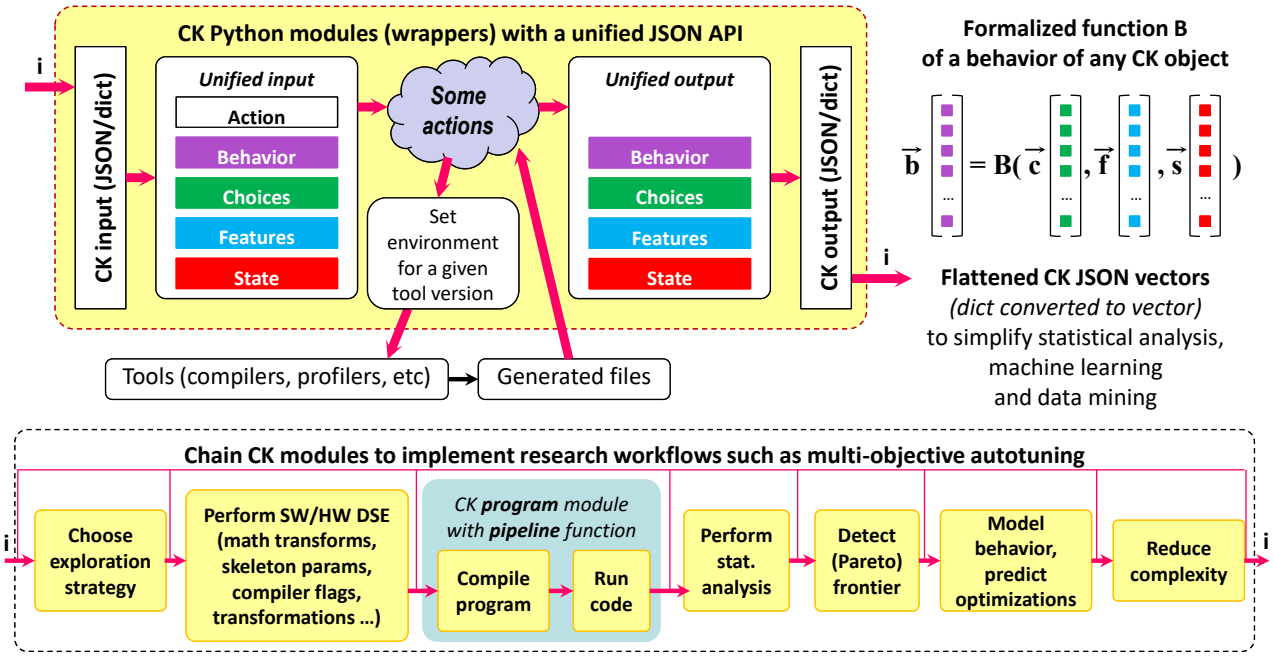


Figure 7: Chaining together various CK modules with JSON API and JSON meta information to implement universal, portable, customizable, multi-dimensional and multi-objective autotuner gradually extended by the community.

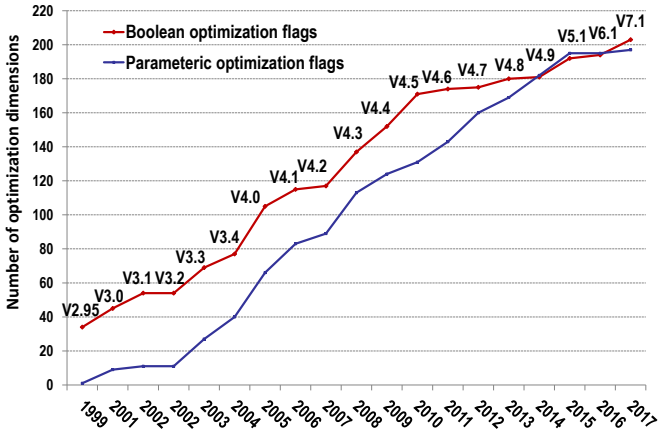


Figure 8: Continuously rising number of boolean and parametric optimization flags in GCC over years (obtained by automatically parsing GCC source code and manual pages, therefore small variation is possible).

design and optimization spaces mean that hardware and compiler designers can afford to explore only a tiny fraction of the whole optimization space using just few ad-hoc benchmarks and data sets on a few architectures in a tough mission to assemble *-O3*, *-Os* and other optimization levels across all supported architectures and workloads.

Our idea is to keep compiler as a simple collection of code analysis and transformation routines and separate it from optimization heuristics. In such case we can use CK autotuning workflow to collaboratively optimize multiple shared benchmarks and realistic workloads across diverse hardware, exchange optimization results, and continuously learn and update compiler optimization heuristics for a given hardware as a compiler plugin. We will demonstrate this approach by randomly optimizing compiler flags for *susan corners* program with aging *GCC 4.9.2*, the latest *GCC 7.1.0* and compare them with *Clang 3.8.1*. We already monitor and optimize execution time and code size of this popular image processing application across different compilers and platforms for many years [61]. That is why we are interested to see if we can still improve it with the CK autotuner on the latest *Raspberry Pi 3 (Model B)* devices (RPi3) extensively used for educational purposes.

First of all, we added *susan* program with *corners* algorithm to the *ctuning-programs* repository with the JSON meta information describing compilation and execution as shown in Figure 9.

We can then test its compilation and execution by invoking the program pipeline as following:

```
$ ck pipeline program:cbench-automotive-susan
```

CK program pipeline will first attempt to detect platform features (OS, CPU, GPU) and embed them to the input dictionary using key *features*. Note that in case of cross-compilation for a target platform different from the host one (Android, remote platform via SSH, etc), it is possible to specify such platform using CK *os* entries and *–target_os=* flag.

```
$ ck pull repo:ctuning-programs
$ ck load program:cbench-automotive-susan --min
{
  "dict": {
    "compile_deps": {
      "compiler": {
        "local": "yes", "name": "C compiler",
        "sort": 10, "tags": "compiler,lang-c"
      },
      "xopenme": {
        "local": "yes", "name": "xOpenME library",
        "sort": 20,  "tags": "lib,xopenme"
      }
    },
    "compiler_env": "CK_CC",
    "extra_ld_vars": "$<<CK_EXTRA_LIB_M>>$",
    "main_language": "c",
    "run_cmds": {
      "corners": {
        "dataset_tags": ["image", "pgm", "dataset"],
        "run_time": {
          "fine_grain_timer_file": "tmp-ck-timer.json",
          "run_cmd_main": "$#BIN_FILE#$
                  $#dataset_path#$$#dataset_filename#$
                  tmp-output.tmp -c",
          "run_correctness_output_files": [
             "tmp-output.tmp", "tmp-output2.tmp"
          ]
        }
      }
    },
    "source_files": ["susan.c"],
    "tags": ["cbench","lang-c","susan","automotive",
              "benchmark","program","small","crowd-tuning"],
    "target_file": "a"
  }
}
```

Figure 9: CK JSON meta information for susan corners (image processing program) to describe software dependencies as well as how to compile and run it.

For example, it is possible to compile and run a given CK program for Android via adb as following:

```
$ ck ls os
$ ck pipeline program:cbench-automotive-susan
--target_os=android21-arm64
```

Next, CK will try to resolve software dependencies and prepare environment for compilation by detecting already installed compilers using CK *soft:compiler.** entries or installing new ones if none was found using CK *package:compiler.**. Each installed compiler for each target will have an associated CK entry with prepared environment to let computer systems researchers work with different versions of different tools:

```
$ ck show env
$ ck show env --target_os=android21-arm64
$ ck show env --tags=compiler
```

Automatically detected version of a selected compiler is used by CK to find and preload all available optimization flags from related *compiler:** entries to the *choices* key of a pipeline input. An example of such flags and tags in the CK JSON format for GCC 4.9 is shown in Figure 10. The community can continue extending such descriptions for different compilers including *GCC, LLVM, Julia, Open64, PathScale, Java, MVCC, ICC and PGI* using either public ck-autotuning repository or their own ones.

Finally, CK program pipeline compiles a given program, runs it on a target platform and fills in sub-dictionary *characteristics* with compilation time, object and binary sizes, MD5 sum of the binary, execution time, used energy (if supported by a used platform), and all other obtained measurements in the common pipeline dictionary.

We are now ready to implement universal compiler flag autotuning coupled with this program pipeline. For a proof-of-concept, we implemented GCC compiler flags exploration strategy which automatically generate N random combinations of compiler flags, compile a given program with each combination, runs it and record all results (inputs and outputs of a pipeline) in a reproducible form in a *local* CK repository using *experiment* module from the ck-analytics repository:

```
$ ck pull repo:ck-crowdtuning
$ ck info
module:experiment.tune.compiler.flags.gcc
```

The JSON meta information of this module describes which keys to select in the program pipeline, how to tune them, and which characteristics to monitor and record as shown in Figure 11. Note that a string starting with ## is used to reference any key in a complex, nested

```
$ ck pull repo:ck-autotuning
$ ck ls compiler | sort
$ ck load compiler:gcc-4.9.0-auto --min
{
  "desc": {
    "all_compiler_flags_desc": {
      "##arch-arm-mfpu-neon": {
        "can_omit": "yes",
        "choice": ["-mfpu=neon",""],
        "desc": "compiler flag (ARM specific): -mfpu=neon",
        "tags": ["basic","optimization","arm-neon"],
        "type": "text"
      },
      "##base_opt": {
        "choice": ["-O3","-O0","-O1","-O2","-Os","-Ofast","-Og"],
        "desc": "base compiler flag",
        "tags": ["base","basic","optimization"],
        "type": "text"
      },
      "##param-align-threshold": {
        "can_omit": "yes",
        "desc": "compiler flag: --param align-threshold= (Select
           fraction of the maximal frequency of executions of
           basic block in function given basic block get alignment)",
        "explore_prefix": "--param align-threshold=",
        "explore_start": 1,  "explore_step": 1,  "explore_stop": 200,
        "tags": ["basic","parametric","optimization"],
        "type": "integer"
      }
      ...
    }
  },
  "dict": {
    "tags": ["compiler","gcc","v4","v4.9","auto"]
  }
}
```

Figure 10: CK JSON description of compiler flags for GCC 4.9 to enable universal autotuning.

```
$ ck pull repo:ck-crowdtuning
$ ck load module:experiment.tune.compiler.flags.gcc –min
{
  "desc": "explore GCC compiler flags",
  "experiment_1_pipeline_update": {
    "choices_order": [   ["##compiler_flags#*"] ],
    "choices_selection": [
      {
        "type": "random",
        "omit_probability": "0.90",
        "tags": "basic,optimization",
        "notags": ""
      }
    ]
  },
  "improvements_keys": [
    "##characteristics#run#execution_time_kernel_0#$#obj#$_imp",
    "##characteristics#compile#obj_size#$#obj#$_imp"
  ],
  "record_keys": [
    "##characteristics#*",  "##features#*", "##choices#*"
  ],
  "solution_conditions": [
    [
      "##characteristics#compile#md5_sum$#objective#$",
      "_imp",  "==", 0
    ]
  ],
  "tags": [
    "program optimization",  "explore",  "program-features",
    "autotuning", "gcc"
  ]
}
```

Figure 11: CK JSON description of random autotuning of compiler flags applied to program pipeline.

JSON or Python dictionary (*CK flat key* [65]). Such *flat key* always starts with *#* followed by *#key* if it is a dictionary key or *@position_in_a_list* if it is a value in a list. CK also supports wild cards in such flat keys such as *"##compiler_flags#"* and *"##characteristics#* to be able to select multiple sub-keys, dictionaries and lists in a given dictionary.

We can now invoke this CK experimental scenario from the command line as following:

```
$ ck autotune program:cbench-automotive-susan
--iterations=300 --repetitions=3
--scenario=experiment.tune.compiler.flags.gcc
--cmd_key=corners
--record_uoa=tmp-susan-corners-gcc4-300-rnd
```

CK will generate 300 random combinations of compiler flags, compile *susan corners* program with each combination, run each produced code 3 times to check variation, and record results in the *experiment:tmp-susan-corners-gcc4-300-rnd*. We can now visualize these autotuning results using the following command line:

```
$ ck plot graph:tmp-susan-corners-gcc4-300-rnd
```

Figure 12 shows manually annotated graph with the outcome of GCC 4.9.2 random compiler flags autotuning applied to susan corners on an RPi3 device in terms

of execution time with variation and code size. Each blue point on this graph is related to one combination of random compiler flags. Red line highlights the frontier of all autotuning results (not necessarily Pareto optimal) to trade off execution time and code size during multi-objective optimization. We also plotted points when default GCC compilation is used without any flags or with *-O3* and *-Os* optimization levels. Finally, we decided to compare optimization results with *Clang 3.8.1* also available on RPi3.

Besides showing that *GCC -O3* (optimization choice **A2**) and *Clang -O3* (optimization choice **A8**) can produce a very similar code, these results confirm well that it is indeed possible to automatically obtain execution time and binary size of *-O3* and *-Os* levels in comparison with non-optimized code within tens to hundreds autotuning iterations (green improvement vectors with 3.6x execution time speedup and 1.6x binary size improvement). The graph also shows that it is possible to improve best optimization level *-O3* much further and obtain 1.3x execution time speedup (optimization solution **A6R** or obtain 11% binary size improvement without sacrifying original execution time (optimization solution **A4R**). Such automatic squeezing of a binary size without sacrificing performance can be very useful for the future IoT devices.

Note that it is possible to browse all results in a user-friendly way via web browser using the following command:

```
$ ck browse
experiment:tmp-susan-corners-gcc4-300-rnd
```

CK will then start internal CK web server available in the ck-web repository, will run a default web browser, and will open a web page with all given experimental results. Each experiment on this page has an associated button with a command line to replay it via CK such as:

```
$ ck replay experiment:7b41a4ac1b3b4f2b
--point=00e81f4e4abb371d
```

CK will then attempt to reproduce this experiment using the same input and then report any differences in the output. This simplifies validation of shared experimental results (optimizations, models, bugs) by the community and possibly with a different software and hardware setup (CK will automatically adapt workflow to a user platform).

We also provided support to help researchers visualize their results as interactive graphs using popular D3.js library as demonstrated in this link.

Looking at above optimization results one may notice that one of the original optimization solutions on a frontier **A4** has 40 optimization flags, while **A4R** only 7 as shown in Table 1. The natural reason is that not all randomly selected flags contribute to improvements. That is why we developed a simple and universal complexity reduction algorithm which iteratively and randomly removes selected those choices from a found solution which do not influence monitored characteristics such as execution time and code size in our example.

Such complexity reduction (pruning) of an existing solution can be invoked as following (flag *–prune_md5* tells CK to exclude a given choice without running code if MD5 of a produced binary didn't change, thus considerably speeding up flag pruning):

```
$ck replay experiment:93974bf451f957eb
--point=74e9c9f14b424ba7 --prune --prune_md5
@prune.json
```

The *'prune.json'* file describes conditions on program pipeline keys when a given choice should be removed as shown in Figure 13.

Such universal complexity reduction approach helps software engineers better understand individual contribution of each flag to improvements or degradations of all monitored characteristics such as execution time and code size as shown in Figure 14.

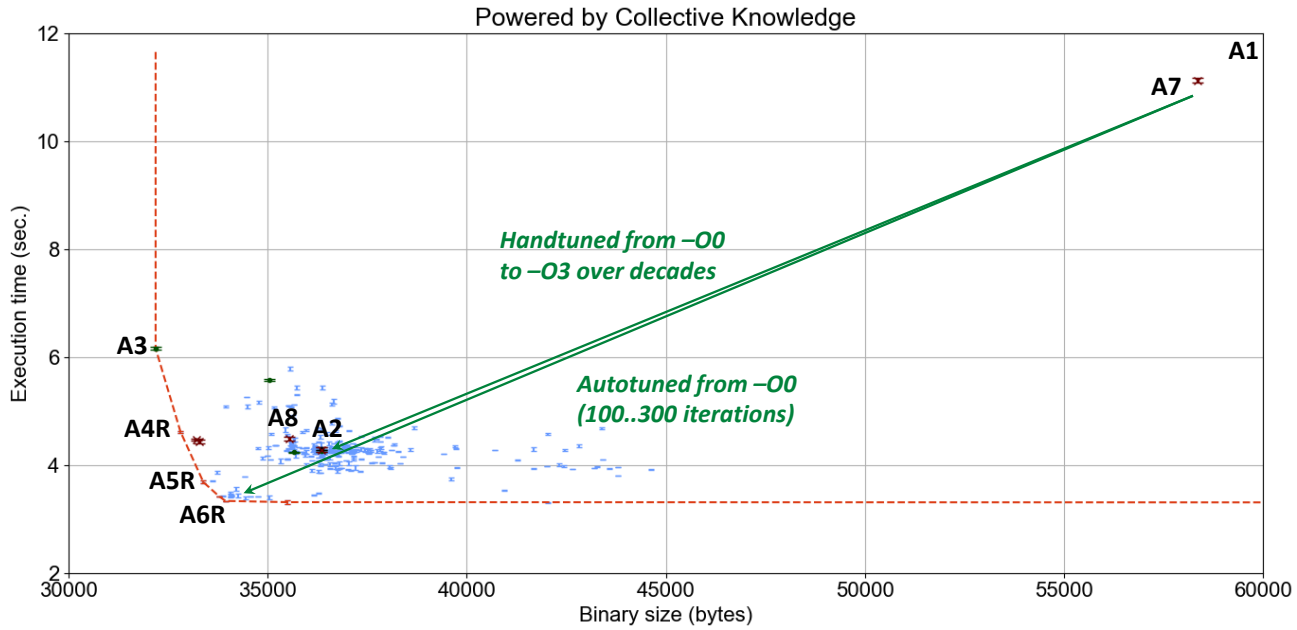
Asked by compiler developers, we also provided an extension to our complexity reduction module to turn off explicitly all available optimization choices one by one if they do not influence found optimization result. Table 2 demonstrates this approach and shows all compiler optimizations contributing to the found optimization solution. It can help improve internal optimization heuristics, global optimization levels such as *-O3*, and improve machine learning based optimization predictions. This extension can be invoked by adding flags *–prune_invert –prune_invert_do_not_remove_key* when reducing complexity of a given solution such as:

```
$ ck replay experiment:93974bf451f957eb
--point=74e9c9f14b424ba7 --prune --prune_md5
--prune_invert --prune_invert_do_not_remove_key
@prune.json
```

We have been analyzing already aging *GCC 4.9.2* because it is still the default compiler for Jessy Debian distribution on RPi3. However, we would also like to check how our universal autotuner works with the latest *GCC 7.1.0*.

Since there is no yet a standard Debian GCC 7.1.0 package available for RPi3, we need to build it from scratch. This is not a straightforward task since we have to pick up correct configuration flags which will adapt GCC build to quite outdated RPi3 libraries. However, once we manage to do it, we can automate this process using CK *package* module.

We created a public ck-dev-compilers repository to automate building and installation of various compilers including GCC and LLVM via CK. It is therefore possible to install GCC 7.1.0 on RPi3 as following (see Appendix or GitHub repository ReadMe file for more details):



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A1	GCC 4.9.2	11.7 ± 0.0	60560	
A2	GCC 4.9.2	4.3 ± 0.1	36360	-O3
A3	GCC 4.9.2	6.2 ± 0.1	32184	-Os
A4R	GCC 4.9.2	4.2 ± 0.0	32448	-O3 -fno-guess-branch-probability -fno-if-conversion -fno-ivopts -fno-schedule-insns -fsingle-precision-constant –param max-unswitch-insns=5
A5R	GCC 4.9.2	3.7 ± 0.1	33376	-O3 -fbranch-probabilities -fno-ivopts -fno-sched-dep-count-heuristic
A6R	GCC 4.9.2	3.4 ± 0.0	33804	-O3 -fno-inline-small-functions -fno-ivopts -fno-tree-partial-pre
A7	CLANG 3.8.1	11.1 ± 0.1	58368	
A8	CLANG 3.8.1	4.5 ± 0.1	35552	-O3

Figure 12: Results of GCC 4.9.2 random compiler flag autotuning of susan corners program on Raspberry Pi 3 (Model B) device using CK with a highlighted frontier (trading-off execution time and code size) and best found combinations of flags on this frontier.

ID	Flags
A4	-O3 -fira-algorithm=priority -fcaller-saves -fno-devirtualize-speculatively -fno-function-cse -fgcse-sm -fno-guess-branch-probability -fno-if-conversion -fno-inline-functions-called-once -fipa-reference -fno-ira-loop-pressure -fira-share-save-slots -fno-isolate-erroneous-paths-dereference -fno-ivopts -floop-nest-optimize -fmath-errno -fmove-loop-invariants -fsched-last-insn-heuristic -fsched2-use-superblocks -fno-schedule-insns -fno-signed-zeros -fsingle-precision-constant -fno-tree-sink -fno-unsafe-loop-optimizations –param asan-instrument-reads=1 –param gcse-unrestricted-cost=5 –param l1-cache-size=11 –param large-function-growth=33 –param loop-invariant-max-bbs-in-loop=636 –param max-completely-peel-loop-nest-depth=7 –param max-delay-slot-live-search=163 –param max-gcse-insertion-ratio=28 –param max-inline-insns-single=282 –param max-inline-recursive-depth-auto=0 –param max-jump-thread-duplication-stmts=6 –param max-last-value-rtl=4062 –param max-pipeline-region-insns=326 –param max-sched-region-blocks=17 –param max-tail-merge-iterations=2 –param max-unswitch-insns=5 –param max-vartrack-expr-depth=6 –param min-spec-prob=1 –param omega-eliminate-redundant-constraints=1 –param omega-max-keys=366 –param omega-max-wild-cards=36 –param sms-dfa-history=0
A4R	-O3 -fno-guess-branch-probability -fno-if-conversion -fno-ivopts -fno-schedule-insns -fsingle-precision-constant –param max-unswitch-insns=5

Table 1: One of original optimization solutions found after autotuning with random selection of compiler flags (A4) and reduced optimization solution (A4R) which results in the same or better execution time and code size.

```
{
  "prune_print_keys":[
     "##characteristics#run#execution_time_kernel_0#min",
     "##characteristics#compile#binary_size#min",
     "##characteristics#compile#md5_sum#min"
  ],
  "prune_conditions": [
    ["##characteristics#run#execution_time_kernel_0$#objective#$",
      "_imp", ">", 0.99],
    ["##characteristics#compile#binary_size$#objective#$",
     "_imp", ">=", 1]
  ]
}
```

Figure 13: CK JSON description of conditions on choices in a pipeline input to reduce choices from a found optimization solution.

```
Characteristics' changes in brackets:
  0 = ##characteristics#run#execution_time_kernel_0#min_imp
        1 = ##characteristics#compile#binary_size#min_imp

(          )  : -O3
(0.962;0.987) : -fno-guess-branch-probability
(0.998;1.000) : -fno-if-conversion
(0.848;0.948) : -fno-ivopts
(1.226;0.972) : -fno-schedule-insns
 (0.988;0.998) : -fsingle-precision-constant
 (1.002;0.996) : --param max-unswitch-insns=5
```

Figure 14: Contribution of individual compiler flags to improvements or degradations of monitored characteristics during universal complexity reduction.

```
$ ck pull repo:ck-dev-compilers
$ ck install
package:compiler-gcc-any-src-linux-no-deps
--env.PARALLEL_BUILDS=1
--env.GCC_COMPILE_CFLAGS=-O0
--env.GCC_COMPILE_CXXFLAGS=-O0
--env.EXTRA_CFG_GCC=--disable-bootstrap
--env.RPI3=YES --force_version=7.1.0
```

This CK package has an *install.sh* script which is customized using environment variables or *–env* flags to build GCC for a target platform. The JSON meta data of this CK package provides optional software dependencies which CK has to resolve before installation (similar to CK compilation). If installation succeeded, you should be able to see two prepared environments for GCC 4.9.2 and GCC 7.1.0 which now co-exist in the system.

```
$ ck show env --tags=gcc
```

Whenever we now invoke CK autotuning, CK software and package manager will detect multiple available versions of a required software dependency and will let you choose which compiler version to use.

Let us now autotune the same *susan corners* program by generating 300 random combinations of *GCC 7.1.0* compiler flags and record results in the *experiment:tmp-susan-corners-gcc7-300-rnd*:

```
$ ck autotune program:cbench-automotive-susan
--iterations=300 --repetitions=3
--scenario=experiment.tune.compiler.flags.gcc
--cmd_key=corners
--record_uoa=tmp-susan-corners-gcc7-300-rnd
```

Figure 15 shows results of such *GCC 7.1.0* compiler flag autotuning (**B** points) and compares them against

ID	Flags
A6R	-O3 -fno-inline-small-functions -fno-ivopts -fno-tree-partial-pre
A6RI	**-O3** -fno-inline-small-functions -fno-ivopts -fno-tree-bit-ccp -fno-tree-partial-pre -fno-tree-pta -fno-associative-math -fno-auto-inc-dec -fno-branch-probabilities -fno-branch-target-load-optimize -fno-branch-target-load-optimize2 -fno-caller-saves -fno-check-data-deps -fno-combine-stack-adjustments -fno-conserve-stack -fno-compare-elim **-fcprop-registers -fcrossjumping -fcse-follow-jumps** -fno-cse-skip-blocks -fno-cx-limited-range -fno-data-sections **-fdce** -fno-delayed-branch -fno-devirtualize -fno-devirtualize-speculatively -fno-early-inlining -fno-ipa-sra -fno-expensive-optimizations -fno-fat-lto-objects -fno-fast-math -fno-finite-math-only -fno-float-store **-fforward-propagate** -fno-function-sections -fno-gcse-after-reload -fno-gcse-las -fno-gcse-lm -fno-graphite-identity -fno-gcse-sm -fno-hoist-adjacent-loads -fno-if-conversion **-fif-conversion2** -fno-indirect-inlining -fno-inline-functions -fno-inline-functions-called-once -fno-ipa-cp -fno-ipa-cp-clone -fno-ipa-pta **-fipa-pure-const** -fno-ipa-reference -fno-ira-hoist-pressure -fno-ira-loop-pressure -fno-ira-share-save-slots **-fira-share-spill-slots -fisolate-erroneous-paths-dereference** -fno-isolate-erroneous-paths-attribute -fno-keep-inline-functions -fno-keep-static-consts -fno-live-range-shrinkage -fno-loop-block -fno-loop-interchange -fno-loop-strip-mine -fno-loop-nest-optimize -fno-loop-parallelize-all -fno-lto -fno-merge-all-constants -fno-merge-constants -fno-modulo-sched -fno-modulo-sched-allow-regmoves **-fmove-loop-invariants** -fno-branch-count-reg -fno-defer-pop -fno-function-cse **-fguess-branch-probability -finline -fmath-errno** -fno-peephole **-fpeephole2** -fno-sched-interblock -fno-sched-spec -fno-signed-zeros -fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss **-fomit-frame-pointer** -fno-optimize-sibling-calls -fno-partial-inlining -fno-peel-loops -fno-predictive-commoning -fno-prefetch-loop-arrays -fno-ree -fno-rename-registers **-freorder-blocks** -fno-reorder-blocks-and-partition -fno-rerun-cse-after-loop -fno-reschedule-modulo-scheduled-loops -fno-rounding-math -fno-sched2-use-superblocks **-fsched-pressure** -fno-sched-spec-load -fno-sched-spec-load-dangerous -fno-sched-group-heuristic **-fsched-critical-path-heuristic** -fno-sched-spec-insn-heuristic -fno-sched-rank-heuristic -fno-sched-dep-count-heuristic **-fschedule-insns -fschedule-insns2** -fno-section-anchors -fno-selective-scheduling -fno-selective-scheduling2 -fno-sel-sched-pipelining -fno-sel-sched-pipelining-outer-loops -fno-shrink-wrap -fno-signaling-nans -fno-single-precision-constant -fno-split-ivs-in-unroller -fno-split-wide-types -fno-strict-aliasing **-fstrict-overflow** -fno-tracer -fno-tree-builtin-call-dce -fno-tree-ccp **-ftree-ch** -fno-tree-coalesce-vars -fno-tree-copy-prop **-ftree-copyrename -ftree-dce -ftree-dominator-opts** -fno-tree-dse **-ftree-forwprop** -fno-tree-fre -fno-tree-loop-if-convert -fno-tree-loop-if-convert-stores **-ftree-loop-im** -fno-tree-phiprop -fno-tree-loop-distribution -fno-tree-loop-distribute-patterns -fno-tree-loop-linear **-ftree-loop-optimize** -fno-tree-loop-vectorize -fno-tree-pre **-ftree-reassoc** -fno-tree-sink **-ftree-slsr -ftree-sra** -fno-tree-switch-conversion -fno-tree-tail-merge **-ftree-ter** -fno-tree-vectorize **-ftree-vrp** -fno-unit-at-a-time -fno-unroll-all-loops -fno-unroll-loops -fno-unsafe-loop-optimizations -fno-unsafe-math-optimizations -fno-unswitch-loops -fno-variable-expansion-in-unroller -fno-vect-cost-model -fno-vpt -fno-web -fno-whole-program -fno-wpa **-fexcess-precision=standard -ffp-contract=off -fira-algorithm=CB -fira-region=all**

Table 2: Explicitly switching off all compiler flags one by one if they do not influence optimization result - useful to understand all compiler optimizations which contributed to the found solution.

GCC 4.9.2 (**A** points). Note that this graph is also available in interactive form online.

It is interesting to see considerable improvement in execution time of susan corners when moving from GCC 4.9 to GCC 7.1 with the best optimization level *-O3*. This graph also shows that new optimization added during past 3 years opened up many new opportunities thus considerably expanding autotuning frontier (light red dashed line versus dark red dashed line). Autotuning only managed to achieve a modest improvement of a few percent over *-O3*.

On the other hand, GCC *-O3* and *-Os* are still far from achieving best trade-offs for execution time and code size. For example, it is still possible to improve program binary size by 10% (reduced solution **B4R**) without degrading best achieved execution time with the *-O3* level (**-O3**), or improve execution time of **-Os** level by 28% while slightly degrading code size by 5%.

Note that for readers' convenience we added scripts to reproduce and validate all results from this section to the following CK entries:

```
$ ck pull repo:ck-rpi-optimization-results
$ ck find script:rpi3-susan*
```

These results confirm that it is difficult to manually prepare compiler optimization heuristic which can deliver good trade offs between execution time and code size in such a large design and optimization spaces. They also suggest that either susan corners or similar code was eventually added to the compiler regression testing suite, or some engineer check it manually and fixed compiler heuristic. However, there is also no guarantee that future GCC versions will still perform well on the susan corners program. Neither these results guarantee that GCC 7.1.0 will perform well on other realistic workloads or devices.

5 Crowdsourcing autotuning

We use our universal CK autotuning workflow to teach students and end-users how to automatically find good trade offs between multiple characteristics for any individual program, data set, compiler, environment and hardware. At the same time, automatically tuning many realistic workloads is very costly and can easily take from days to weeks and months [61].

Common experimental frameworks can help tackle this problem too by crowdsourcing autotuning across diverse hardware provided by volunteers and combining it with online classification, machine learning and run-time adaptation [58, 75, 64]. However, our previous frameworks did not cope well with "big data" problem (cTuning framework [58, 60] based on MySQL database) or were too "heavy" (Collective Mind aka cTuning 3 framework [65]).

Extensible CK workflow framework combined with our cross-platform package manager, internal web server and machine learning, helped solve most of the above issues. For example, we introduced a notion of a remote repository in the CK - whenever such repository is accessed CK simply forward all JSON requests to an appropriate web server.

CK always has a default remote repository *remote-ck* connected with a public optimization repository running CK web serve at `cKnowledge.org/repo`:

```
$ ck load repo:remote-ck --min
```

For example, one can see publicly available experiments from command line as following:

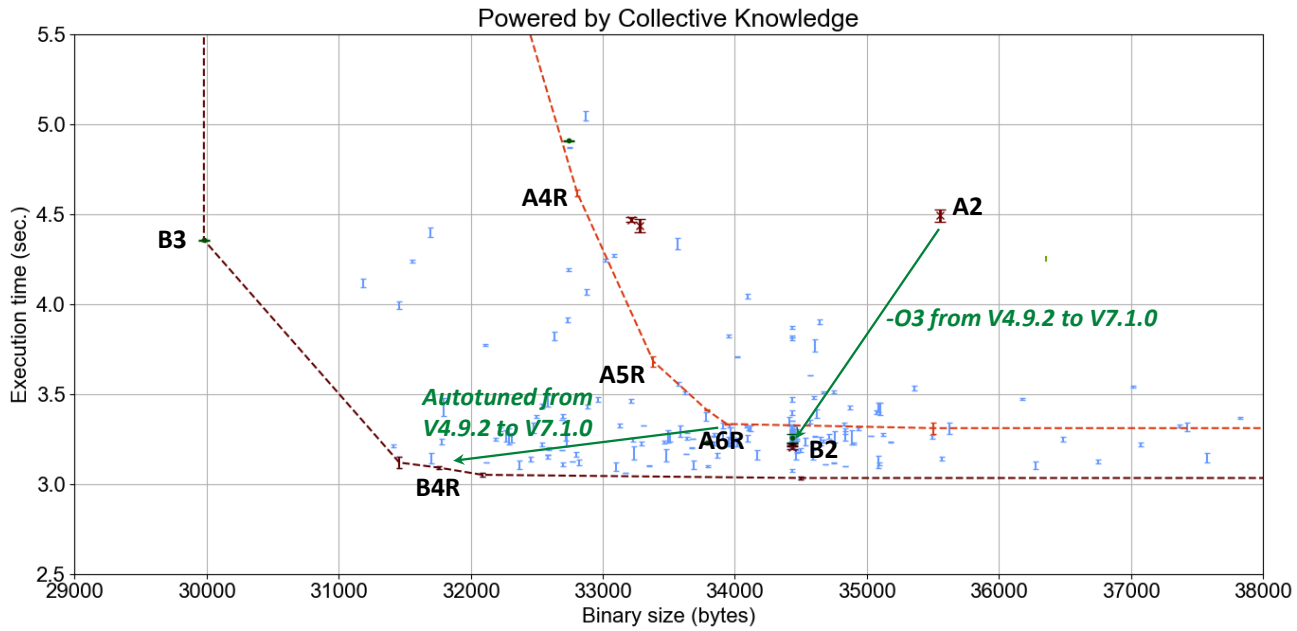
```
$ ck list remote-ck:experiment:* | sort
```

Such organization allows one to crowdsource autotuning, i.e. distributing autotuning of given shared workloads in a cloud or across diverse platforms simply by using remote repositories instead of local ones. On the other hand, it does not address the problem of optimizing larger applications with multiple hot spots. It also does not solve the "big data" problem when large amount of data from multiple participants needed for reproducibility will be continuously aggregated in a CK server.

However, we have been already addressing the first problem by either instrumenting, monitoring and optimizing hot code regions in large applications using our small "XOpenME" library, or even extracting such code regions from a large application with a run-time data set and registering them in the CK as standalone programs (codelets or computational species) as shown in Figure 16 ([65]).

In the MILEPOST project [61] we used a proprietary "codelet extractor" tool from CAPS Entreprise (now dissolved) to automatically extract such hot spots with their data sets from several real software projects and 8 popular benchmark suits including NAS, MiBench, SPEC2000, SPEC2006, Powerstone, UTDSP and SNU-RT. We shared those of them with a permissive license as CK programs in the ctuning-programs repository to be compatible with the presented CK autotuning workflow. We continue adding real, open-source applications and libraries as CK program entries (GEMM, HOG, SLAM, convolutions) or manually extracting and sharing interesting code regions from them with the help of the community. Such a large collection of diverse and realistic workloads should help make computer systems research more applied and practical.

As many other scientists, we also faced a big data problem when continuously aggregating large amounts of raw optimization data during crowd-tuning for further processing including machine learning [60]. We managed to solve this problem in the CK by using online pre-processing of raw data and online classification to record only the most efficient



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A2	GCC 4.9.2	4.3 ± 0.1	36360	-O3
A5R	GCC 4.9.2	3.7 ± 0.1	33376	-O3 -fbranch-probabilities -fno-ivopts -fno-sched-dep-count-heuristic
A6R	GCC 4.9.2	3.4 ± 0.0	33804	-O3 -fno-inline-small-functions -fno-ivopts -fno-tree-partial-pre
B1	GCC 7.1.0	11.5 ± 0.0	58008	
B2	GCC 7.1.0	3.2 ± 0.0	34432	-O3
B3	GCC 7.1.0	4.4 ± 0.0	29980	-Os
B4	GCC 7.1.0	3.1 ± 0.1	31460	-O3 -fno-cx-fortran-rules -fno-devirtualize -fno-expensive-optimizations -fno-if-conversion -fira-share-save-slots -fno-ira-share-spill-slots -fno-ivopts -fno-loop-strip-mine -finline -fno-math-errno -frounding-math -fno-sched-rank-heuristic -fno-sel-sched-pipelining-outer-loops -fno-semantic-interposition -fsplit-wide-types -fno-tree-ccp -ftree-dse
B4R	GCC 7.1.0	3.1 ± 0.1	31420	-O3 -fno-expensive-optimizations -fno-ivopts -fno-math-errno

Figure 15: Results of GCC 7.1.0 random compiler flag autotuning of susan corners program on Raspberry Pi 3 (Model B) device using CK with a highlighted frontier (trading-off execution time and code size), best combinations of flags on this frontier, and comparison with the results from GCC 4.9.2.

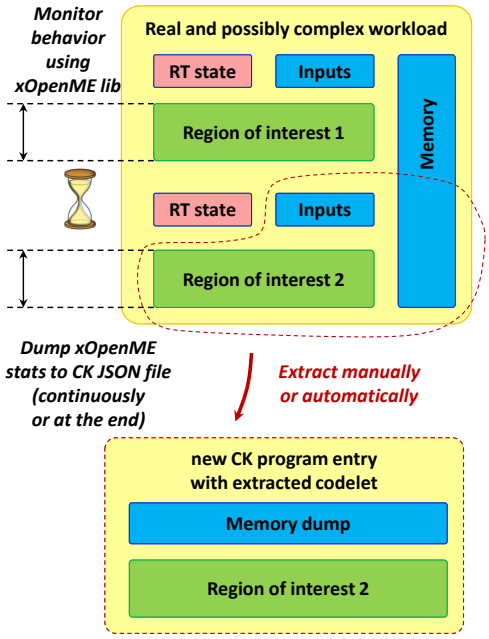


Figure 16: Preparing larger applications such as Firefox and Chrome for CK-based autotuning: a) instrumenting, monitoring and optimizing hot code regions using "XOpenME" library b) extracting code regions from a large application with a run-time data set and register them in the CK as standalone programs (codelets)

optimization solutions (on a frontier in case of multi-objective autotuning) along with unexpected behavior (bugs and numerical instability) [64]. It is now possible to invoke crowd-tuning of GCC compiler flags (improving execution time) in the CK as following:

```
$ ck crowdtune program --iterations=50
--scenario=8289e0cf24346aa7
```

or

```
$ ck crowdsource program.optimization
--iterations=50 --scenario=8289e0cf24346aa7
```

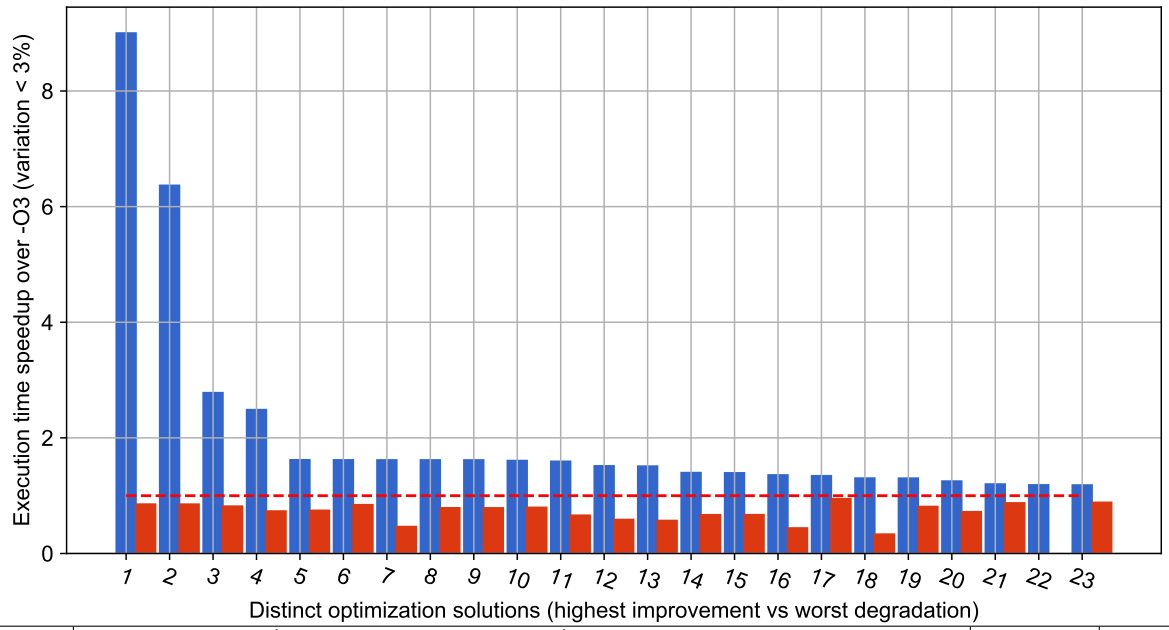
In contrast with traditional autotuning, CK will first query *remote-ck* repository to obtain all most efficient optimization choices aka solutions (combinations of random compiler flags in our example) for a given trade-off scenario (GCC compiler flag tuning to minimize execution time), compiler version, platform and OS. CK will then select a random CK program (computational species), compiler and run it with all these top optimizations, and then try N extra random optimizations (random combinations of GCC flags) to continue increasing design and optimization space coverage. CK will then send the highest improvements of monitored characteristics (execution time in our example) achieved for each optimization solution as well as worst degradations back to a public server. If a new optimization solution if also found during random autotuning, CK will assign it a unique ID (*solution_uid* and will record it in a public repository. At the public server side, CK will merge improvements and degradations for a given program from a participant with a global statistics while recording how many programs achieved the highest improvement (best species) or worst degradation (worst species) for a given optimization as shown in Figure 17.

This figure shows a snapshot of public optimization results with top performing combinations of GCC 4.9.2 compiler flags on RPi3 devices which minimize execution time of shared CK workloads (programs and data sets) in comparison with *-O3* optimization level. It also shows the highest speedup and the worse degradation achieved across all CK workloads for a given optimization solution, as well as number of workloads where this solution was the best or the worst (online classification of all optimization solutions). Naturally this snapshot automatically generated from the public repository at the time of publication may slightly differ from continuously updated live optimization results available at this link. These results confirm that GCC 4.9.2 misses many optimization opportunities not covered by *-O3* optimization level.

Figure 18 with optimization results for GCC 7.1.0 also confirms that this version was considerably improved in comparison with GCC 4.9.2 (latest live results are available in our public optimization repository at this link): there are fewer efficient optimization solutions found during crowd-tuning 14 vs 23 showing the overall improvement of the *-O3* optimization level.

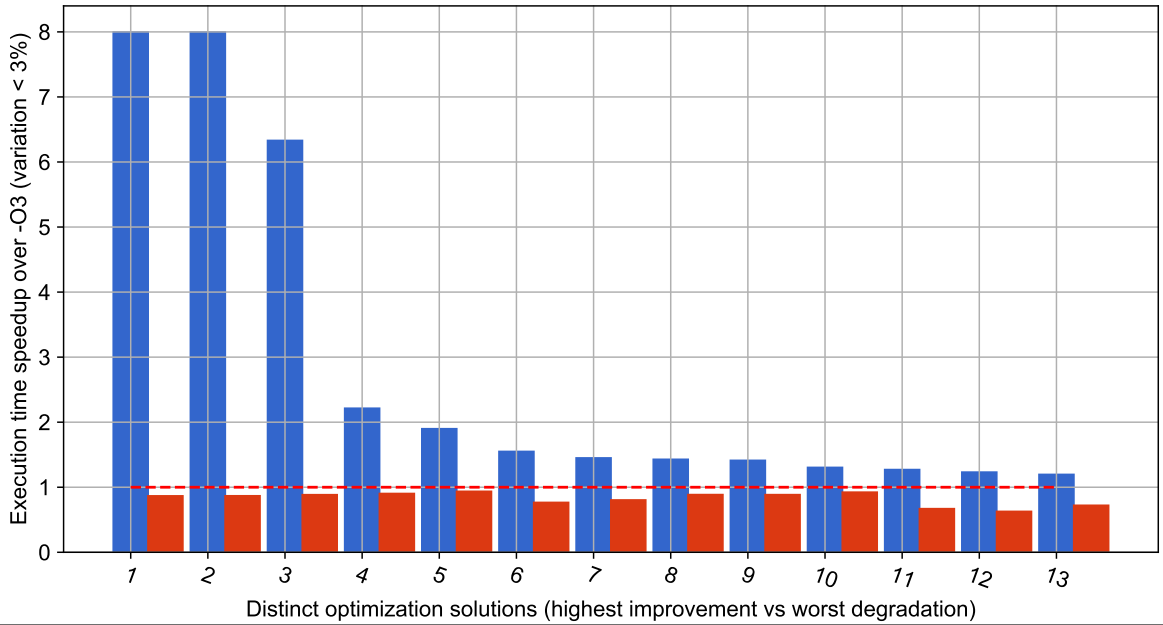
Nevertheless, GCC 7.1.0 still misses many optimization opportunities simply because our long-term experience suggests that it is infeasible to prepare one universal and efficient optimization heuristics with good multi-objective trade-offs for all continuously evolving programs, data sets, libraries, optimizations and platforms. That is why we hope that our approach of combining a common workflow framework adaptable to software and hardware changes, public repository of optimization knowledge, universal and collaborative autotuning across multiple hardware platforms (e.g. provided by volunteers or by HPC providers), and community involvement should help make optimization and testing of compilers more automatic and sustainable [64, 62]. Rather than spending considerable amount of time on writing their own autotuning and crowd-tuning frameworks, students and researchers can quickly reuse shared workflows, reproduce and learn already existing optimizations, try to improve optimization heuristics, and validate their results by the community.

Furthermore, besides using *-Ox* compiler levels, academic and industrial users can immediately take advantage of various shared optimizations solutions automatically found by volunteers for a given compiler



Solution	Pruned flags (complexity reduction)	Best species	Worst species
1	-O3 -flto	6	3
2	-O3 -fno-inline -flto	1	1
3	-O3 -fno-if-conversion2 -funroll-loops	2	1
4	-O3 -fpeel-loops -ftracer	1	3
5	-O3 -floop-nest-optimize -fno-sched-interblock -fno-tree-copy-prop -funroll-all-loops	4	1
6	-O3 -funroll-loops	2	3
7	-O3 -floop-strip-mine -funroll-loops	1	1
8	-O3 -fno-inline -fno-merge-all-constants -fno-tree-ccp -funroll-all-loops	2	3
9	-O3 -fno-tree-loop-if-convert -funroll-all-loops	3	2
10	-O3 -fno-section-anchors -fselective-scheduling2 -fno-tree-forwprop -funroll-all-loops	2	2
11	-O3 -fno-ivopts -funroll-loops	4	1
12	-O3 -fno-tree-ch -funroll-all-loops	1	1
13	-O3 -fno-move-loop-invariants -fno-tree-ch -funroll-loops	1	2
14	-O3 -fira-algorithm=priority -fno-ivopts	1	2
15	-O3 -fno-ivopts	2	4
16	-O3 -fno-sched-spec -fno-tree-ch	1	2
17	-O3 -fno-ivopts -fselective-scheduling -fwhole-program	1	1
18	-O3 -fno-omit-frame-pointer -fno-tree-loop-optimize	1	4
19	-O3 -fno-auto-inc-dec -ffinite-math-only	1	2
20	-O3 -fno-guess-branch-probability -fira-loop-pressure -fno-toplevel-reorder	1	5
21	-O3 -fselective-scheduling2 -fno-tree-pre	2	2
22	-O3 -fgcse-sm -fno-move-loop-invariants -fno-tree-forwprop -funroll-all-loops -fno-web	1	0
23	-O3 -fno-schedule-insns -fselective-scheduling2	1	2

Figure 17: Snapshot of top performing combinations of GCC 4.9.2 compiler flags together with highest speedups and worst degradations achieved across all shared CK workloads on RPi3.



Solution	Pruned flags (complexity reduction)	Best species	Worst species
1	-O3 -fno-delayed-branch -flto -fno-selective-scheduling2 -fno-whole-program	6	0
2	-O3 -flto	4	1
3	-O3 -fno-inline -flto	2	1
4	-O3 -fno-cprop-registers -flto -funroll-all-loops	3	1
5	-O3 -fno-tree-fre -funroll-all-loops	2	1
6	-O3 -fno-predictive-commoning -fno-schedule-insns -funroll-loops	3	3
7	-O3 -funroll-loops	3	0
8	-O3 -fno-tree-ter -funroll-all-loops	3	1
9	-O3 -fno-merge-all-constants -fselective-scheduling2 -funroll-loops	1	0
10	-O3 -fno-devirtualize-at-ltrans -fno-predictive-commoning -fno-tree-pre	1	2
11	-O3 -fcheck-data-deps -fira-loop-pressure -fno-isolate-erroneous-paths-dereference -fno-sched-dep-count-heuristic -fsection-anchors -fsemantic-interposition -fno-tree-ch -fno-tree-loop-linear -fno-tree-partial-pre	2	2
12	-O3 -fno-schedule-insns -ftracer	2	3
13	-O3 -fno-auto-inc-dec -fguess-branch-probability -fipa-pure-const -freorder-blocks -fselective-scheduling2 -ftree-ccp -fno-tree-pre -ftree-tail-merge	1	1

Figure 18: Snapshot of top performing combinations of GCC 7.1.0 compiler flags together with highest speedups and worst degradations achieved across all shared CK workloads on RPi3.

and hardware via CK using *solution_uid* flag. For example, users can test the most efficient combination of compiler flags which achieved the highest speedup for GCC 7.1.0 on RPi3 (see "Copy CID to clipboard for a given optimization solution at this link) for their own programs using CK:

```
$ ck benchmark program:{new program}
  --shared_solution_cid=27bc42ee449e880e:
  79bca2b76876b5c6-8289e0cf24346aa7-
  f49649288ab0accd
```

or

```
$ ck benchmark program:{new program}
  -O27bc42ee449e880e:79bca2b76876b5c6-
  8289e0cf24346aa7-f49649288ab0accd
```

6 Autotuning and crowd-tuning real workloads

In this section we would like to show how we can apply universal autotuning and collaboratively found optimization solutions to several popular workloads used by RPi community: *zlib decode, zlib encode, 7z encode, aubio, ccrypt, gzip decode, gzip encode, minigzip decode, minigzip encode, rhash, sha512sum, unrar*. We added the latest versions of these real programs to the CK describing how to compile and run them using CK JSON meta data:

```
$ ck ls ck-rpi-optimization:program:*
```

We can now autotune any of these programs via CK as described in Section 4. For example, the following command will autotune *zlib decode* workload with 150 random combinations of compiler flags including parametric and architecture specific ones, and will record results in a local repository:

```
$ ck autotune program:zlib --cmd_key=decode
--iterations=150 --repetitions=3
--scenario=experiment.tune.compiler.flags.gcc
--parametric_flags --cpu_flags --base_flags
--record_uoa=tmp-rpi3-zlib-decode-gcc4-150bpc-rnd
```

Figure 19 (link with interactive graph) shows manually annotated graph with the outcome of such autotuning when using GCC 4.9.2 compiler on RPi3 device in terms of execution time with variation and code size. Each blue point on this graph is related to one combination of random compiler flags. Red line highlights the frontier of all autotuning results to let users trade off execution time and code size during multi-objective optimization. Similar to graphs in Section 4, we also plotted points when using several main GCC and Clang optimization levels.

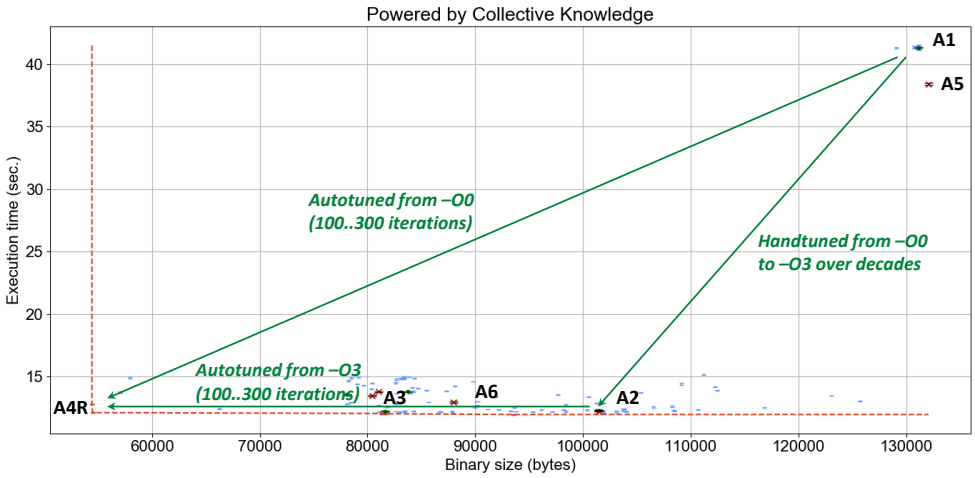
In contrast with *susan corners* workload, autotuning did not improve execution time of *zlib decode* over *-O3* level most likely because this algorithm is present in many benchmarking suits. On the other hand, autotuning impressively improved code size over *-O3* by nearly 2x without sacrificing execution time, and by 1.5x with 11% execution time improvement over *-Os* (reduced optimization solution **A4R**), showing that code size optimization is still a second class citizen.

Since local autotuning can still be quite costly (150 iterations to achieve above results), we can now first check 10..20 most efficient combinations of compiler flags already found and shared by the community for this compiler and hardware (Figure 17). Note that programs from this section did not participate in crowd-tuning to let us have a fair evaluation of the influence of shared optimizations on these programs similar to leave-one-out cross-validation in machine learning.

Figure 20 shows "reactions" of *zlib decode* to these optimizations in terms of execution time and code size (online interactive graph). We can see that crowd-tuning solutions indeed cluster in a relatively small area close to *-O3* with one collaborative solution (**C1**) close to the best optimization solution found during lengthy autotuning (**A4R**) thus providing a good trade off between autotuning time, execution time and code size.

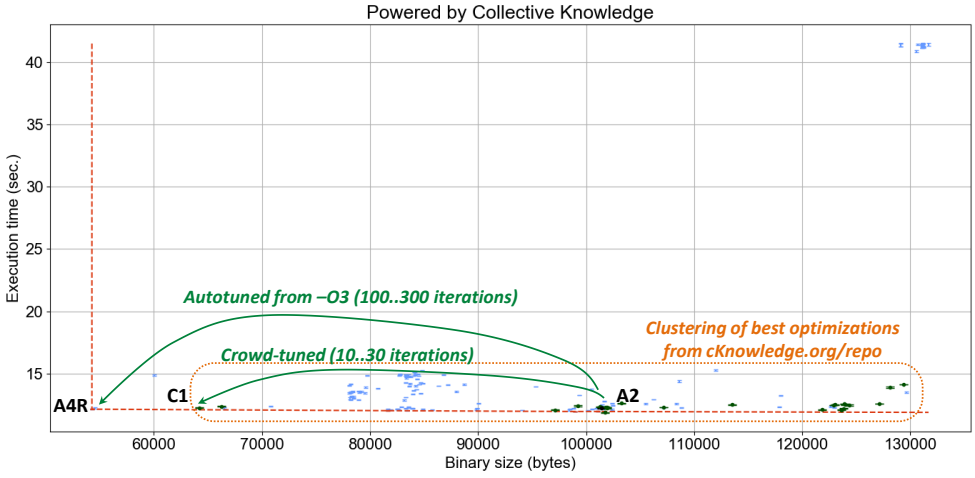
Autotuning *zlib decode* using *GCC 7.1.0* revels even more interesting results in comparison with *susan corners* as shown in Figure 21 (online interactive graph). While there is practically no execution time improvements when switching from *GCC 4.9.2* to *GCC 7.1.0* on *-O3* and *-Os* optimization levels, *GCC 7.1.0 -O3* considerably degraded code size by nearly 20%. Autotuning also shows few opportunities on *GCC 7.1.0* in comparison with *GCC 4.9.2* where best found optimization **B4R** is worse in terms of code size than **A4R** also by around 20%. These results highlight issues which both end-users and compiler designers face when searching for efficient combinations of compiler flags or preparing the default optimization levels -Ox.

CK crowd-tuning can assist in this case too - Figure 22 shows reactions of *zlib decode* to the most efficient combinations of GCC 7.1.0 compiler flags shared by the community for RPi3 (online interactive graph). Shared optimization solution **C2** achieved the same results in terms of execution time and code size as reduced solution **B4R** found during 150 random autotuning iterations. Furthermore, another shared optimization solution **C1** improved code size by 15% in comparison with GCC 7.1.0 autotuning solution **B4R** and is close to the best solution GCC 4.9.2 autotuning solution **A4R**. These results suggest that 150 iterations with random combinations of compiler flags may not be enough to find efficient solution for *zlib decode*. In turn, crowd-tuning can help considerably accelerate and focus such optimization space exploration.



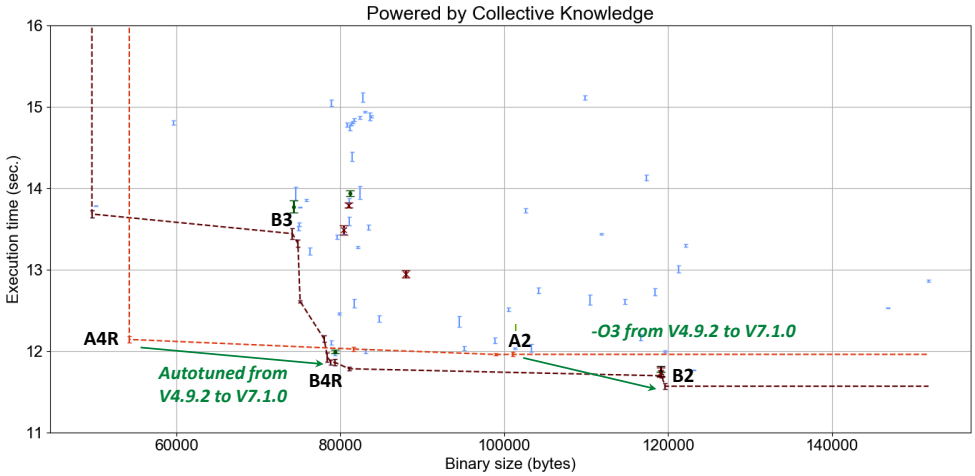
ID	Compiler	Time (sec.)	Size (bytes)	Flags
A1	GCC 4.9.2	41.3 ± 0.0	131140	
A2	GCC 4.9.2	12.2 ± 0.0	101448	-O3
A3	GCC 4.9.2	13.6 ± 0.0	78116	-Os
A4R	GCC 4.9.2	12.1 ± 0.1	54272	-O2 -flto -fno-tree-fre
A5	CLANG 3.8.1	38.5 ± 0.0	132080	
A6	CLANG 3.8.1	12.9 ± 0.1	90076	-O3

Figure 19: Results of GCC 4.9.2 random compiler flag autotuning of a zlib decode workload on RPi3 device using CK with a highlighted frontier (trading-off execution time and code size) and best found combinations of flags on this frontier.



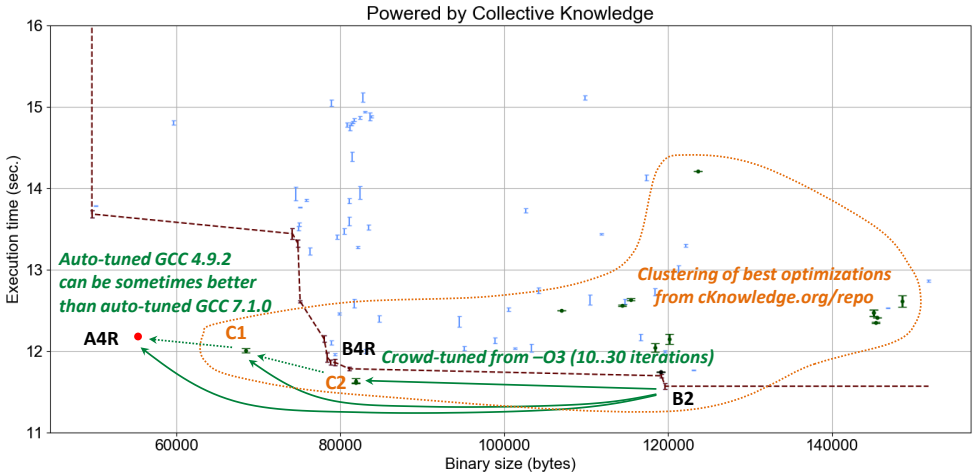
ID	Compiler	Time (sec.)	Size (bytes)	Flags
A2	GCC 4.9.2	12.2 ± 0.0	101448	-O3
A4R	GCC 4.9.2	12.1 ± 0.1	54272	-O2 -flto -fno-tree-fre
C1	GCC 4.9.2	12.2 ± 0.1	64184	-O3 -fno-inline -flto

Figure 20: Speeding up GCC 4.9.2 autotuning of a zlib decode workload on RPi3 device using 10..20 best performing combinations of compiler flags already found and shared by the community during crowd-tuning.



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A2	GCC 4.9.2	12.2 ± 0.0	101448	-O3
A4R	GCC 4.9.2	12.1 ± 0.1	54272	-O2 -flto -fno-tree-fre
B1	GCC 7.1.0	41.3 ± 0.0	128376	
B2	GCC 7.1.0	11.7 ± 0.1	119084	-O3
B3	GCC 7.1.0	13.7 ± 0.1	74280	-Os
B4R	GCC 7.1.0	11.9 ± 0.1	78700	-O2 -fno-early-inlining -fno-tree-fre

Figure 21: Results of GCC 7.1.0 random compiler flag autotuning of zlib decode on RPi3 device with a highlighted frontier (trading-off execution time and code size), best combinations of flags on this frontier, and comparison with the results from GCC 4.9.2.



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A4R	GCC 4.9.2	12.1 ± 0.1	54272	-O2 -flto -fno-tree-fre
B2	GCC 7.1.0	11.7 ± 0.1	119084	-O3
B4R	GCC 7.1.0	11.9 ± 0.1	78700	-O2 -fno-early-inlining -fno-tree-fre
C1	GCC 7.1.0	12.0 ± 0.0	68464	-O3 -fno-inline -flto
C2	GCC 7.1.0	11.6 ± 0.1	81880	-O3 -flto

Figure 22: Testing reactions of zlib decode to top most efficient GCC 7.1.0 optimizations shared by the community for RPi3 devices vs GCC 4.9.2.

We performed the same autotuning and crowd-tuning experiments for *zlib encode* workload with results shown in Figures 23, 24, 25, 26. The results show similar trend that *-O3* optimization level of both *GCC 4.7.2* and *GCC 7.1.0* perform well in terms of execution time, while there is the same degradation in code size when moving to a new compiler (since we monitor the whole zlib binary size for both decode and encode functions). Crowd-tuning also helped improve code size though optimizations **A4R**, **B4R** and **C1** are not the same as in case of *zlib decode*. The reason is that algorithms are different and need different optimizations to keep execution time intact while improving code size. Such result provides an extra motivation for function-level optimizations already available in GCC.

Besides *zlib*, we applied crowd-tuning with best found and shared optimizations to other RPi programs using *GCC 4.9.2* and *GCC 7.1.0*. Table 3 shows reactions of these optimizations with best trade-offs for execution time and code size. One may notice that though *GCC 7.1.0 -O3* level improves execution time of most of the programs apart from a few exceptions, it also considerably degrades code size in comparison *GCC 4.9.2 -O3* level. These results also confirm that neither *-O3* nor *-Os* on both *GCC 4.9.2* and *GCC 7.1.0* achieves best trade-offs for execution time and code size thus motivating again our collaborative and continuous optimization approach.

Indeed, a dozen of shared most efficient optimizations at cKnowledge.org/repo is enough to either improve execution time of above programs by up to 1.5x or code size by up to 1.8x or even improve both size and speed at the same time. It also helps end-users find best optimization no matter which compiler, environment and hardware are used.

We can also notice that 11 workloads (computational species) share *-O3 -fno-inline -flto* combination of flags to achieve best trade-off between execution time and code size. This result supports our original research to use workload features, hardware properties, crowd-tuning and machine learning to predict such optimizations [58, 61, 64]. However, in contrast with past work, we are now able to gradually collect a large, realistic (i.e. not randomly synthesized) set of diverse workloads with the help of the community to make machine learning statistically meaningful.

All scripts to reproduce experiments from this section are available in the following CK entries:

```
$ ck find script:rpi3-zlib-decode*
$ ck find script:rpi3-zlib-encode*
$ ck find script:rpi3-all-autotune
```

7 Crowd-fuzzing compilers

When distributing compiler autotuning and learning across diverse environments, compilers and devices [58, 64] we have noticed that about 10..15% of randomly generated combinations of flags can crash a compiler or produce wrong code with segmentation faults or incorrect output. Indeed our approach stresses various unexpected combinations of compiler optimizations across diverse and possibly untested platforms and workloads thus helping automatically detect software and hardware bugs. It complements well-known fuzzing techniques for automatic software testing [54, 109, 115].

Our CK-based customizable autotuning workflow can assist in creating, learning and improving such collaborative fuzzers which can distribute testing across diverse platforms and workloads provided by volunteers while sharing and reproducing bugs. We just need to retarget our autotuning workflow to search for bugs instead of or together with improvements in performance, energy, size and other characteristics.

We prepared an example scenario *experiment.tune.compiler.flags.gcc.fuzz* to randomly generate compiler flags for any GCC and record only cases with failed program pipeline. One can use it in a same way as any CK autotuning while selecting above scenario as following:

```
$ ck autotune program:cbench-automotive-susan
--iterations=150 --repetitions=3
--scenario=experiment.tune.compiler.flags.gcc.fuzz
--cmd_key=corners
--record_uoa=tmp-susan-corners-gcc7-150-rnd-fuzz
```

It is then possible to view all results with unexpected behavior in a web browser and reproduce individual cases on a local or different machine as following:

```
$ ck browser
experiment:tmp-susan-corners-gcc7-150-rnd-fuzz
$ ck replay
experiment:tmp-susan-corners-gcc7-150-rnd-fuzz
```

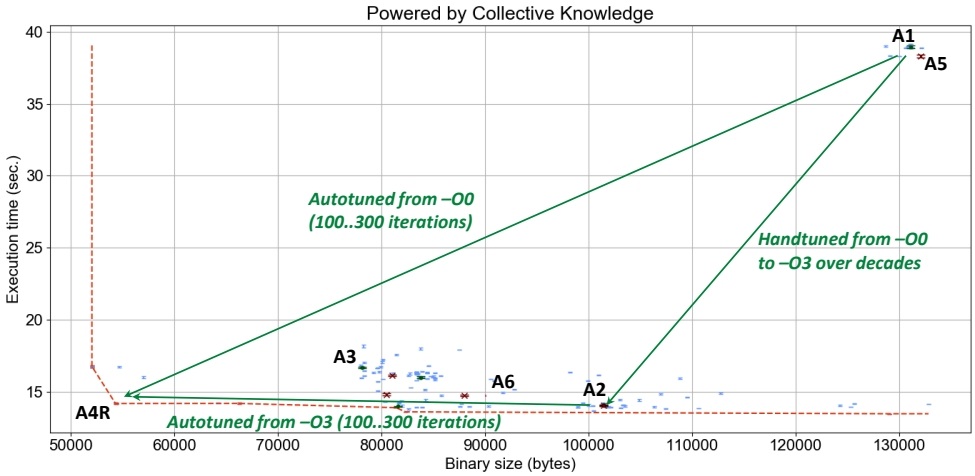
We performed the same auto-fuzzing experiments for *susan corners* program with both *GCC 4.9.2* and *GCC 7.1.0* as in Section 4. These results are available in the following CK entries:

```
$ ck search experiment:rpi3-*fuzz*
```

It is also possible to browse them online.

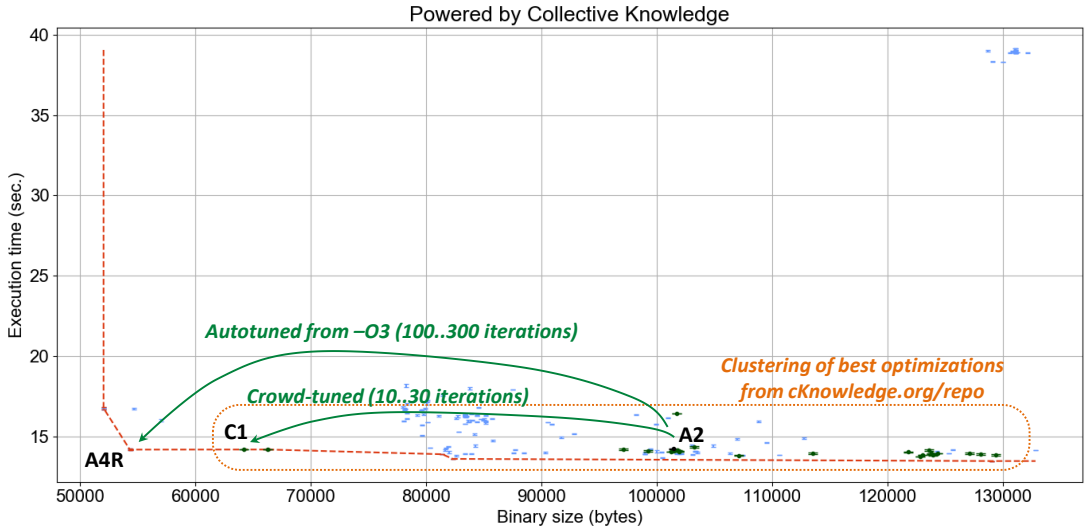
Figure 27 shows a simple example of reproducing a GCC bug using CK together with the original random combination of flags and the reduced one. GCC flag *-fcheck-data-deps* compares several passes for dependency analysis and report a bug in case of discrepancy. Such discrepancy was automatically found when autotuning *susan corners* using *GCC 4.9.2* on RPi3.

Since CK automatically adapts to a user environment, it is also possible to reproduce the same bug using a different compiler version. Compiling the same program



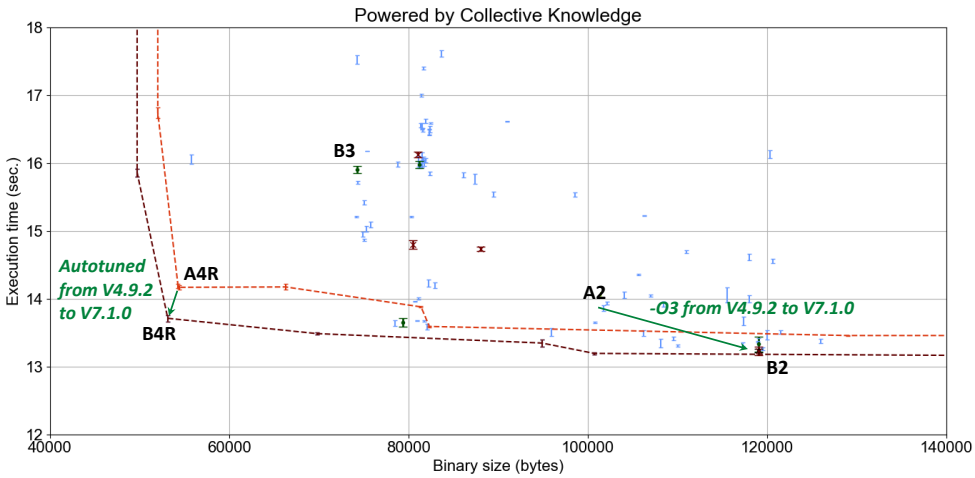
ID	Compiler	Time (sec.)	Size (bytes)	Flags
A1	GCC 4.9.2	39.0 ± 0.1	131140	
A2	GCC 4.9.2	14.0 ± 0.1	101448	-O3
A3	GCC 4.9.2	16.7 ± 0.1	78116	-Os
A4R	GCC 4.9.2	14.2 ± 0.1	54284	-O2 -flto
A5	CLANG 3.8.1	38.2 ± 0.1	132080	
A6	CLANG 3.8.1	14.7 ± 0.1	90076	-O3

Figure 23: Results of GCC 4.9.2 random compiler flag autotuning of a zlib encode workload on RPi3 device using CK with a highlighted frontier (trading-off execution time and code size) and best found combinations of flags on this frontier.



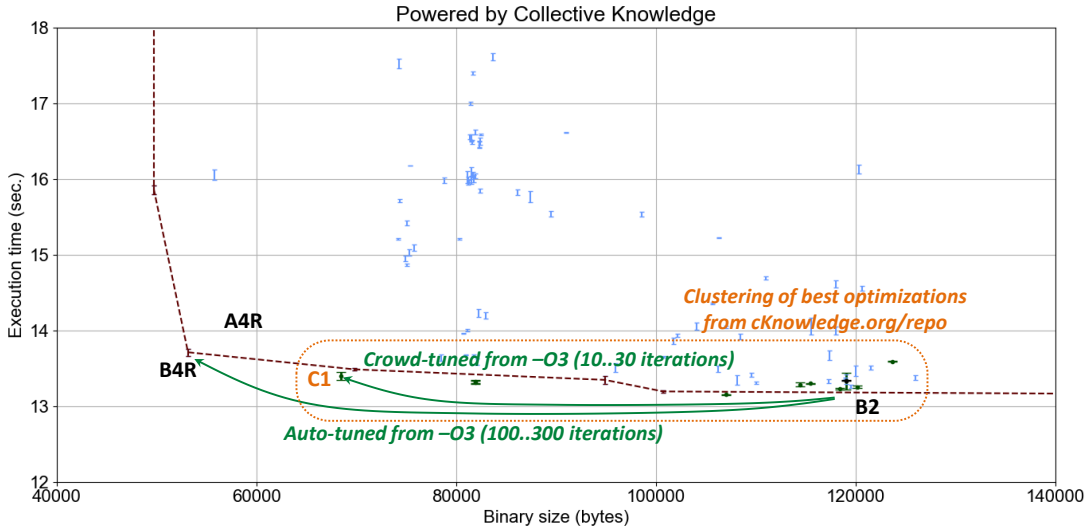
ID	Compiler	Time (sec.)	Size (bytes)	Flags
A2	GCC 4.9.2	14.0 ± 0.1	101448	-O3
A4R	GCC 4.9.2	14.2 ± 0.1	54284	-O2 -flto
C1	GCC 4.9.2	14.2 ± 0.0	64184	-O3 -fno-inline -flto

Figure 24: Accelerating GCC 4.9.2 autotuning of a zlib encode workload on RPi3 device using 10..20 best performing combinations of compiler flags already found and shared by the community during collaborative optimization.



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A2	GCC 4.9.2	14.0 ± 0.1	101448	-O3
A4R	GCC 4.9.2	14.2 ± 0.1	54284	-O2 -flto
B1	GCC 7.1.0	38.8 ± 0.0	128376	
B2	GCC 7.1.0	13.2 ± 0.1	119084	-O3
B3	GCC 7.1.0	15.9 ± 0.1	74280	-Os
B4R	GCC 7.1.0	13.7 ± 0.0	52424	-O2 -fgcse-after-reload -flto -fschedule-fusion -fno-ssa-phiopt -fno-tree-fre

Figure 25: Results of GCC 7.1.0 random compiler flag autotuning of zlib encode on RPi3 device with a highlighted frontier (trading-off execution time and code size), best combinations of flags on this frontier, and comparison with the results from GCC 4.9.2.



ID	Compiler	Time (sec.)	Size (bytes)	Flags
A4R	GCC 4.9.2	14.2 ± 0.1	54284	-O2 -flto
B2	GCC 7.1.0	13.2 ± 0.1	119084	-O3
B4R	GCC 7.1.0	13.7 ± 0.0	52424	-O2 -fgcse-after-reload -flto -fschedule-fusion -fno-ssa-phiopt -fno-tree-fre
C1	GCC 4.9.2	13.3 ± 0.1	68464	-O3 -fno-inline -flto

Figure 26: Analyzing reactions of zlib encode to top most efficient GCC 7.1.0 optimizations shared by the community for RPi3 devices vs GCC 4.9.2.

```
$ ck replay experiment:f14372bd49376cd5 --point=05df924a4f02adc2
```

Compiler flags before reduction: -O3 -fno-function-sections -fno-gcse-lm -fcheck-data-deps -fno-gcse-sm -fno-ivopts -fzero-initialized-in-bss -fomit-frame-pointer -frename-registers -frounding-math -fno-tree-coalesce-vars -fno-tree-forwprop -funsafe-math-optimizations

GCC 4.9.2 on RPi3 crashes:

```
(Number of distance vectors differ: Banerjee has 1, Omega has 0.
Banerjee dist vectors:
 4
Omega dist vectors:
data dependence relation:
(Data Dep:
#(Data Ref:
#  bb: 5
#  stmt: _59 = *p_57;
#  ref: *p_57;
#  base_object: *in_22(D) + ((sizetype) x_size_6(D) + (sizetype)
pretmp_1219);
#  Access function 0: {1B(OVF), +, 1}_2
#)
#(Data Ref:
#  bb: 5
#  stmt: _90 = *p_82;
#  ref: *p_82;
#  base_object: *in_22(D) + ((sizetype) x_size_6(D) + (sizetype)
pretmp_1219);
#  Access function 0: {5B(OVF), +, 1}_2
#)
 access_fn_A: {1B(OVF), +, 1}_2
 access_fn_B: {5B(OVF), +, 1}_2

(subscript
 iterations_that_access_an_element_twice_in_A: [4 + 1 * x_1]
 last_conflict: 2147483637
 iterations_that_access_an_element_twice_in_B: [0 + 1 * x_1]
 last_conflict: 2147483637
 (Subscript distance: 4 ))
 inner loop index: 0
 loop nest: (2 )
)
)
../susan.c: In function 'susan_principle':
../susan.c:495:6: internal compiler error: in
compute_affine_dependence, at tree-data-ref.c:4253
 void susan_principle(uchar* in, int* r, uchar* bp,
      ^
Please submit a full bug report,
with preprocessed source if appropriate.
See <file:///usr/share/doc/gcc-4.9/README.Bugs> for instructions.
Preprocessed source stored into /tmp/ccW0PUrM.out file, please attach
this to your bugreport.
```

```
$ ck replay experiment:f14372bd49376cd5 --point=05df924a4f02adc2 –reduce_bug
```

Compiler flags after reduction: -O3 -fcheck-data-deps

Figure 27: Basic example of reproducing and reducing GCC bugs after random compiler flag autotuning.

Workload	Compiler	Time improvement over -O3 (-O3 time in brackets)	Binary size improvement over -O3 (-O3 size in brackets)	Flags
7z encode	GCC 4.9.2	1.02 (5.5 ± 0.1)	1.52 (859728)	-O3 -fno-inline -flto
7z encode	GCC 7.1.0	no (6.0 ± 1.0)	no (887464)	-O3
ccrypt encrypt	GCC 4.9.2	no (7.0 ± 2.0)	no (61772)	-O3
ccrypt encrypt	GCC 7.1.0	1.16 (7.6 ± 0.1)	1.00 (59996)	-O3 -fno-auto-inc-dec -fguess-branch-probability -fipa-pure-const -freorder-blocks -fselective-scheduling2 -ftree-ccp -fno-tree-pre -ftree-tail-merge
gzip decode	GCC 4.9.2	1.04 (4.2 ± 0.0)	1.12 (85956)	-O3 -fno-inline -flto
gzip decode	GCC 7.1.0	1.04 (4.2 ± 0.0)	1.18 (90568)	-O3 -fno-inline -flto
gzip decode	GCC 7.1.0	1.08 (4.2 ± 0.0)	0.81 (90568)	-O3 -fno-cprop-registers -flto -funroll-all-loops
gzip encode	GCC 4.9.2	0.98 (12.3 ± 0.1)	1.10 (85956)	-O3 -fno-omit-frame-pointer -fno-tree-loop-optimize
gzip encode	GCC 7.1.0	1.01 (12.3 ± 0.8)	1.18 (90568)	-O3 -fno-inline -flto
minigzip decode	GCC 4.9.2	1.24 (10.0 ± 4.0)	1.60 (101432)	-O3 -fno-inline -flto
minigzip decode	GCC 4.9.2	1.32 (10.0 ± 4.0)	1.00 (101432)	-O3 -fselective-scheduling2 -fno-tree-pre
minigzip decode	GCC 7.1.0	1.14 (8.0 ± 3.0)	1.76 (119088)	-O3 -fno-inline -flto
minigzip encode	GCC 4.9.2	0.89 (9.9 ± 0.0)	1.60 (101432)	-O3 -fno-inline -flto
minigzip encode	GCC 7.1.0	1.00 (9.6 ± 0.0)	1.76 (119088)	-O3 -fno-inline -flto
rhash sha3	GCC 4.9.2	1.00 (4.8 ± 0.0)	1.12 (14848)	-O3 -flto
rhash sha3	GCC 7.1.0	1.35 (5.2 ± 0.0)	1.30 (16396)	-O3 -fno-inline -flto
rhash sha3	GCC 7.1.0	1.48 (5.2 ± 0.0)	1.07 (16396)	-O3 -fno-schedule-insns -ftracer
sha512sum sha512	GCC 4.9.2	1.12 (7.8 ± 0.0)	1.06 (125372)	-O3 -fno-schedule-insns -fselective-scheduling2
sha512sum sha512	GCC 7.1.0	1.22 (7.3 ± 0.0)	1.07 (121180)	-O3 -fno-predictive-commoning -fno-schedule-insns -funroll-loops
unrar	GCC 4.9.2	0.97 (18.0 ± 4.0)	1.38 (326572)	-O3 -fno-inline -flto
unrar	GCC 4.9.2	1.13 (18.0 ± 4.0)	0.80 (326572)	-O3 -fno-section-anchors -fselective-scheduling2 -fno-tree-forwprop -funroll-all-loops
unrar	GCC 7.1.0	0.96 (18.0 ± 6.0)	1.38 (326572)	-O3 -fno-inline -flto
unrar	GCC 7.1.0	1.07 (18.0 ± 6.0)	0.78 (326572)	-O3 -fno-tree-ter -funroll-all-loops

Table 3: Best found improvements (degradations) in execution time and binary size for several important RPi3 programs as reactions to top most efficient shared optimizations for GCC 4.9.2 and GCC 7.1.0.

with the same combination of flags on the same platform using *GCC 7.1.0* showed that this bug has been fixed in the latest compiler.

We hope that our extensible and portable benchmarking workflow will help students and engineers prototype and crowdsource different types of fuzzers. It may also also assist even existing projects [27, 26] to crowdsource fuzzing across diverse platforms and workloads. For example, we collaborate with colleagues from Imperial College London to develop CK-based, continuous and collaborative OpenGL and OpenCL compiler fuzzers [82, 79, 23] while aggregating results from users in public or private repositories (link to public OpenCL fuzzing results across diverse desktop and mobile platforms).

All scripts to reproduce experiments from this section are available in the following CK entry:

```
$ ck find script:rpi3-susan-fuzz-bugs
```

8 Unifying and crowdsourcing machine learning

Having all optimization statistics continuously aggregated in a repository in a common format with JSON meta description makes it relatively straightforward to apply various machine learning and predictive analytics techniques including decision trees, nearest neighbor classifiers, support vector machines (SVM) and deep learning [45, 102]. These techniques can help automate detection of regularities and consistent patterns in program behavior, build models, and predict efficient optimizations rather than continuously re-optimizing each new program as we previously demonstrated in the MILEPOST project [61, 47]. Furthermore, we can now teach students how to collaboratively model the behavior of all computer systems, speed up optimization space exploration, and improve predictions of the most efficient software and hardware optimizations based on various program, data set, platform and run-time features [65, 64].

To demonstrate our approach, we converted all our past research artifacts on machine learning based optimization and SW/HW co-design to CK modules. We then assembled them to a universal Collective Knowledge workflow shown in Figure 28. If you do not know about machine learning based compiler optimizations, we suggest you to start from our MILEPOST GCC paper [61] to make yourself familiar with terminology and methodology for machine learning training and prediction used further. Next, we will briefly demonstrate the use of this customizable workflow to continuously classify shared workloads presented in this report in terms of the most efficient compiler optimizations while using MILEPOST models and features.

First, we query the public CK repository [8] to collect all optimization statistics together with all associated objects (workloads, data sets, platforms) for a given optimization scenario. In our compiler flag optimization scenario, we retrieve all most efficient compiler flags combinations found and shared by the community when crowd-tuning GCC 4.9.2 on RPi3 device (Figure 17).

Note that our CK crowd-tuning workflow also continuously apply such optimization to all shared workloads. This allows us to analyze "reaction" of any given workload to all most efficient optimizations. We can then group together those workloads which exhibit similar reactions.

The top graph in Figure 29 shows reactions of all workloads to the most efficient optimizations as a ratio of the default execution time (-O3) to the execution time of applied optimization. It confirms yet again ([64]) that there is no single "winning" combination of optimizations and they can either considerably improve or degrade execution time on different workloads. It also confirms that it is indeed possible to group together multiple workloads which share the most efficient combination of compiler flags, i.e. which achieve the highest speedup for a common optimization as shown in the bottom graph in Figure 29. Figure 30 shows similar trends for GCC 7.1.0 on the same RPi3 device even though the overall number of the most efficient combinations of compiler flags is smaller than for GCC 4.9.2 likely due to considerably improved internal optimization heuristics over past years (see Figure 18).

Having such groups of labeled objects (where labels are the most efficient optimizations and objects are workloads) allows us to use standard machine learning classification methodology. One must find such a set of objects' features and a model which maximizes correct labeling of previously unseen objects, or in our cases can correctly predict the most efficient software optimization and hardware design for a given workload. As example, we extracted 56 so-called MILEPOST features described in [61] (static program properties extracted from GCC's intermediate representation) from all shared programs, stored them in *program.static.features*, and applied simple nearest neighbor classifier to above data. We then evaluated the quality of such model (ability to predict) using prediction accuracy during standard leave-one-out cross-validation technique: for each workload we remove it from the training set, build a model, validate predictions, sum up all correct predictions and divide by the total number of workloads.

Table 4 shows this prediction accuracy of our MILEPOST model for compiler flags from GCC 4.9.2 and GCC 7.1.0 across all shared workloads on RPi3 device. One may notice that it is nearly twice lower than in the original MILEPOST paper [61]. As we explain in [64], in the MILEPOST project we could only use a dozen of similar workloads and just a few

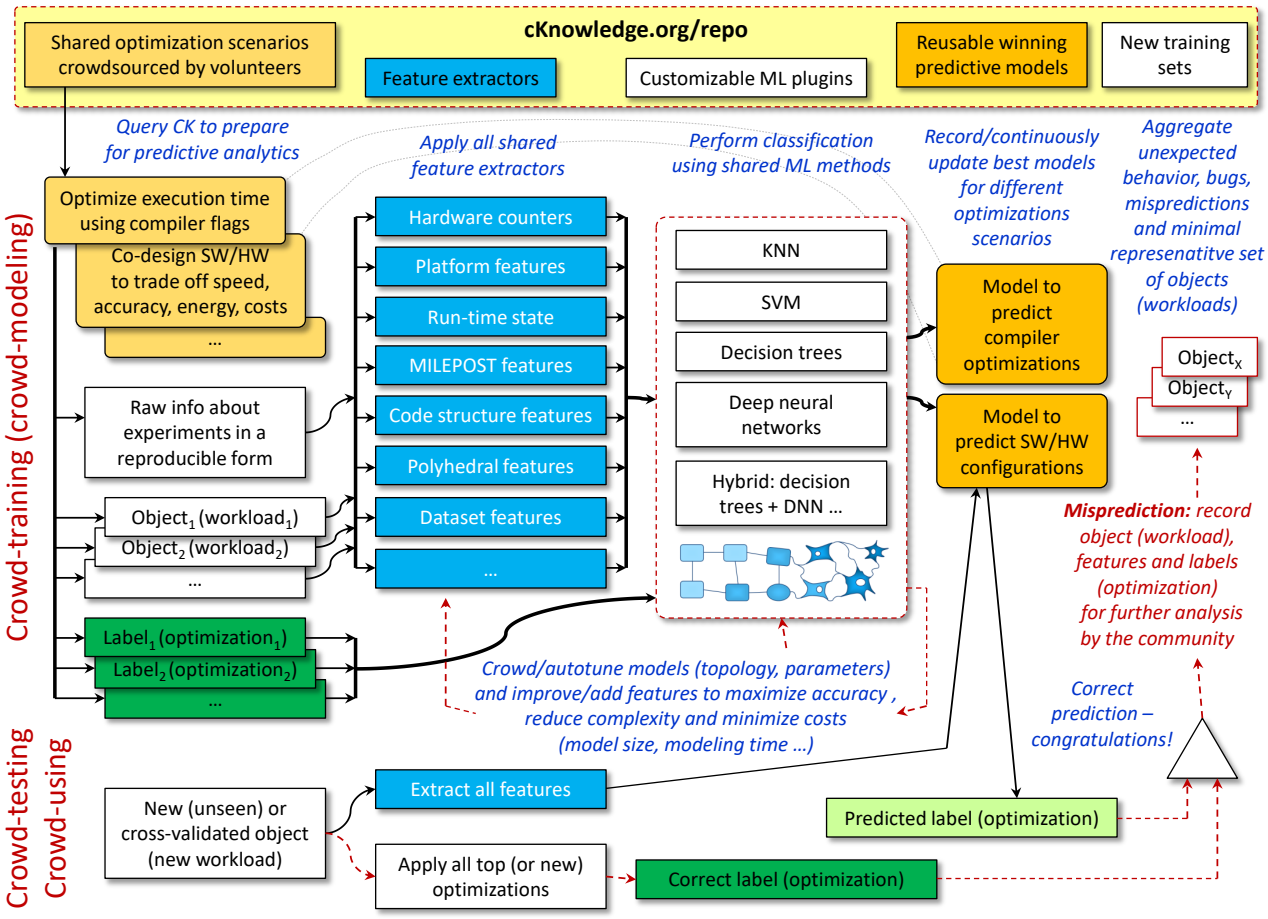


Figure 28: Universal and high-level Collective Knowledge workflow to connect various communities for collaborative, continuous and semi-automatic learning of multi-objective optimizations using shared machine learning modules (plugins) with the unified CK API.

Model	**Features**	**Accuracy (GCC 4.9.2)**	**Accuracy (GCC 7.1.0)**
milepost nn	ft1 .. ft56	0.37	0.30

Table 4: Accuracy of the nearest neighbor classifier with MILEPOST features to predict the most efficient combinations of compiler flags for GCC 4.9.2 and GCC 7.1.0 flags on RPi3 device.

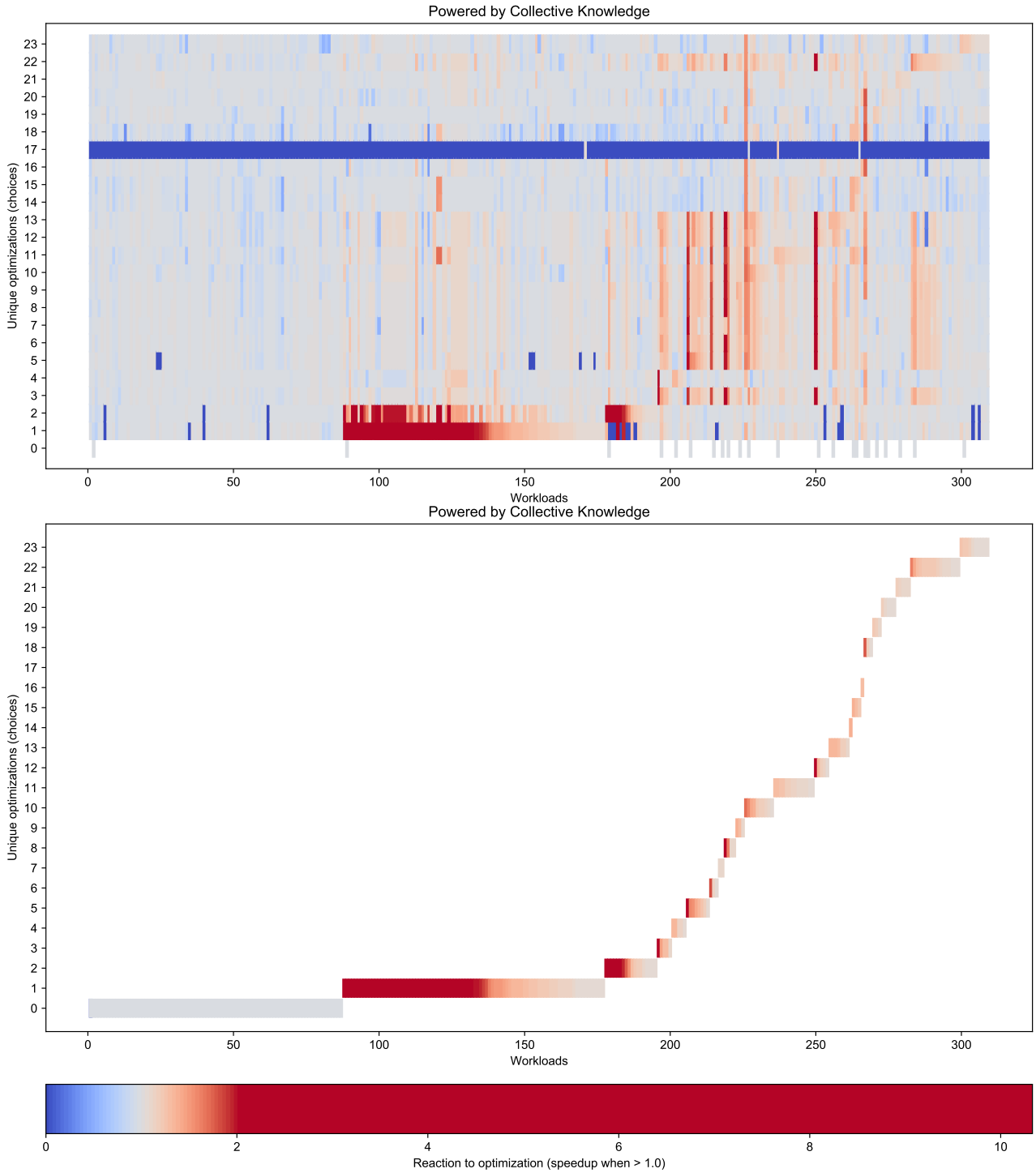


Figure 29: Top graph: reactions of all workloads to all top performing combinations of optimizations for GCC 4.9.2 on RPi3 device (speedups if value is more than 1.0). Bottom graph: groups of workloads achieving the highest speedup for a given unique combination of optimizations.

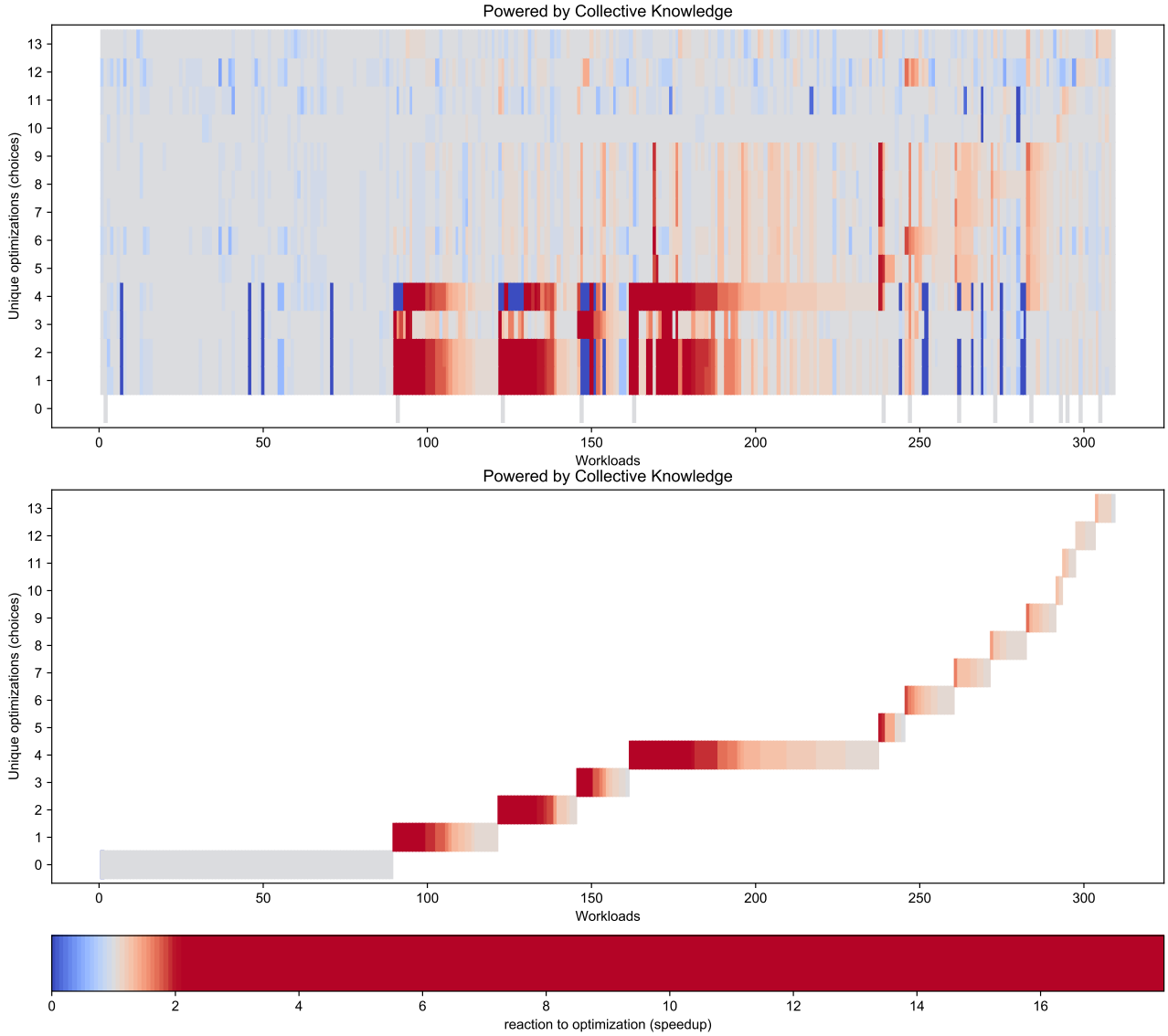


Figure 30: Top graph: reactions of all workloads to all top performing combinations of optimizations for GCC 7.1.0 on RPi3 device (speedup if value is more than 1.0). Bottom graph: groups of workloads achieving the highest speedup for a given unique combination of optimizations.

most efficient optimizations to be able to perform all necessary experiments within a reasonable amount of time (6 months). After brining the community on hoard, we could now use a much larger collective training set with more than 300 shared, diverse and non-synthesized workloads while analyzing much more optimizations by crowdsourcing autotuning. This helps obtain a more realistic limit of the MILEPOST predictor.

Though relatively low, this number can now become a reference point to be further improved by the community. It is similar in spirit to the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [101] which reduced image classification error rate from 25% in 2011 to just a few percent with the help of the community. Furthermore, we can also keep just a few representative workloads for each representative group as well as misclassified ones in a public repository thus producing a minimized, realistic and representative training set for systems researchers.

We shared all demo scripts which we used to generate data and graphs in this section in the following CK entry (however they are not yet user-friendly and we will continue improving documentation and standardizing APIs of reusable CK modules with the help of the community):

```
$ ck find script:rpi3-crowdmodel
```

9 Improving and autotuning models and features

There are many publications demonstrating interesting machine learning algorithms, features and models to predict efficient program optimizations and hardware designs [91, 108, 87, 107, 116, 35, 47, 53, 75, 103, 97, 81, 51, 41]. Though all these techniques can be potentially useful, the lack of common interfaces and meta information for artifacts and experimental workflows makes it extremely challenging to compare, reuse and build upon them particularly in industrial projects with tough deadlines.

Even artifact evaluation which we introduced at systems conferences [21] to partially solve these issues is not yet enough because our community does not have a common, portable and customizable workflow framework. Bridging this gap between machine learning and systems research served as an additional motivation to develop Collective Knowledge workflow framework. Our idea is to help colleagues and students share various workloads, data sets, machine learning algorithms, models and feature extractors as plugins (CK modules) with a common API and meta description. Plugged to a common machine learning workflow such modules can then be applied in parallel to continuously compete for the most accurate predictions for a given optimization scenario. Furthermore, the community can continue improving and autotuning models, analyzing various combination of features, experimenting with hierarchical models, and pruning models to reduce their complexity across shared data sets to trade off prediction accuracy, speed, size and the ease of interpretation.

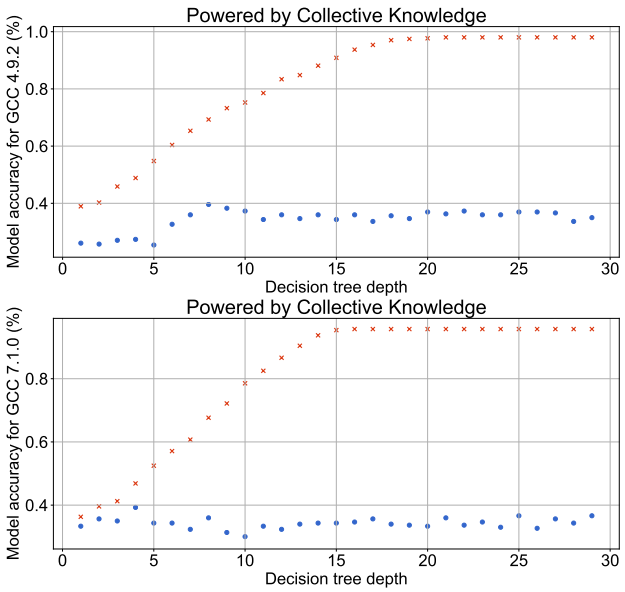


Figure 31: Accuracy of an automatically generated decision tree to predict compiler flags (GCC 4.9.2 on the top graph and GCC 7.1.0 on the bottom graph) on RPi3 when autotuning the tree depth. Round blue dots show prediction accuracy with cross-validation while blue crosses show prediction accuracy without cross-validation.

For a proof-of-concept of such collaborative learning approach, we shared a number of customizable CK modules (see *ck search module:*model**) for several popular classifiers including nearest neighbor, decision trees and deep learning. These modules serve as wrappers with a common CK API for TensorFlow, scikit-learn, R and other machine learning frameworks. We also shared several feature extractors (see *ck search module:*features**) assembling the following groups of program features which may influence predictions:

- **ft1 .. ft56** - original MILEPOST features (see [61]);
- **ft57 .. ft65** - additional features designed and shared by our colleague, Dr. Jeremy Singer [12];
- **ft66 .. ft121** - original MILEPOST features normalized by the total number of instructions (ft24);

We then attempted to autotune various parameters of machine learning algorithms exposed via CK API. Figure 31 shows an example of autotuning the depth of a decision tree (available as customizable CK plugin) with all shared groups of features and its impact on prediction

accuracy of compiler flags using MILEPOST features from the previous section for GCC 4.9.2 and GCC 7.1.0 on RPi3. Blue round dots obtained using leave-one-out validation suggest that decision trees of depth 8 and 4 are enough to achieve maximum prediction accuracy of 0.4% for GCC 4.9.2 and GCC 7.1.0 respectively. Model autotuning thus helped improve prediction accuracy in comparison with original nearest neighbor classifier from the MILEPOST project.

Figure 32 shows a few examples of such automatically generated decision trees with different depths for GCC 7.1.0 using CK. Such trees are easy to interpret and can therefore help compiler and hardware developers quickly understand the most influential features and analyze relationships between different features and the most efficient optimizations. For example, above results suggest that the number of binary integer operations (ft22) and the number of distinct operators (ft59) can help predict optimizations which can considerably improve execution time of a given method over -O3.

Turning off cross-validation can also help developers understand how well models can perform on all available workloads (in-sample data) (red dots on Figure 31). In our case of GCC 7.1.0, the decision tree of depth 15 shown in Figure 32) is enough to capture all compiler optimizations for 300 available workloads.

To complete our demonstration of CK concepts for collaborative machine learning and optimization, we also evaluated a deep learning based classifier from TensorFlow [33] (see *ck help module:model.tf*) with 4 random configurations of hidden layers ([10,20,10], [21,13,21], [11,30,18,20,13], [17]) and training steps (300..3000). We also evaluated the nearest neighbor classifier used in the MILEPOST project but with different groups of features and aggregated all results in Table 5. Finally, we automatically reduced the complexity of the nearest neighbor classifier (1) by iteratively removing those features one by one which do not degrade prediction accuracy and (2) by iteratively adding features one by one to maximize prediction accuracy. It is interesting to note that our nearest neighbor classifier achieves a slightly better prediction accuracy with a reduced feature set than with a full set of features showing inequality of MILEPOST features and overfitting.

As expected, deep learning classification achieves a better prediction accuracy of 0.68% and 0.45% for GCC 4.9.2 and GCC 7.1.0 respectively for RPi3 among currently shared models, features, workloads and optimizations. However, since deep learning models are so much more computationally intensive, resource hungry and difficult to interpret than decision trees, one must carefully balance accuracy vs speed vs size. That is why we suggest to use hierarchical models where high-level and coarse-grain program behavior is quickly captured using decision trees, while all fine-grain behavior is captured by deep learning and similar techniques. Another possible use of deep learning can be in automatically capturing influential features from the source code, data sets and hardware.

All scripts to generate above experiments (require further documentation) are available in the following CK entry:

```
$ ck find script:rpi3-crowdmodel
```

10 Enabling input-aware optimization

Current prediction accuracy which we achieved for the most efficient compiler flags is still disappointing: around 0.45% for GCC 7.1.0. We explained this in more detail in [64, 65] by missing features particularly available at run-time from data sets and hardware. Having a customizable experimental workflow with pluggable artifacts makes it relatively straightforward to analyze reactions of a given program to the most efficient optimization across multiple data sets and search for missing features.

First, we converted 474 different data sets from the MiDataSet suite [59] as pluggable CK artifacts and shared them as a zip archive (800MB). It is possible to download it from the Google Drive from `https://drive.google.com/open?id=0B-wXENVfIO82OUpZdWIzckhlRk0` (we plan to move it to a permanent repository in the future) and then install via CK as following:

```
$ ck add repo --zip=ckr-ctuning-datasets.zip
--quiet
$ ck ls dataset --all
$ ck search dataset --tags=image,jpeg
```

All these data sets will be immediately visible to all related programs via the CK autotuning workflow. For example, if we now run *susan corners* program, CK will prompt user a choice of 20 related images from the above data sets:

```
$ ck compile program:cbench-automotive-susan
--speed
$ ck run program:cbench-automotive-susan
```

Next, we can apply all most efficient compiler optimizations to all data sets for a given programs. Figure 33 shows such reactions (ratio of an execution time with a given optimization to an execution time with the default -O3 compiler optimization) of a jpeg decoder across 20 different jpeg images from the above MiDataSet on RPi3.

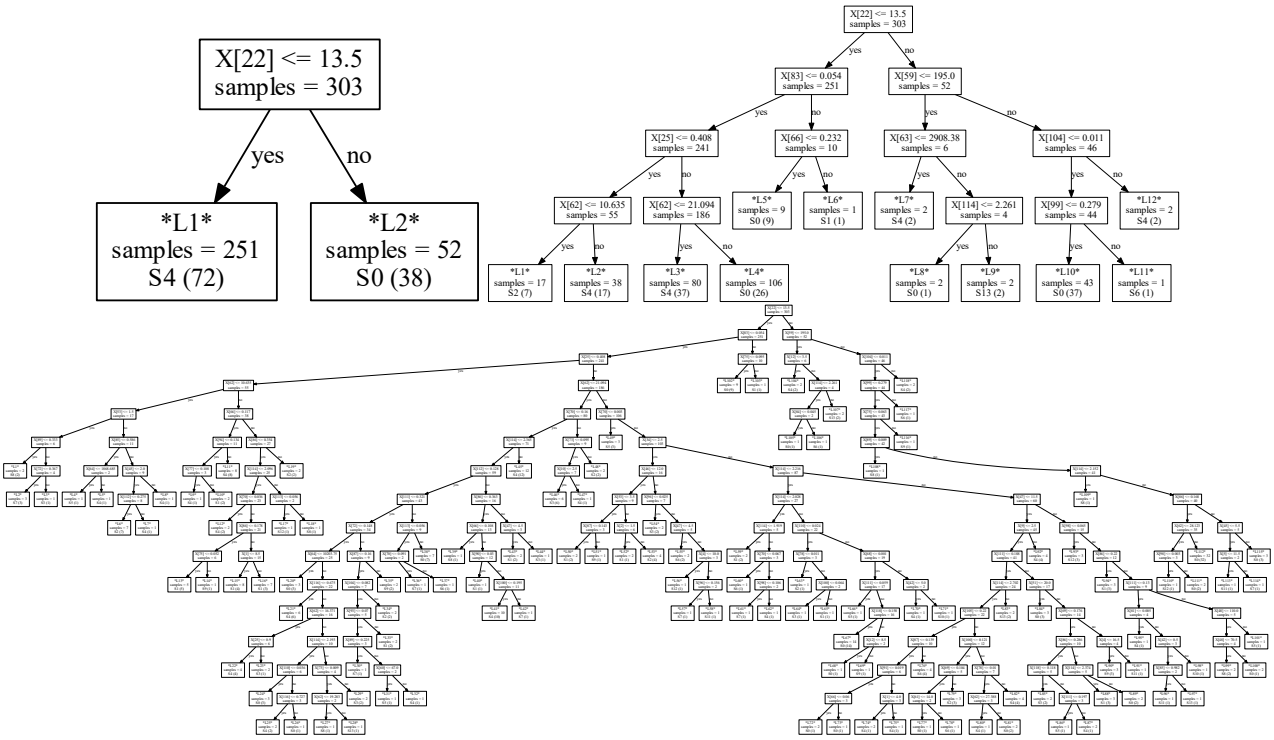


Figure 32: Example of automatically generated decision trees of depth 1 and 4 with leave-one-out cross-validation, and 15 without cross-validation to predict GCC 7.1.0 compiler optimizations using CK modules.

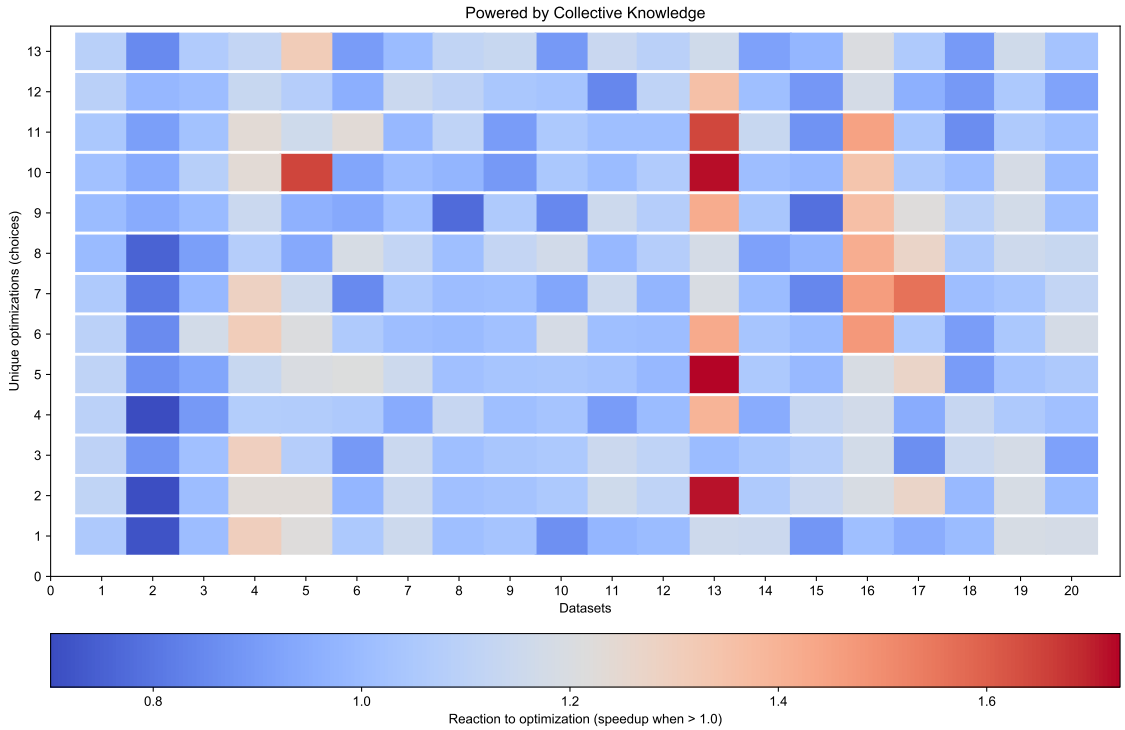


Figure 33: Reactions of a jpeg decoder across 20 distinct data sets (jpeg images) to all top performing combinations of compiler optimizations for GCC 7.1.0 on RPi3 device (speedups if value is more than 1.0).

Model	Features	Accuracy (GCC 4.9.2)	Accuracy (GCC 7.1.0)
decision trees with cross validation; depth 1	ft1 .. ft65	0.26	0.33
decision trees with cross validation; depth 2	ft1 .. ft65	0.26	0.36
decision trees with cross validation; depth 4	ft1 .. ft65	0.27	0.39
decision trees with cross validation; depth 8	ft1 .. ft65	0.40	0.36
decision trees with cross validation; depth 16	ft1 .. ft65	0.36	0.35
decision trees with cross validation; depth 20	ft1 .. ft65	0.37	0.33
decision trees with cross validation; depth 25	ft1 .. ft65	0.37	0.37
decision trees with cross validation; depth 29	ft1 .. ft65	0.35	0.37
decision trees without cross validation; depth 1	ft1 .. ft65	0.39	0.36
decision trees without cross validation; depth 2	ft1 .. ft65	0.40	0.40
decision trees without cross validation; depth 4	ft1 .. ft65	0.49	0.47
decision trees without cross validation; depth 8	ft1 .. ft65	0.69	0.68
decision trees without cross validation; depth 16	ft1 .. ft65	0.94	0.96
decision trees without cross validation; depth 20	ft1 .. ft65	0.98	0.96
decision trees without cross validation; depth 25	ft1 .. ft65	0.98	0.96
decision trees without cross validation; depth 29	ft1 .. ft65	0.98	0.96
dnn tf with cross validation; iteration 1	ft1 .. ft65	0.68	0.30
dnn tf with cross validation; iteration 2	ft1 .. ft65	0.64	0.33
dnn tf with cross validation; iteration 3	ft1 .. ft65	0.61	0.45
dnn tf with cross validation; iteration 4	ft1 .. ft65	0.64	0.44
dnn tf without cross validation; iteration 1	ft1 .. ft65	0.72	0.29
dnn tf without cross validation; iteration 2	ft1 .. ft65	0.72	0.47
dnn tf without cross validation; iteration 3	ft1 .. ft65	0.72	0.48
dnn tf without cross validation; iteration 4	ft1 .. ft65	0.68	0.62
milepost nn	ft1 .. ft121	0.30	0.30
milepost nn	ft1 .. ft56	0.37	0.30
milepost nn	ft1 .. ft65	0.30	0.30
milepost nn	ft57 .. ft121	0.30	0.30
milepost nn	ft57 .. ft65	0.30	0.30
milepost nn	ft66 .. ft121	0.36	0.32
milepost nn	ft1 .. ft121 (normalized)	0.37	0.37
milepost nn	ft1 .. ft56 (normalized)	0.37	0.33
milepost nn	ft1 .. ft65 (normalized)	0.39	0.32
milepost nn	ft57 .. ft121 (normalized)	0.37	0.39
milepost nn	ft57 .. ft65 (normalized)	0.37	0.35
milepost nn	ft66 .. ft121 (normalized)	0.38	0.38
milepost nn (reduce complexity1)	ft1 .. ft121 (normalized)	0.45	0.44
milepost nn (reduce complexity2)	ft1 .. ft121 (normalized)	0.45	0.40

Table 5: Prediction accuracy when autotuning or reducing complexity of decision tree, nearest neighbor and deep learning classifiers across different groups of program features.

One can observe that the same combination of compiler flags can both considerably improve or degrade execution time for the same program but across different data sets. For example, data sets 4,5,13,16 and 17 can benefit from the most efficient combination of compiler flags found by the community with speedups ranging from 1.2 to 1.7. On the other hand, it's better to run all other data sets with the default -O3 optimization level.

Unfortunately, finding data set and other features which could easily differentiate above optimizations is often very challenging. Even deep learning may not help if a feature is not yet exposed. We explain this issue in [64] when optimizing real B&W filter kernel - we managed to improve predictions by exposing a "time of the day" feature only via human intervention. However, yet again, the CK concept is to bring the interdisciplinary community on board to share such cases in a reproducible way and then collaboratively find various features to improve predictions.

Another aspect which can influence the quality of predictive models, is that the same combinations of compiler flags are too coarse-grain and can make different internal optimization decisions for different programs. Therefore, we need to have an access to fine-grain optimizations (inlining, tiling, unrolling, vectorization, prefetching, etc) and related features to continue improving our models. However, this follows our top-down optimization and modeling methodology which we implemented in the Collective Knowledge framework. We want first to analyze, optimize and model coarse-grain behavior of shared workloads together with the community and students while gradually adding more workloads, data sets, models and platforms. Only when we reached the limit of prediction accuracy, we start gradually exposing finer-grain optimizations and features via extensible CK JSON interface while avoiding explosion in design and optimization spaces (see details in [65] for our previous version of the workflow framework, Collective Mind). This is much in spirit of how physicists moved from Newton's three coarse-grain laws of motion to fine-grain quantum mechanics.

To demonstrate this approach, we shared a simple skeletonized matrix multiply kernel from [57] in the CK format with blocking (tiling) parameter and data set feature (square matrix size) exposed via CK API:

```
$ ck compile program:shared-matmul-c2
--flags="-DUSE_BLOCKED_MATMUL=YES
$ ck run program:shared-matmul-c2
--env.CT_MATRIX_DIMENSION=128
--env.CT_BLOCK_SIZE=16
```

We can then reuse universal autotuning (exploration) strategies available as CK modules or implement specialized ones to explore exposed fine-grain optimizations versus different data sets. Figure 34 shows matmul performance in GFLOPS during random exploration of a blocking parameter for different square matrix sizes on RPi3. These results are in line with multiple past studies showing that unblocked matmul is more efficient for small matrix sizes (less than 32 on RPi3) since all data fits cache, or between 32 and 512 (on RPi3) if they are not power of 2. In contrast, the tiled matmul is better on RPi3 for matrix sizes of power of 2 between 32 and 512, since it can help reduce cache conflict misses, and for all matrix sizes more than 512 where tiling can help optimize access to slow main memory.

Our customizable workflow can help teach students how to build efficient, adaptive and self-optimizing libraries including BLAS, neural networks and FFT. Such libraries are assembled from the most efficient routines found during continuous crowd-tuning across numerous data sets and platforms, and combined with fast and automatically generated decision trees or other more precise classifiers [99, 83, 85, 64]. The most efficient routines are then selected at run-time depending on data set, hardware and other features as conceptually shown in Figure 35..

All demo scripts to generate data and graphs in this section are available in the following CK entries:

```
$ ck find
script:rpi3-all-autotune-multiple-datasets
$ ck find
script:rpi3-input-aware-autotune-blas
```

11 Reinventing computer engineering via reproducible competitions

Having a common and customizable workflow framework with "plug&play" artifacts opens up another interesting opportunity for computer engineering. Researchers can use it to compare and improve their techniques (optimizations, models, algorithms, architectures) against each other via open and reproducible competitions while being on the same page.

This is in spirit with existing machine learning competitions such as Kaggle and ImageNet challenge [15, 14] to improve prediction accuracy of various models. The main difference is that we want to focus on optimizing the whole software/hardware/model stack while trading off multiple metrics including speed, accuracy, and costs [31, 64, 24].

Experimental results from such competitions can be continuously aggregated and presented in the live Collective Knowledge scoreboard [8]. Other academic and industrial researchers can then pay a specific attention to the "winning" techniques close

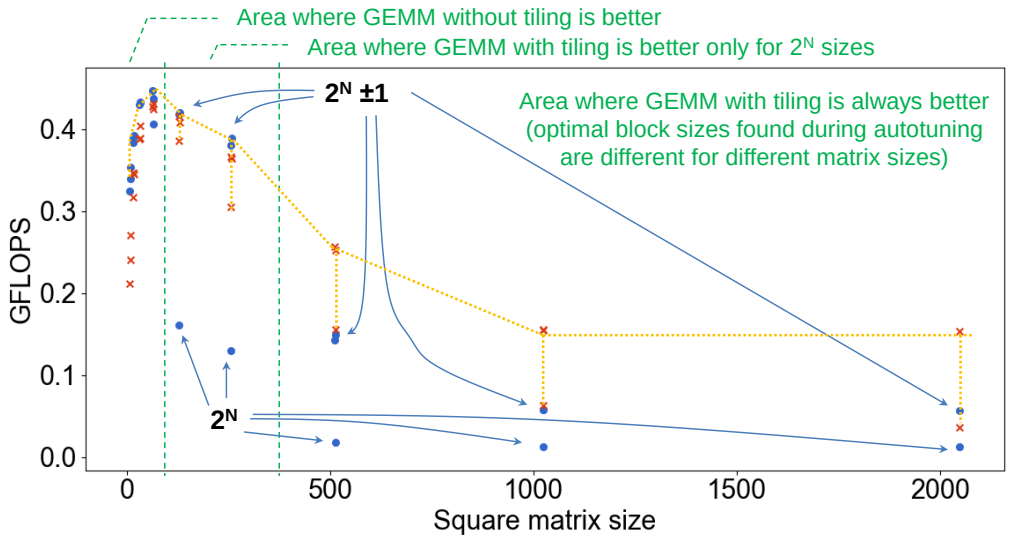


Figure 34: Performance of a tiled matrix multiply in GFLOPS for different square matrix sizes. Blue circles show performance of original (non-blocked) matrix multiply while red crosses show best performance found during autotuning on RPi3 device.

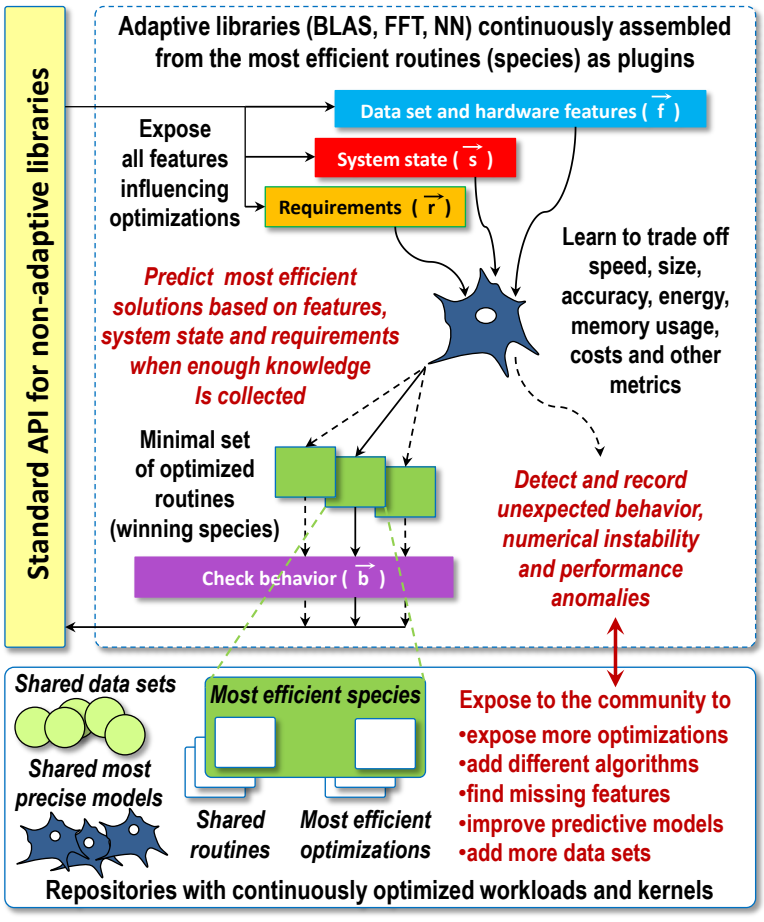


Figure 35: Enabling adaptive and self-optimizing libraries assembled from the most efficient routines continuously optimized by the community across different platforms and data sets. These routines are selected automatically at run-time based on platform, data set and other features.

to a Pareto frontier in a multi-dimensional space of accuracy, execution time, power/energy consumption, hardware/code/model footprint, monetary costs etc thus speeding up technology transfer. Furthermore, "winning" artifacts and workflows can now be recompiled, reused and extended on the newer platforms with the latest environment thus improving overall research sustainability.

For a proof-of-concept, we started helping some authors convert their artifacts and experimental workflows to the CK format during Artifact Evaluation [11, 36, 20]. Association for Computing Machinery (ACM) [1] also recently joined this effort funded by the Alfred P. Sloan Foundation to convert already published experimental workflows and artifacs from the ACM Digital Library to the CK format [55].

We can then reuse CK functionality to crowdsource benchmarking and multi-objective autotuning of shared workloads across diverse data sets, models and platforms. For example, Figure 36 shows results from random exploration of various SLAM algorithms (Simultaneous localization and mapping) and their parameters from [93] in terms of accuracy (average trajectory error or ATE) versus speed (frames per second) on RPi3 using CK [3]. Researchers may easily spend 50% of their time developing experimental, benchmarking and autotuning infrastructure in such complex projects, and then continuously updating it to adapt to ever changing software and hardware instead of innovating. Worse, such ad-hoc infrastructure may not even survive the end of the project or if leading developers leave project.

Using common and portable workflow framework can relieve researchers from this burden and let them reuse already existing artifacts and focus on innovation rather than re-developing ad-hoc software from scratch. Other researchers can also pick up the winning designs on a Pareto frontier, reproduce results via CK, try them on different platforms and with different data sets, build upon them, and eventually try to develop more efficient algorithms. Finally, researchers can implement a common experimental methodology to evaluate empirical results in systems research similar to physics within a common workflow framework rather than writing their own ad-hoc scripts. Figure 37 shows statistical analysis of experimental results implemented in the CK to compare different optimizations depending on research scenarios. For example, we report minimal execution time from multiple experiments to understand the limits of a given architecture, expected value to see how a given workload performs on average, and max time to detect abnormal behavior. If more than one expected value is detected, it usually means that system was in several different run-time states during experiments (often related to adaptive changes in CPU and GPU frequency due to DVFS) and extra analysis is required.

We now plan to validate our Collective Knowledge approach in the 1st reproducible ReQuEST tournament at the ACM ASPLOS'18 conference [31] as presented in Figure 38. ReQuEST is aimed at providing a scalable tournament framework, a common experimental methodology and an open repository for continuous evaluation and optimization of the quality vs. efficiency Pareto optimality of a wide range of real-world applications, libraries, and models across the whole hardware/software stack on complete platforms. ReQuEST also promote reproducibility of experimental results and reusability/customization of systems research artifacts by standardizing evaluation methodologies and facilitating the deployment of efficient solutions on heterogeneous platforms.

ReQuEST will use CK and our artifact evaluation methodology [20] to provide unified evaluation and a live scoreboard of submissions. Figure 39 shows a proof-of-concept example of such a scoreboard powered by CK to collaboratively benchmark inference (speed vs. platform cost) across diverse deep learning frameworks (TensorFlow, Caffe, MXNet, etc.), models (AlexNet, GoogleNet, SqueezeNet, ResNet, etc.), real user data sets, and mobile devices provided by volunteers (see the latest results at cKnowledge.org/repo). Our goal is to teach students and researchers how to

- release research artifacts of their on-going or accomplished research as portable and reusable components, standardize evaluation workflows, and facilitate deployment and tech transfer of state-of-the-art research,
- continuously optimize various algorithms across diverse models, data sets and platforms in terms of speed, accuracy, size, energy usage and other costs,
- build upon each others' work to develop next generation of efficient software and hardware stack for emerging workloads.

12 Conclusions and Future Work

Researchers are now in a race to bring artificial intelligence to all possible devices from IoT to supercomputers which will require much more efficient software and hardware then currently available. At the same time, computer engineers already struggle for many years to develop efficient sub-components of computer systems including algorithms, compilers and run-time systems.

The major issues including raising complexity, lack of a common experimental framework and lack of practical knowledge exchange between academia and industry. Rather than innovating, researchers have to spend more and more time writing their own, ad-hoc and not easily customizable support tools to perform experiments such as multi-objective autotuning.

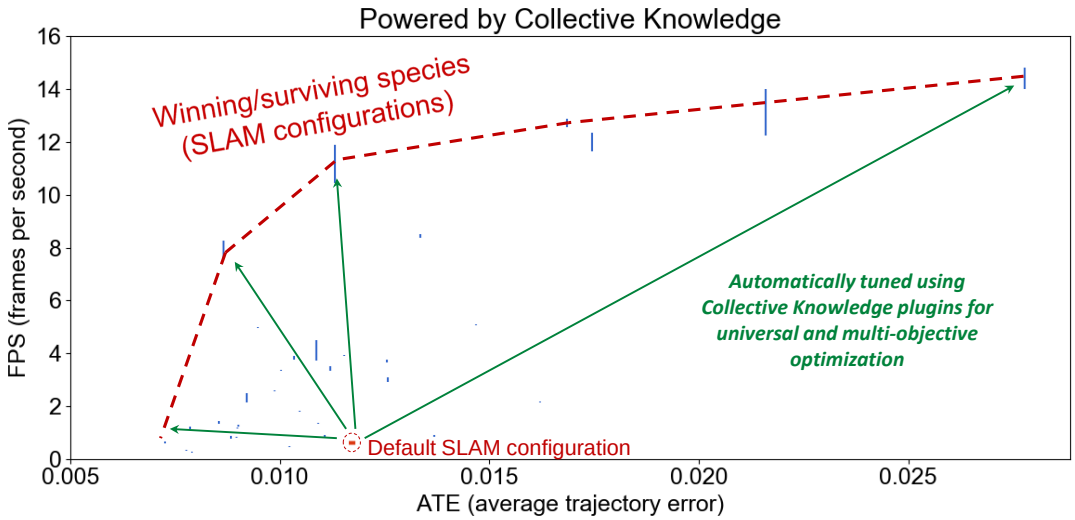


Figure 36: Random exploration of various SLAM algorithms and their parameters (Simultaneous localization and mapping) in terms of accuracy (average trajectory error or ATE) versus speed (frames per second) on RPi3 using CK.

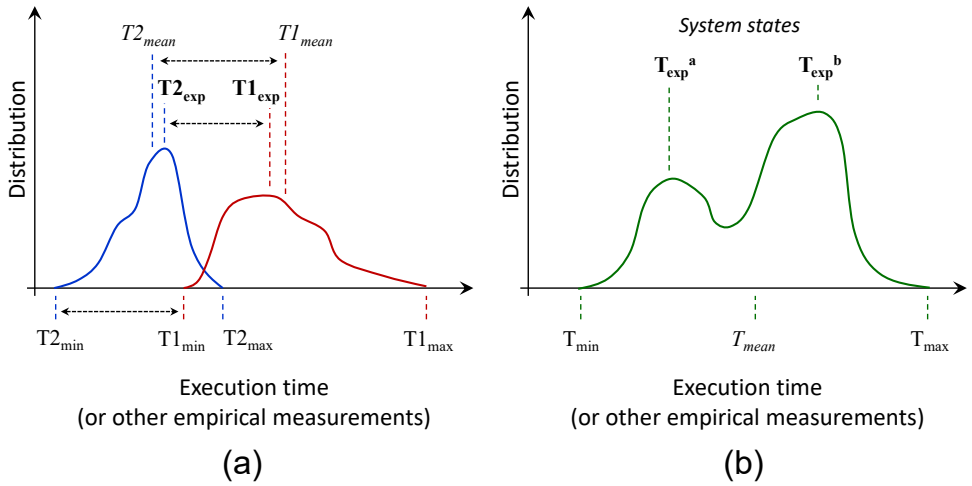


Figure 37: Developing common evaluation methodology for empirical results in systems research: (a) Calculating speedups between two optimizations T1 and T2 using min, mean and expected values, and reporting max difference. (b) Reporting a problem when several system states are detected

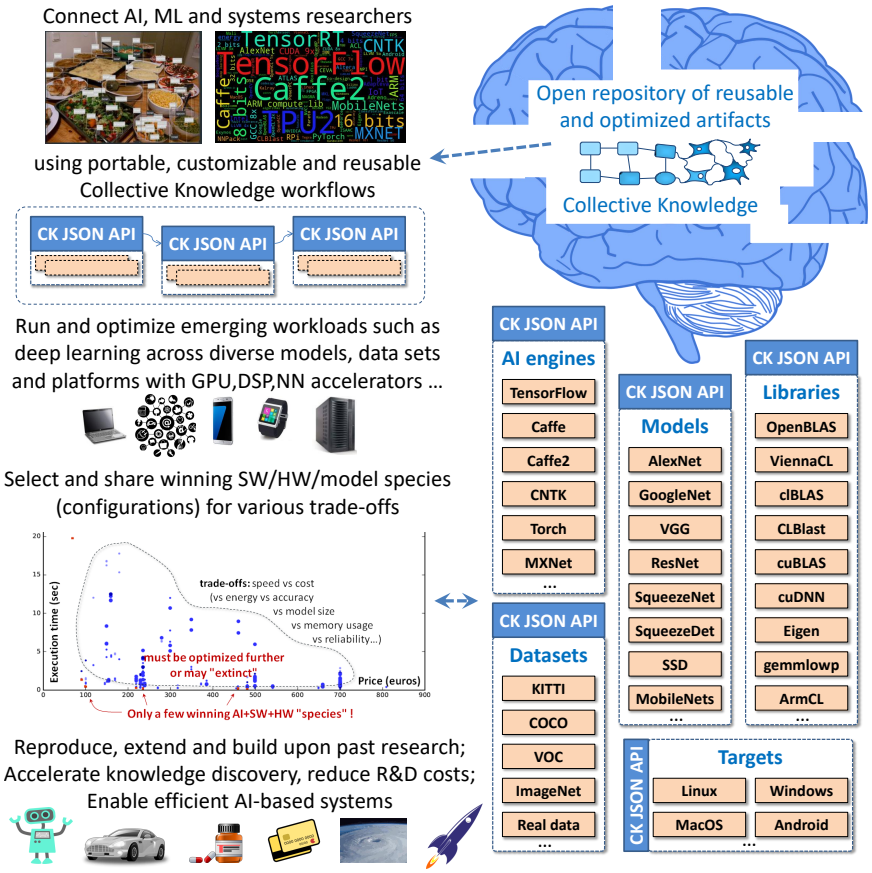


Figure 38: Collective Knowledge framework as an open platform to support software/hardware/model co-design tournaments for Pareto-efficient deep learning and other emerging workloads in terms of speed, accuracy, energy and various costs.

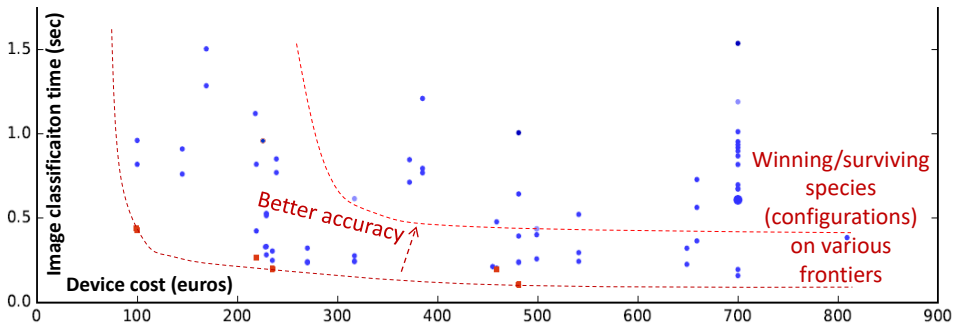


Figure 39: An example of a live Collective Knowledge scoreboard to crowd-benchmark inference in terms of speed, accuracy and platform cost across diverse deep learning frameworks, models, data sets, and Android devices provided by volunteers. Red dots are associated with the winning workflows (model/software/hardware) on different frontiers.

We presented our long-term educational initiative to teach students and researchers how to solve above problems using customizable workflow frameworks similar to other sciences. We showed how to convert ad-hoc, multi-objective and multi-dimensional autotuning into a portable and customizable workflow based on open-source Collective Knowledge workflow framework. We then demonstrated how to use it to implement various scenarios such as compiler flag autotuning of benchmarks and realistic workloads across Raspberry Pi 3 devices in terms of speed and size. We also demonstrated how to crowdsource such autotuning across different devices provided by volunteers similar to SETI@home, collect most efficient optimizations in a reproducible way in a public repository of knowledge at cKnowledge.org/repo, apply various machine learning techniques including decision trees, nearest neighbor classifier and deep learning to predict the most efficient optimizations for previously unseen workloads, and then continue improving models and features as a community effort. We now plan to develop an open web platform together with the community to provide a user-friendly front-end to all presented workflows while hiding all complexity.

We use our methodology and open-source CK workflow framework and repository to teach students how to exchange their research artifacts and results as reusable components with a a unified API and meta-information, perform collaborative experiments, automate Artifact Evaluation at journals and conferences [20], build upon each others' work, make their research more reproducible and sustainable, and eventually accelerate transfer of their ideas to industry. Students and researchers can later use such skills and unified artifacts to participate in our open ReQuEST tournaments on reproducible and Pareto-efficient co-design of the whole software and hardware stack for emerging workloads such as deep learning and quantum computing in terms of speed, accuracy, energy and costs [31]. This, in turn, should help the community build an open repository of portable, reusable and customizable algorithms continuously optimized across diverse platforms, models and data sets to assemble efficient computer systems and accelerate innovation.

13 Acknowledgments

We would like to thank Raspberry Pi foundation for initial financial support. We are also grateful to dividiti and cTuning foundation colleagues, Flavio Vella, Marco Cianfriglia, Nikolay Chunosov, Daniil Efremov, Yuriy Kashnikov, Peter Green, Thierry Moreau and ReQuEST colleagues, and the Collective Knowledge community for evaluating Collective Knowledge concepts and providing useful feedback.

References

[1] Association for Computing Machinery (ACM). http://www.acm.org.

[2] Blog article: CK concepts by Michel Steuwer. http://michel.steuwer.info/About-CK.

[3] Collective Knowledge workflows for SLAMBench. https://github.com/ctuning/reproduce-pamela-project.

[4] cTuning.org: public portal for collaborative and reproducible computer engineering. http://cTuning.org.

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

[6] Introducing JSON. http://www.json.org.

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

[8] Open collective knowledge repository with shared optimization results from crowdsourced experiments across diverse platforms and data sets. http://cKnowledge.org/repo.

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

[10] Public optimization results when crowd-tuning gcc 7.1.0 across raspberry pi3 devices. link.

[11] Public repositories with artifact and workflows in the Collective Knowledge format. http://cKnowledge.org/shared-repos.

[12] Static Features available in MILEPOST GCC V2.1. http://ctuning.org/wiki/index.php/CTools:MilepostGCC:StaticFeatures:MILEPOST_V2.1.

[13] Digital Object Identifier or DOI - a persistent identifier or handle used to uniquely identify objects, standardized by the iso. http://doi.org, 2000.

[14] Imagenet challenge (ILSVRC): Imagenet large scale visual recognition challenge where software programs compete to correctly classify and detect objects and scenes. http://www.image-net.org, 2010.

[15] Kaggle: platform for predictive modelling and analytics competitions. https://www.kaggle.com, 2010.

[16] Figshare - online digital repository where researchers can preserve and share their research outputs, including figures, datasets, images, and videos. http://figshare.com, 2011.

[17] Zenodo - research data repository. http://zenodo.org, 2013.

[18] Proceedings of the 1st workshop on reproducible research methodologies and new publication models in computer engineering (acm sigplan trust'14). ACM, 2014.

[19] TETRACOM - eu fp7 project to support technology transfer in computing systems. https://www.tetracom.eu, 2014.

[20] Artifact Evaluation for Computer Systems Conferences including CGO,PPoPP,PACT and SuperComputing: developing common experimental methodology and tools for reproducible and sustainable research. http://cTuning.org/ae, 2014-cur.

[21] Artifact evaluation for computer systems research. http://ctuning.org/ae, 2014-cur.

[22] Ck repository with multi-platform software and package manager implemented as ck modules which detect or install various software (compilers, libraries, tools). https://github.com/ctuning/ck-env, 2015.

[23] Ck workflow for opencl crowd-fuzzing. https://github.com/ctuning/ck-clsmith, 2015.

[24] LPIRC: low-power image recognition challenge. https://rebootingcomputing.ieee.org/lpirc, 2015.

[25] Collective Knowledge: open-source, customizable and cross-platform workflow framework and repository for computer systems research. https://github.com/ctuning/ck, 2016.

[26] Continuous fuzzing of open source software. https://github.com/google/oss-fuzz, 2016.

[27] Microsoft security risk detection. https://www.microsoft.com/en-us/security-risk-detection/, 2016.

[28] Artifacts and experimental workflows in the Collective Knowledge Format for the CGO'17 paper "Software Prefetching for Indirect Memory Accesses". https://github.com/SamAinsworth/reproduce-cgo2017-paper, 2017.

[29] The HiPEAC vision on high-performance and embedded architecture and compilation (2012-2020). http://www.hipeac.net/roadmap, 2017.

[30] Open benchmarking: automated testing & benchmarking on an open platform. http://openbenchmarking.org, 2017.

[31] ReQuEST: open tournaments on collaborative, reproducible and pareto-efficient software/hardware co-design of emerging workloads such as deep learning using collective knowledge technology. http://cKnowledge.org/request, 2017.

[32] 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.

[33] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[34] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra. Performance, design, and autotuning of batched GEMM for gpus. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 21–38, 2016.

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

[36] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 305–317, Piscataway, NJ, USA, 2017. IEEE Press.

[37] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, Nov. 2002.

[38] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. 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.

[39] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[40] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[41] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Trans. Archit. Code Optim.*, 14(3):29:1–29:28, Sept. 2017.

[42] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):21, 2016.

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

[44] D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. PERI auto-tuning. *Journal of Physics: Conference Series*, 125(1):012089, 2008.

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

[46] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1997.

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

[48] B. R. Childers, G. Fursin, S. Krishnamurthi, and A. Zeller. Artifact Evaluation for Publications (Dagstuhl Perspectives Workshop 15452). 5(11), 2016.

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

[50] C. Ţăpuş, I.-H. Chung, and J. 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.

[51] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, pages 219–232, 2017.

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

[53] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. 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.

[54] J. W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[55] P. Flick, C. Jain, T. Pan, and S. Aluru. A parallel connectivity algorithm for de bruijn graphs in metagenomic applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 15:1–15:11, New York, NY, USA, 2015. ACM.

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

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

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

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

[60] G. Fursin and C. Dubach. 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.

[61] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. F. P. O'Boyle. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming (IJPP)*, 39:296–327, 2011. 10.1007/s10766-010-0161-2.

[62] G. Fursin, A. Lokhmotov, and E. Plowman. Collective Knowledge: towards R&D sustainability. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'16)*, March 2016.

[63] G. Fursin, A. Lokhmotov, and E. Upton. Collective knowledge repository with reproducible experimental results from collaborative program autotuning on raspberry pi (program reactions to most efficient compiler optimizations). https://doi.org/10.6084/m9.figshare.5789007.v2, Jan 2018.

[64] 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.

[65] 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*, 22(4):309–329, July 2014.

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

[67] G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency: Practice and Experience*, 16(2-3):271–292, 2004.

[68] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 40:1–40:12, New York, NY, USA, 2015. ACM.

[69] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

[70] D. Grewe, Z. Wang, and M. F. P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 22:1–22:10, 2013.

[71] M. Hall, D. Padua, and K. Pingali. Compiler research: The next 50 years. *Commun. ACM*, 52(2):60–67, Feb. 2009.

[72] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–11, 2009.

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

[74] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt. Easybuild: Building software with ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pages 572–582, 2012.

[75] V. Jimenez, I. Gelado, L. Vilanova, M. Gil, G. Fursin, and N. Navarro. Predictive runtime code scheduling for heterogeneous architectures. In *Proceedings of the International Conference*

on High Performance Embedded Architectures & Compilers (HiPEAC 2009), January 2009.

[76] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Archit. Code Optim.*, 9(4):31:1–31:25, Jan. 2013.

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

[78] 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.

[79] A. Lascu and A. F. Donaldson. Integrating a large-scale testing campaign in the CK framework. *CoRR*, abs/1511.02725, 2015.

[80] C. Lattner and V. 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.

[81] H. Leather, E. V. Bonilla, and M. F. P. O'Boyle. Automatic feature generation for machine learning-based optimising compilation. *TACO*, 11(1):14:1–14:32, 2014.

[82] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 65–76, New York, NY, USA, 2015. ACM.

[83] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, May 2009.

[84] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism*, volume 6, 2004.

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

[86] I. Manotas, L. Pollock, and J. 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.

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

[88] J. Mars, N. Vachharajani, R. Hundt, and M. L. 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.

[89] 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.

[90] R. 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.

[91] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.

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

[93] 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.

[94] 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.

[95] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

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

[97] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming*, 41(5):704–750, 2013.

[98] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[99] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. R. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[100] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, July 2010.

[101] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

[102] C. Sammut and G. Webb. *Encyclopedia of Machine Learning and Data Mining*. Springer reference. Springer Science + Business Media.

[103] J. Shen, A. L. Varbanescu, H. J. Sips, M. Arntzen, and D. G. Simons. Glinda: a framework for accelerating imbalanced applications on heterogeneous platforms. In *Conf. Computing Frontiers*, page 14, 2013.

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

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

[106] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.

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

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

[109] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., Norwood, MA, USA, 1 edition, 2008.

[110] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[111] Y. M. Tsai, P. Luszczek, J. Kurzak, and J. J. Dongarra. Performance-portable autotuning of opencl kernels for convolutional layers of deep neural networks. In *2nd Workshop on Machine Learning in HPC Environments, MLHPC@SC, Salt Lake City, UT, USA, November 14, 2016*, pages 9–18, 2016.

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

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

[114] D. Wilkinson, B. Childers, R. Bernard, W. Graves, and J. Davidson. Acm pilot demo 1 - collective knowledge: Packaging and sharing. version 3. 2017.

[115] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

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

A Artifact Appendix

Submission guidelines:

cTuning.org/ae/submission-20161020.html

This is an example of an Artifact Appendix which we introduced at the computer systems conferences including CGO, PPoPP, PACT and SuperComputing to gradually unify artifact evaluation, sharing and reuse [20, 48, 60, 58]. We briefly describe how to install and use our autotuning workflow, visualize optimization results and reproduce them. We also shared all scripts which we used to generate data and graphs in all sections from this report though we did not yet have time to thoroughly document them. In fact, we plan to gradually document them and standardize APIs of shared CK modules with the help of the community and motivated students.

A.1 Abstract

We provided the whole Collective Knowledge workflow with all dependencies for collaborative, customizable, multi-dimensional and multi-objective autotuning of realistic workloads on Raspberry Pi 3 and other devices.

Current optimization results are available for GCC 7.1.0 (link) and for GCC 4.9.2 (link). They are also available as a CK repository and can be replayed on another platform via CK.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:** -
- **Program:** shared programs from the CK ctuning-programs repository
- **Compilation:** any GCC
- **Transformations:** compiler flag optimizations
- **Binary:** will be produced during autotuning
- **Data set:** real inputs from the CK ctuning-datasets-min repository
- **Run-time environment:** Raspbian (or any other)
- **Hardware:** Raspberry Pi 3 (or any other)
- **Run-time state:** will be monitored by CK (CPU frequency)
- **Execution:** empirical measurements of the execution time of autotuned workloads via CK workflow
- **Output:** best combinations of GCC compiler flags that improve execution time and code size
- **Experiment workflow:** autotuning, crowd-tuning and collaborative machine learning workflow implemented using CK framework
- **Experiment customization:** standard customization via CK API: select compiler, programs and data sets for autotuning, crowd-tuning and predictive modeling
- **Publicly available?:** yes - CK autotuning and machine learning workflow (available under BSD 3-clause license) and all related artifacts are shared as reusable and customizable components via GitHub.

A.2.2 How software can be obtained (if available)

You can obtain CK repositories with optimization results, shared programs and data sets, workflow for autotuning and crowd-tuning as following:

```
$ sudo pip install ck
$ ck pull repo:ck-rpi-optimization-results
```

Note that you may need around 1GB of free space. You can install 2 additional CK repositories [63] from the public FigShare repository as following (need 3GB of free space):

```
$ ck add
repo:ck-rpi-optimization-results-reactions
--zip=https://ndownloader.figshare.com/files/10218435
--quiet
$ ck add
repo:ck-rpi-optimization-results-reactions2
--zip=https://ndownloader.figshare.com/files/10218441
--quiet
$ ck ls experiment:rpi3-*
```

These repositories are so large because they contain all experiments from this report in a reproducible way (we also plan to considerably reduce this size by removing duplicate information in the future). But if you want to prepare and run your own repositories you will likely need less than 100MB. See this artifact in the CK from the ACM CGO'17 paper [36] as example: github.com/SamAinsworth/reproduce-cgo2017-paper

A.2.3 Hardware dependencies

Tested on Raspberry Pi Model B 3 devices with 4-core BCM2709 processor but should work on any platform.

A.2.4 Software dependencies

- Raspbian GNU/Linux 8 (jessie)
- Collective Knowledge Framework [25, 62]
- Python 2.7+ or 3.4+
- Git client
- GCC 4.9.2 or GCC 7.1.0

A.2.5 Data sets

A minimal set of inputs for cTuning benchmarks available from the CK ctuning-datasets-min repository.

A.3 Installation

Installation is performed using CK with the help of integrated cross-platform package manager [22]:

```
$ sudo pip install ck
$ ck pull repo:ck-rpi-optimization-results
$ ck compile program:zlib --speed
```

CK will automatically detect required software which is already installed on your platform, install missing packages, and prepare autotuning workflow for execution.

Note that CK allows multiple versions of different software to natively co-exist. Therefore, you can install several versions of GCC which will be automatically detected by CK and their environment prepared accordingly. For example, you can install (build) GCC 7.1.0 on RPi 3 via CK as following:

```
$ ck pull repo:ck-dev-compilers
$ ck install
package:compiler-gcc-any-src-linux-no-deps
--env.PARALLEL_BUILDS=1
--env.GCC_COMPILE_CFLAGS=-O0
--env.GCC_COMPILE_CXXFLAGS=-O0
--env.EXTRA_CFG_GCC=--disable-bootstrap
--env.RPI3=YES --force_version=7.1.0
$ ck show env --tags=gcc
```

Note that you may need to install extra dependencies including

```
$ sudo apt-get install texinfo build-essential
libgmp-dev libmpfr-dev libisl-dev
libcloog-isl-dev libmpc-dev
```

You may also want to increase swap size on RPi 3 to speed up GCC building. You can change "CONF_SWAPSIZE=100" in /etc/dphys-swapfile to "CONF_SWAPSIZE=1000". But do not forget to change it back after successful build to avoid damaging your SD card.

A.4 Experiment workflow

Autotuning example

You can run zlib autotuning via CK as following:

```
$ ck autotune program:zlib --iterations=150
--repetitions=3 --scenario=9d88674c45b94971
--cmd_key=decode
--record_uoa=my-first-experiment
```

CK will automatically detect available compilers, will ask user to select data set, and will evaluate 150 combinations of random compiler flags (repeating each experiment 3 times for statistical analysis of empirical variation of results).

Experimental results will be aggregated in a CK entry "experiment:my-first-experiment" a local CK repository:

```
$ ck find experiment:my-first-experiment
```

You can plot graph (execution time vs binary size) or view results in a web browser as following:

```
$ ck plot graph:my-first-experiment
$ ck browser experiment:my-first-experiment
```

You can compile and run zlib program via CK as following:

```
$ ck compile program:zlib --flags="some flags"
$ ck run program:zlib
```

Finally, you can participate in GCC crowd-tuning as following:

```
$ ck crowdsource program.optimization --gcc
```

A.5 Evaluation and expected result

You can find all scripts to perform experiments from this article as following:

```
$ ck ls ck-rpi-optimization-results:script:* |
sort
```

You can then go to each individual entry and see related scripts:

```
$ ls ‘ck find script:rpi3-susan-autotune‘
```

You can find all experimental results in the following entries:

```
$ ck ls
ck-rpi-optimization-results:experiment:* |
sort
```

You can then browse all results in your web browser as following:

```
$ ck browser
experiment:rpi3-zlib-decode-gcc4-150b-rnd-frontier
```

You can find information about how to replay each autotuning iteration there, for example:

```
$ ck replay experiment:b0f31c56475aa510
--point=46049203405c5347
```

CK should normally show expected and new results while reporting any unexpected behavior (if difference is more than some threshold such as 5%).

A.6 Experimental methodology

One of the most important points of using Collective Knowledge framework is to take advantage of the experimental methodology for computer systems research continuously improved by the community. For this purpose, we instrument programs using small xOpenME library which allows us to monitor behavior of some code regions and dump final statistics to a JSON file in the CK format. CK will then repeat each autotuning iteration N times, apply statistical analysis on all exposed characteristics, report min, max and mean values, and calculate expected value based on a histogram of all results (if supported by used Python) as shown in Figure 37 (a).

We then calculate improvements of a given optimization over reference one (-O3) using minimal and expected execution times, and record differences. If the difference is more than 5%, we mark such experient is noise and untrustable to be analyzed and improved later by the community. If several system states are detected as shown in Figure 37 (b), CK will not be able to reproduce them - it then means that the common CK experimental workflow should also be improved for this hardware and environment to be able to distinguish such states (such as CPU and GPU frequency due to DVFS for example).

A.7 Notes

We did not have time to thoroughly document experiments from sections 7+ of this report. However we shared all CK modules, workflows and scripts we used in this report in the following CK entries:

```
$ ck ls script:rpi3-*
$ ck ls converting-ad-hoc-works-to-ck-*
```

Scripts from Sections 3 and 4 to invoke portable and customizable CK autotuning workflow:

```
$ ck find script:rpi3-susan-autotune
$ ck find script:rpi3-susan-graphs
$ ck find script:rpi3-susan-reduce
$ ck find script:rpi3-all-autotune
```

Scripts from Section 5:

```
$ ck find script:rpi3-all-autotune
$ ck find script:rpi3-crowdtune
```

Scripts from Section 6:

```
$ ck find script:rpi3-zlib-decode-autotune
$ ck find script:rpi3-zlib-decode-graphs
$ ck find script:rpi3-zlib-decode-reduce
$ ck find script:rpi3-zlib-encode-autotune
$ ck find script:rpi3-zlib-encode-graphs
$ ck find script:rpi3-zlib-encode-reduce
```

Scripts from Section 7:

```
$ ck find script:rpi3-susan-fuzz-bugs
```

Scripts from Sections 8 and 9:

```
$ ck find script:rpi3-crowdmodel
```

Scripts from Section 10: //input-aware

```
$ ck find
script:rpi3-all-autotune-multiple-datasets
$ ck find
script:rpi3-input-aware-autotune-blas
```

Scripts from Section 11:

```
$ ck find
script:converting-ad-hoc-works-to-ck-slambench-autotuning
```

A.8 Conclusion

We hope that our customizable autotuning and machine learning workflow can teach students, scientists and engineers learn how to collaboratively co-design Pareto-efficient software and hardware stack for emerging workloads. Please feel free to send us updates and patches to fix, help us to improve or extend our artifacts with documentation, and keep in touch with our community via CK mailing list: groups.google.com/d/forum/collective-knowledge!