BenchBuild: A Large-Scale Empirical-Research Toolkit

Andreas Simbürger, Floarian Sattler, Armin Größlinger, Christian Lengauer

Department of Informatics and Mathematics, University of Passau
{simbuerg,sattler,groessli,lengauer}@fim.uni-passau.de

Technical Report, Number MIP-1602
Faculty of Computer Science and Mathematics
University of Passau, Germany
June 2016

BenchBuild: A Large-Scale Empirical-Research Toolkit

Andreas Simbürger Florian Sattler Armin Größlinger
Christian Lengauer
Faculty of Informatics and Mathematics
University of Passau
{simbuerg,sattlerf,groessli,lengauer}@fim.uni-passau.de

ABSTRACT

The efficiency of software is commonly evaluated with one or more suites of experiments at compile time or run time. The manual preparation of such experiments is tedious and error-prone, involving tasks such as intercepting the compiler at hand or the resulting binaries with custom measurements. This imposes a practical limit on the number of case studies. We present BenchBuild, a large-scale empirical-research toolkit that supports 18978 projects for compile-time and 188 projects for run-time testing. BenchBuild automates most of the tasks involved and provides tools that reduce the amount of effort required to increase the test coverage.

https://youtu.be/VtXdveMT1Rk

CCS Concepts

•**Software and its engineering** → **Software usability; Software testing and debugging;** *Extra-functional properties; Preprocessors;*

Keywords

Benchmarking; compile-time testing; run-time testing; build systems

1. INTRODUCTION

Automatic benchmarking is not only essential for compiler writers, but also for software engineering research. Providing a sound and thorough evaluation of software experiments always faces the same trade-off: do we focus on comparability, by using custom-tailored benchmarks, or on generality, by using a large number of real-world case-studies. The former is usually easier to implement, while the latter provides insight into wide-range applicability. However, preparing a large number of case studies for the setup of an experiment is a daunting task and, therefore, one often sticks with the easier-to-handle, ready-to-use benchmark suites.

ACM ISBN -.
DOI: -

Sites like SPEC[1] provide ready-to-use test suites that facilitate program testing by building and running them with preselected inputs automatically. Many such suites are widely accepted inside their respective communities and have the advantage that the results of different experiments are comparable. The downsides are that preselected inputs may be unsuitable for the hypotheses and, additionally, the test programs selected may not be representative of real-world scenarios. The identification of test inputs that address the given hypotheses is often laborious. In addition, if the test suite does not provide a suitable input set, the custom generation of one reduces comparability. If one is interested in compile-time tests, adjusting the test suites causes further problems: it becomes necessary to understand the build process of all programs in the customized suite and to be able to intercept it with the customized measurement. One way to do so is by substituting the compiler with a hand-crafted script, which performs the compilation and the measurement. This is a tedious and time-consuming process – time that is better spent on identifying good test inputs and answering the research question itself.

Our automated testing tool BenchBuild is meant to lend support for the latter by providing a highly adaptable and easy-to-use testing framework and contributes:

- a lightweight toolkit for wrapping compilation and binary invocation with user-defined measurements,
- tool uchroot for changing the file system root as unprivileged user.
- customizable support for automatic experiments
  - for 18978 software systems at compile time,
  - for 188 software systems at run time.

BenchBuild is available at:
https://github.com/simbuerg/benchbuild.git

2. RELATED WORK

BenchBuild is not the only framework and test suite for automatic test configuration.

SPEC provides different benchmark suites for performance tests, e.g., for CPUs CPU2006 [5]. Each suite is a collection of programs with different test inputs. It facilitates the automatic build all programs, measurement of the run time, and generation of performance reports.

Another tool for performance testing is the LLVM-based Codelet Extractor and REplayer (CERE) [1]. The tool takes applications as input and extracts codelets, i.e., performance-relevant parts of the program. The REplayer executes

[1]https://spec.org

the codelet in roughly the context of the normal program run by rebuilding the memory and bringing the cache into a similar state. This enables performance prediction and optimizations of small parts of the program without running the entire program.

For compiler developers, the LLVM framework provides its own testing infrastructure LNT[2]. This aids the developer in writing compiler tests and benchmarks, but also ties him to LLVM. The emphasis lies on compile-time testing of a variety of input programs, ranging from simple unit tests to multi-file applications in different programming domains.

A useful tool for setting up a testing infrastructure is DOCKER [3]. It provides a virtual Linux operating system and eases the setup of lightweight Linux containers. The user creates a base image for testing, e.g., UBUNTU, and then different DOCKER containers for each program that is part of the experiment. Combined with DOCKER, services like Open Build Systems[3] can build packages for different Linux distributions. This setup enables compile-time and run-time tests. However, the container setup and the integration of the experiment with the containers remain complex.

Software mining tools like Boa [2] enable the user to analyze software projects hosted on SourceForge[4]. Boa evaluates the program's source code to generate information like the churn rate, i.e., the number of files changed in a revision. In contrast to BENCHBUILD, Boa does not compile or run the program. Additionally, it is currently limited to Java source code, whereas BENCHBUILD is language-independent.

3. BENCHBUILD

In the following subsections, we introduce BENCHBUILD, our large-scale toolkit for empirical research that aims at remedying the shortcomings of the manual approach described previously. We focus on the key tools that are necessary to provide automated wrapping of the compiler and binary with customized measurement functions offered by a wide variety of easily extensible case studies.

3.1 The goal of BenchBuild

Frequently, during research, a point arrives at which a new result has to be evaluated against a set of custom-tailored case studies, benchmarks, or a large code base of an individual program – a process that is laborious, tedious and error-prone. Depending on the kind of measurement setup that is required, it is necessary to acquire a deep understanding of each program's build system and run-time behavior.

A simple setup might just require a script that wraps the default compiler that translates the program. This seemingly easy task becomes much more complicated when one has to consider a variety of build systems such as GNU MAKE, CMAKE, or NINJA. All of them support a degree of freedom that makes it very error-prone to reliably intercept the compilation process with custom measurements.

Expanding the experimental setup with run-time tests for each case-study adds new tasks besides the existing ones that one has to tackle for compilation wrapping. Formulating run-time tests for new case-studies always risks to introduce measurement bias to the experiment setup even if there is a set of program inputs that domain experts agreed upon.

[2]http://llvm.org/docs/lnt/
[3]http://openbuildservice.org/
[4]https://sourceforge.net/

All these tasks require a deep understanding of the program. This often leads to a low number of case-studies used during the evaluation of our research results.

Here is where BENCHBUILD tries to remedy the current situation. We remove the need for a deep understanding of a case-study's build-system. The user only has to find representative test-inputs in case he wants conduct run-time testing on new case-studies.

3.2 Terminology

Observed from the outside, BENCHBUILD executes experiments on a set of programs. At the inside, it maintains the configuration necessary for all programs to conduct their experiments. The configuration includes instructions on how to build the program, where to put the binaries to be measured and (optionally) a set of representative program inputs to each of them. Each project itself is treated as an atomic unit within BENCHBUILD. However, actual process isolation is not enforced automatically. The enforcement might introduce unwanted overhead during the measurements and is, therefore, left to the user-defined experiment. As a first step, BENCHBUILD includes support for the Simple Linux Utility for Resource Management (SLURM) [7]. Besides, resource management it also enables parallel execution on remote systems.

Project.

A *project* in BENCHBUILD's terminology aggregates all information that is necessary to download and build components required by the program to be tested, and to execute a representative set of commands that facilitate the run-time testing. Like other build systems and package managers, BENCHBUILD only provides precise dependency tracking if specified by the metadata supplied by the user.

A project is identified by a `NAME` and a `DOMAIN` – a grouping attribute that eases mass selection of project groups. Projects must satisfy the following interface:

`def download()`: Imports all necessary sources into the local build directory. This will also download the sources from remote locations, if possible. All downloads are cached locally and hashed, to provide stable input for all further experiments.

`def configure()`: Prepares the project for a compilation. Depending on the experiment, this is the place at which the optional compiler extension is placed in the build directory.

`def build()`: Builds the project with all its dependences. All compiler measurements take place during the execution of this method.

`def run_tests(experiment)`: Executes all run-time tests of the project. Takes the experiment-specific run function and creates a wrapped binary that executes it instead of the actual binary.

Experiment.

An *experiment* in BENCHBUILD is characterized by two concepts. First, there is the flow of actions, i.e., the sequence of calls to some projects' API methods. Second, a serialized compile and/or run function to perform the measurements is combined with the binaries of interest. These are completely user-definable and are required to return an execution plan for each project. Predefined actions are available in BENCHBUILD to form an execution plan. Examples include wrappers that

call a project's API methods or group-actions that require all contained actions to succeed.

3.3 Database schema

All experiments conducted by BenchBuild are stored in a simple – extensible – database schema that provides the user with different levels of precision. Let us go through them from coarse to fine grain:

`experiment`: At the coarsest level, BenchBuild keeps track of the experiment instances to be run. An instance can be selected via its unique identifier, name, or start/end time.

`rungroup`: Inside an experiment instance, we can combine measurements in a *rungroup*. As soon as a project executes its run-time tests, all single binary runs are made one rungroup, i.e., a defined set of – possibly multiple – binary calls.

`run`: The level of highest precision provides the *run*. Here we can analyze every single binary call that executed inside a BenchBuild measurement.

`config`: Beside the multiple levels of precision above, BenchBuild also logs the configuration metadata of each experiment and rungroup. This way, it keeps track of the characteristics of the measurement, such as program versions, host configuration, or low-level information such as the number of cores used in the execution of an individual rungroup.

3.4 Compiler and run-time wrapping

To enable BenchBuild to abstract from the tedious task of manual compile-time and run-time measurements it needs a way of augmenting the compiler, as well as any other binaries of interest with a customized measurement *extension function*. In BenchBuild, the user can specify custom experiments or case studies not in a custom domain-specific language but in the implementation language of BenchBuild, Python. A binary is augmented with a customized measurement function is done via generic wrapper scripts that load a serialized form of the measurement function and execute it instead of the original binary. Python provides – among others – the `pickle` package to perform arbitrary object serialization.

The substitution of the compiler is a common task of any build system. Even without external measurements wrapped around the compilation process, build systems provide a means to choose the compiler that should be used, e.g., via environment variables (`CC`, `CXX`). However, special support for different compilers needs to be considered inside the project's build system itself to accommodate for deviating call syntax and/or behavior of the compiler.

A custom compiler can be integrated via a simple script that exposes the same API as a supported compiler. Listing 1 contains a minimal script in shell syntax. It can be used as the default compiler for any project that supports gcc and enables the transformation of the incoming arguments and the invocation of any binary call in place of the call to the original compiler. The script can be extended to construct more sophisticated wrappers – a manual and, thus, error-prone procedure.

Beside our own custom measurement, we need to be able to complete the compilation process successfully. A simple way of ensuring a successful compilation is to call the originally intended compiler for the project as well, liberates the compiler extension from explicitly invoking the intended compilation command. In addition, BenchBuild's compiler extension includes the capability to fall back to the originally

```
#!/bin/sh
custom-cc $*
gcc $*
```

Listing 1: Minimal script to intercept the default compilation call

intended compilation command in case of error. This strips away any customizations that may have been configured in a BenchBuild experiment and attempts completion of the build without them.

```
from pprof.compiler import mycflags, myldflags, CC
import sys
def invoke_external_measurement(*args): pass
def custom_cc(*args): pass
def default_cc(*args): pass

flags = sys.argv[1:]
retcode = 0
try:
  retcode, _, _ = \
    custom_cc(CC, flags,
              mycflags, myldflags, input_files)
  invoke_external_measurement(fc)
except ProcessExecutionError:
  retcode, _, _ = \
    default_cc(CC, flags, input_files)

sys.exit(retcode)
```

Listing 2: Schematic overview of BenchBuild's compiler wrapper. First, we run the compiler with all custom flags the user might have added due to his experiment. On success, we invoke the external measurement, otherwise we revert all custom flags and execute the original compiler with its default flags for the project.

Listing 2 provides a schematic overview of the generic compiler wrapper that is generated by BenchBuild. At first, we run the compiler with all custom flags specified for the experiment. Upon success, the compiler extension is invoked. This ensures that the flags added during the experiment do not trip up the extended compiler. Should the compilation with customized flags fail, compilation is repeated without them. Errors encountered during compilation are logged.

Wrapping a binary is – in principle – the same as wrapping the compiler. The same approach of a simple wrapper script that takes care of error handling applies, except for fallback on error, in addition.

Our aim is to be minimally invasive. The approach of wrapping ensures that the impact of a modification stays invisible to the environment. Neither the build system nor a customized test suite can detect the modification without explicitly validating the binaries that it calls.

3.5 Invoking an external measurement

So far, we are able to wrap any binary with a customized wrapper script that loads and executes an extension by calling the `def invoke_external_measurement` function. A call of this function loads and executes a serialized version of our

custom extension. Both the configurable compiler and run-time extension can be implemented likewise. For simplicity, from here on, we refer to the run-time extension function when we mention an *extension*.

An extension is required to obey the following signature:

```
def fn(cmd, args, **kwargs)
```

Here, `cmd` is the command that actually needs to run, `args` are the program arguments, and `**kwargs` are any keyword arguments supplied by the wrapper script.

Any function whose signature matches this interface can be used as a BenchBuild run-time extension.

The user only needs to provide this one run-time extension that performs the measurement and stores the results in any desired format. BenchBuild provides additional tools to identify each binary run in the scope of the entire experiment for further analysis.

3.6 Switching the root file system

Unix-like operating systems have long offered a mechanism for multiple installations that can be used in parallel: change root (chroot) environments. A chroot environment is a file system directory which, for a certain process to be started, becomes the new base of the file system hierarchy (i.e., the directory "/") after the chroot operation. Unfortunately, performing a chroot operation requires superuser privileges and is, therefore, usually not available to users in a multi-user environment. A more heavy-weight alternative is to run a virtual machine (VM), which can be allowed for users without security risks. The drawback of using virtual machines is that the management of the VM's file systems is less convenient (usually, a big image file serves as the VM's storage device) and running a separate operating system kernel in the VM incurs overhead.

A third option that has become available recently in Linux is to use containers. Containers are similar to chroot environments. In addition to chroots, the kernel offers more isolation between the main system and containers by (optionally) virtualizing additional resources, e.g., the space of user IDs and the space of process IDs. Due to the virtualization of user IDs, the superuser (user ID 0) in the container need not actually be privileged on the system. These so-called unprivileged containers allow users to execute processes in such containers with any user ID (including 0) inside the container but, from the outside perspective, the process runs with some unprivileged user ID. The relation between inner and outer user IDs is given by an ID mapping which has to be set during the initialization of the container. For an easy use of unprivileged containers in BenchBuild, we have implemented a tool called uchroot ("user change root") which allows a user to perform a change root operation to a new base directory B. In detail, uchroot performs the following steps:

1. Create an unprivileged container.
2. Mount `/dev`, `/proc` and `/sys` file systems under B.
3. As root in the container perform a chroot syscall to change the container's file system base to B.
4. Establish the mapping between the container's user and group IDs and outside user and group IDs, respectively.
5. Execute a user-supplied command with a given user and group ID in the container.

Note that none of these operations require that the user invoking uchroot have superuser privileges (nor does uchroot need to be installed with setuid or setgid rights). There are two options (both supported by uchroot) of defining the user and group ID mappings. The simple option is that only one user ID (for example, ID 0) is necessary inside the container. In this case, the inner ID (e.g., 0) can simply be mapped to the ID of the invoking user (outside the container). The second option is that more than one user ID are necessary inside the container. This requires so-called subordinate user and group IDs that the system administrator can assign by configuring the files `subuid` and `subgid` in `/etc`.

A further feature of uchroot is file system attribute emulation. Not every file system can deal with subordinate user IDs; in particular, NFS does not support them at present, so the processes running under subordinate user IDs would not be able to create files on the file system. To allow users to store files that have arbitrary user IDs and file access modes inside the container, uchroot also provides a virtual file system, implemented using FUSE (File System in Userspace) which emulates user IDs and access modes. When this emulation is activated, the virtual file system is mounted on a temporary mount point. This mount point becomes the new base of the file system hierarchy in the container after the chroot operation. All accesses to the virtual file system are redirected transparently to the directory B (provided by the user) with access modes and user IDs stored in additional files (in directory B). All files created under B are owned by the invoking user, i.e., the user does not need elevated privileges on this file system.

Putting all these ingredients together, uchroot (by exploiting Linux's unprivileged containers) allows an unprivileged user to create and use a change root environment without the need for special privileges, setuid binaries[5] or file system access for user IDs other than the user's normal ID.

3.7 Available projects

Up to this point, we have described all tools necessary to infuse an arbitrary measurement into any case study, enabling the user to add any software system to BenchBuild's project registry. The downside is that it still remains a – one-time – task to resolve all dependencies and understand the build system. On the upside, BenchBuild already provides 18978 ready-to-use software systems.

BenchBuild provides compile-time and run-time test support for LLVM's test suite lnt, the SPEC benchmark suite and a specialized suite of real-world programs used by us [4]. Together, they consist of 188 software systems.

In addition, BenchBuild also comes with support for all software systems that are included in the highly customizable Linux distribution Gentoo Linux[6] that, on package installation, compiles all software packages from source by default [6]. Other Linux distributions, e.g., Debian-based can be used too, but are not implemented at present.

Via the uchroot environment introduced in Section 3.6, BenchBuild provides a means to set up a base image that contains a minimal Gentoo Linux installation and all dependencies necessary to conduct experiments. The user can extend this image, if additional dependencies are required for the experiment at hand. All that has to be done is to unpack the base image and invoke the Gentoo package manager

[5] apart from the standard utilities `newuidmap` and `newgidmap` (which are installed with setuid rights) when more than one user ID is needed in the container.

[6] https://www.gentoo.org/

Portage for the given project. This compiles the project and all its' dependences inside the uchroot environment with our customized compiler extension.

BenchBuild offers a single template to compile any of 18978 Gentoo packages. Listing 3 shows a simplified version of project `AutoPortage`. The resulting list of dynamically generated projects can be filtered further by arbitrary properties such as the source code language used, e.g., "only select projects written in C/C++".

```
class AutoPortage(GentooGroup):
  def build(self):
    with local.cwd(self.builddir):
      emerge = uchroot()["/usr/bin/emerge"]
      unmask = emerge["--autounmask-only=y",
      ↪  "--autounmask-write=y"]
      atom = package_name(self.DOMAIN, self.NAME)
      with local.env(CONFIG_PROTECT="-*"):
        unmask(atom, retcode=None)
      run(emerge[atom])
```

Listing 3: The template for the compilation of Gentoo projects. Gentoo's package manager Portage is invoked twice. First, all necessary configuration changes required for a successful package installation are applied. Second, the actual installation is carried out.

4. WORKFLOW

This section illustrates the typical workflow of BenchBuild by an example of conducting a new experiment that evaluates the applicability and performance of PolyJIT, our own just-in-time compiler. PolyJIT provides a clang plugin for the generation of compile-time statistics on a potential run-time applicability. The preparation of the experiment consists of three steps: (1) select suitable candidates for run-time evaluation by running a compile-time experiment, (2) generate a set of representative run-time tests for each project, (3) conduct the run-time experiment for all selected candidates. Step 3 is reduced to project bzip2 with a subset of its default run-time tests, which is derived from the reference input given by the SPEC benchmark suite.

A new compile-time experiment.

As stated previously, the selection of suitable projects from all available projects is guided by a new compile-time experiment.

Listing 4 shows a (simplified) experiment skeleton that executes the following actions in order: `MakeBuildDir`, `Prepare`, `Download`, `Configure`, `Build`, and `Clean`. Each project requires an experiment-specific configuration returned by `custom_cflags()` and `custom_ldflags()` which, in the example given, are the settings required for PolyJIT.

The compile-time experiment is completed with the definition of the compiler extension `collect_compilestats(*args)` in Listing 5. The existing compiler command `clang` is extended to output additional compilation statistics before its invocation `run(clang, project, experiment.name)`. The output is then postprocessed (details encapsulated in call `get_compilestats(...)`) and the final values of all statistic variables are stored (`store_compilestats(...)`).

```
from pprof.experiment import Experiment
def collect_compilestats(*args): pass
def custom_cflags(): pass
def custom_ldflags(): pass

class CompileStats(Experiment):
  def configure(self, p):
    p.cflags = project.cflags + custom_cflags()
    p.ldflags = project.ldflags + custom_ldflags()
    p.compiler_extension = collect_compilestats
    return p

  def actions_for_project(self, p):
    p = self.configure(p)
    return [MakeBuildDir(p),
            Prepare(p),
            Download(p),
            Configure(p),
            Build(p),
            Clean(p)]
```

Listing 4: Compile-time experiment skeleton that compiles all projects of BenchBuild with customized compilation configuration. The implementation of structurally unimportant methods is omitted.

```
from pprof.utils.run import guarded_execution
from pprof.utils.db import persist_compilestats

def get_compilestats(*args): pass

def collect_compilestats(
    project, experiment, config, clang, **kwargs):
  clang = clang["-mllvm", "-stats"]
  with guarded_execution() as run:
    run_result = run(
        clang, project, experiment.name)

  if run_result == 0:
    stats = get_compilestats(run_result.stderr)
    persist_compilestats(run_result, stats)
```

Listing 5: Compiler extension for the generation and collection of all compilation statistics provided by clang. BenchBuild stores the serialized version next to the wrapped clang compiler, which then invokes the compiler extension after successful compilation.

A run of the experiment reveals that, of the 1558 projects available in BenchBuild that are supported by PolyJIT, 284 projects are suitable for PolyJIT.

The compile-time experiment `CompileStats` identifies Gentoo packages suitable for run-time performance evaluation with PolyJIT. For each package, one project must be instantiated from the `AutoPortage` template, which already provides a project definition that lacks only the definition of run-time tests. An example definition for bzip2 is shown in Listing 6. The binary to be tested (`bzip2_path`) is wrapped with the run-time extension function (`rt_extension`) and a new command is generated that calls the wrapped binary (`"/bin/bzip2"`) inside a user-change-root environment. The

```
def run_tests(self, rt_extension):
  from pprof.run import uchroot
  from pprof.project import wrap

  bzip2_path = self.builddir + "/bin/bzip2"
  wrap(bzip2_path, experiment, self.builddir)
  bzip2 = uchroot()["/bin/bzip2"]

  opts = ["-f", "-k"]
  run(bzip2[opts, "-z", "-9", "text.html"])
  run(bzip2[opts, "-d", "text.html.bz2"])
```

Listing 6: Run-time test method for GENTOO project BZIP2. This method extends the generic `AutoPortage` class found in Listing 3. The preparation of the input files for compression/decompression is omitted for brevity.

actual measurement compresses and decompresses one file using the generated command. Such a run-time test must be defined for each selected project candidate.

The run-time experiment.

The run-time experiment completes the definition of our example. We have chosen to extend the already existing compile-time experiment `CompileStats` (an independent experiment would be another option). The run-time extension shown in Listing 7 performs three tasks: (1) create a new command (`cmd`) that wraps the original binary command (passed in as function argument `rt_call`) with the binary `time`, (2) invoke the measurement and capture its output, (3) postprocess the output and store all extracted measurement results.

5. FUTURE WORK

The research prototype BENCHBUILD comes with 18978 projects for compile-time tests via its GENTOO project template `AutoPortage`. One interesting question answered easily with BENCHBUILD is how the GENTOO and the DEBIAN versions of packages like, e.g., BZIP2 compare.

It is often of interest how a project's experimental results evolve between different versions. BENCHBUILD is able to conduct such experiments conceptually, but offers no integrated support as of yet.

With SLURM support, BENCHBUILD has access to a resource manager that is capable of guaranteeing resource constraints for running experiments. However, beside resource management, SLURM is also targeted at large parallel compute clusters. Integrating support for the LINUX CGROUP subsystem in our UCHROOT tool can provide a lightweight alternative to SLURM on lower-performance processing environments.

6. CONCLUSIONS

We propose BENCHBUILD, our tool for compile-time and run-time testing of arbitrary case studies. Conducting compile-time and run-time experiments successfully involves tedious and error-prone tasks such as the substitution of the compiler used by a project's build system, or the wrapping of binaries with hand-written scripts. With BENCHBUILD's tools for wrapping any binary with customized measurements, all steps other than the definition of the experiment and the selection

```
def run_with_time(project, experiment, rt_call):
  from pprof.utils.db import persist_time
  from pprof.utils.run import guarded_execution
  from plumbum.cmd import time

  def get_time_output(*args): pass

  cmd = time["-f", "%U-%S-%e", rt_call]
  with guarded_execution() as run:
    run_result = run(
      cmd, project, experiment.name)

  timings = get_time_output(
    "{:g}-{:g}-{:g}", stderr)
  persist_time(run_result, timings)
```

Listing 7: Test function to measure all BENCHBUILD projects that support run-time tests with `time`.

of representative run-time test inputs can be performed automatically. With 18978 projects available for compile-time testing and 188 projects available for both compile-time and run-time testing, potential users have an effective device for running large case studies without burden.

7. ACKNOWLEDGEMENTS

Financial support was received from the German Research Foundation (DFG), project POLYJIT, grant no. LE 912/14.

8. REFERENCES

[1] P. D. O. Castro, C. Akel, E. Petit, M. Popov, and W. Jalby. CERE: LLVM-based Codelet Extractor and REplayer for piecewise benchmarking and optimization. *ACM Trans. Architectural Code Optimimization (TACO)*, 12(1):6:1–6:24, Apr. 2015.

[2] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Software Engineering Methodology (TOSEM)*, 25(1):7:1–7:34, Dec. 2015.

[3] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.*, (239):76–91, Mar. 2014.

[4] A. Simbürger, S. Apel, A. Größlinger, and C. Lengauer. The potential of polyhedral optimization: An empirical study. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 508–518. IEEE Computer Society, Nov. 2013.

[5] C. D. Spradling. SPEC CPU2006 benchmark tools. *SIGARCH Computer Architecture News*, 35(1):130–134, Mar. 2007.

[6] G. K. Thiruvathukal. Gentoo Linux: The next generation of Linux. *Computing in Science & Engineering*, 6(5):66–74, Sept. 2004.

[7] A. B. Yoo, M. A. Jette, and M. Grondona. SLURM: Simple Linux utility for resource management. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing (JSSPP)*, LNCS 2862, pages 44–60. Springer, 2003.