2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

FEX: A Software Systems Evaluator

Oleksii Oleksenko† Dmitrii Kuvaiskii† Pramod Bhatotia‡ Christof Fetzer†
† Technical University of Dresden ‡ The University of Edinburgh

Abstract—**Software systems research relies on experimental evaluation to assess the effectiveness of newly developed solutions. However, the existing evaluation frameworks are rigid (do not allow creation of new experiments), often simplistic (may not reveal issues that appear in real-world applications), and can be inconsistent (do not guarantee reproducibility of experiments across platforms).**

This paper presents FEX, a software systems evaluation framework that addresses these limitations. FEX is extensible (can be easily extended with custom experiment types), practical (supports composition of different benchmark suites and real-world applications), and reproducible (it is built on container technology to guarantee the same software stack across platforms). We show that FEX achieves these design goals with minimal end-user effort—for instance, adding Nginx web-server to evaluation requires only 160 LoC. Going forward, we discuss the architecture of the framework, explain its interface, show common usage scenarios, and evaluate the efforts for writing various custom extensions.

I. INTRODUCTION

Software systems research primarily relies on experimental evaluation to validate the effectiveness of proposed solutions. Therefore, a sound experimental evaluation setup is at the core of systems research.

At the same time, evaluating new systems can be tedious, time-consuming, and error-prone. Furthermore, since systems research usually requires multiple iterations over the "design-implement-evaluate" cycle, the resulting evaluation effort can be significant. Ideally, a sound evaluation mechanism requires a wide variety of benchmarks to be built with varying parameters, run in a controlled "bias-free" fashion, and the results have to be aggregated, processed, and neatly plotted.

Unfortunately, there is no unifying evaluation framework which could be reused and extended in new projects. Current best practice lies in taking a benchmark suite, e.g., SPEC CPU2006, modifying its configuration files with custom parameters, and writing a number of scripts to automate experiment runs, aggregate results, and finally plot them. This ad-hoc method has three major limitations.

First, existing benchmark suites are *rigid*. They cannot be easily combined together and it is either cumbersome or outright impossible to modify experiments, e.g., add security evaluation to the existing suite. We experienced these problems during a project that evaluated a new security-related tool [1]. For the evaluation to be holistic, we used three benchmark suites: SPEC CPU2006 [2] (de-facto standard but supports only single-threaded applications), Phoenix [3] (represents I/O- and memory-intensive workloads), and PARSEC [4] (contains complex multithreaded programs). Each has its own management system and no support for plotting. Without a unifying framework, we would be forced to replicate the same configuration parameters in ad-hoc scripts, possibly resulting in hard-to-diagnose performance bugs. Additionally, our security evaluation would require yet another collection of scripts.

The second limitation of existing benchmark suites is that they are often *simplistic*. Recent works show that using only one benchmark suite may be insufficient for adequate evaluation [5]. Moreover, choosing a wrong suite may significantly skew the results, either because the included benchmarks do not represent current computing problems [6, 7] or because they do not fully capture the specifics of a particular domain [8].

The third limitation of the existing evaluation frameworks is that they can be *inconsistent*: they do not enforce the same software stack (in particular, specific versions and build flags of used compilers, libraries, and tools) which may lead to inconsistent results across different platforms. This aspect is crucial for reproducibility of results [9]. While other areas of computer science have developed domain-specific solutions to this problem [10, 11], we are unaware of similar tools in the systems community.

This paper presents FEX, a software systems evaluation framework that overcomes the aforementioned limitations. FEX is *extensible* and *practical*—adding a new type of experiment, a benchmark suite, or a standalone application requires minimal effort. Furthermore, it leverages the Docker container technology to secure the software stack and achieve *reproducibility* [12, 13]. To our knowledge, FEX is the first effort to combine the entire build-run-plot evaluation process across different benchmark suites and standalone programs. It can be used to evaluate compiler extensions and optimizations, dynamic and static instrumentation tools, new libraries or library versions, and any other tools that affect the application behavior.

Out-of-the-box, FEX is integrated with several well-known benchmark suites (SPLASH, Phoenix, PARSEC), standalone programs (Apache, Memcached, Nginx), compilers (GCC, Clang), and measurement tools (perf, time). Internally, our framework is comprised of a number of tools for automating installation, building and running benchmarks, collecting logs, and plotting the final results.

To highlight FEX's extensibility and ease of use, we evaluate the efforts to incorporate SPLASH-3 benchmark suite [14], Nginx web-server [15], and RIPE security testbed [16]: 326, 166, and 75 LoC respectively. In terms of time spent, the whole effort took less than 8 man-hours. These results allow us to conclude that FEX significantly simplifies the software systems evaluation process.

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.25

II. Design of Fex

A. System Interface

Fex was developed with reproducibility as one of the main goals. Therefore, we prepare the environment and run all experiments in a Docker container in such a way that they are as independent from the actual host system as possible [12, 13]. The Docker image contains a bare minimum to run the experiments: sources for all programs in benchmark suites with corresponding makefiles, Bash scripts for environment setup, Python scripts to actually perform experiments and to aggregate and plot their results. Note that the packages put in the image—git, python3, wget, perf, etc—are used by the framework itself and do *not* influence the experiments (i.e., they do not affect measurements neither through the build system nor through dynamically linked libraries).

Figure 1 shows the general workflow and the exposed system interface of Fex. Any of the actions in the workflow can be executed via call to the framework's entry point—`fex.py` file:

```
>> fex.py <action> -n <name> [other_arguments]
```

For example, running Phoenix benchmark suite with GCC will look like this:

```
>> fex.py run -n phoenix -t gcc_native
```

The workflow is divided into two stages: setup and run. The first stage prepares an environment for the second stage by installing all the necessary components from the Internet.

Experiment Setup. The Docker image we ship contains only the source codes of benchmarks and a set of scripts to build and run them. The actual dependencies—compilers to build, shared libraries to link against, additional tools and benchmarks—are downloaded from the Internet and installed at the experiment setup stage. The reasons for this flow are twofold. First, the Docker image would swell to approx. 17GB in size[1] if all dependencies would be built-in. Images of such size would be cumbersome to distribute. Second, this allows the end user to install only those dependencies and only those versions needed for her experiments. (For reproducibility, it is important that the exact versions of software crucial for experiments are installed.)

For simplicity, the installation scripts are written in Bash (they can also be written in Python). To run an installation script, Fex provides an "install" command. The following example installs GCC 6.1.

```
>> fex.py install -n gcc-6.1
```

As shown in Figure 1 (top), this stage includes three steps:

- *Installing compilers* with specific versions is a prerequisite. Fex cannot rely on Linux default package managers to automatically install required compilers, e.g., APT or RPM, because compiler versions in their repositories change over time and thus hinder reproducibility.
- *Installing dependencies* implies tools required for the build process or for specialized measurements. For example, several PARSEC benchmarks require `gettext` system for Autoconf—this software does not affect performance but is simply needed to resolve all build dependencies of a particular benchmark. These tools are optional and may not be needed for simple experiments.
- *Installing additional benchmarks* may be necessary to perform experiments on large unmodified programs. Fex encourages to put program sources in its repository; this simplifies changing and tweaking original code to the user's needs. However, sometimes it is simpler to fetch the sources from elsewhere. For example, we install Apache and Nginx in this way because we want to experiment with their different versions (those that are vulnerable to a particular bug and those that are not).

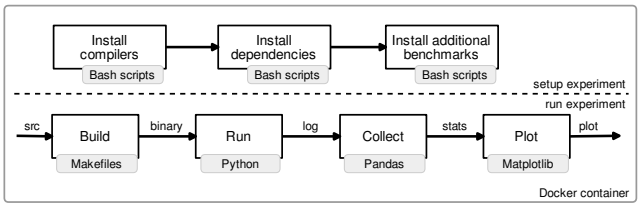

Fig. 1: System interface of Fex.

Experiment Runs. After installing all prerequisites, users can start running experiments. All experiments are usually performed as a sequence of steps depicted in Figure 1 (bottom):

- The *build* step is performed once before running each benchmark in the experiment. Fex consults the makefile corresponding to the benchmark-to-run and puts a final binary in the `build` directory. It is important to re-build all benchmarks for each experiment, otherwise a mix of old and new compilation flags and/or libraries could skew the results. For quick preliminary experiments, the build step can be omitted via `--no-build` flag.
- The *run* step is the experiment itself. Fex includes a Run component, which provides several Python hooks to specify a list of benchmarks-to-run with their inputs and to control how exactly these benchmarks are started. For example, we implement an additional "dry run" for Phoenix benchmarks using a `per_benchmark_action` hook. Multithreaded benchmarks are automatically run with a set of number of threads specified in the command line, e.g., `-m 1 2 4`.
- The *collect* step parses the log, extracts the measurement results, processes them in a user-specified way, and stores into a CSV table. We use Pandas Python library [17] for efficient data analysis and aggregation.
- The *plot* step is performed after the whole experiment is finished. It is usually performed on a local user machine and a the remote server. The powerful matplotlib [18] is used to emit different kinds of

[1] Our current image is 1.04GB, with 122MB Ubuntu files, 300MB of benchmarks' source files, and the rest helper packages

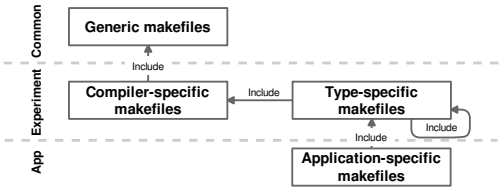

Fig. 2: Build system hierarchy.

plots. Again, the user is provided with hooks to change the appearance of emitted plots.

Building and running the benchmarks is sensitive to environment variables. FEX provides a convenience wrapper to specify default variables for these steps (see §II-B).

B. System Architecture

Following the workflow presented in §II-A, FEX consists of 4 subsystems: building, running, collecting, and plotting. The later two have plain structure which does not require any explanations. The former two, however, are slightly more complex and we will discuss them in detail.

Build subsystem. In the build stage, two scenarios are possible: (1) a single application can be built many times with varying build parameters or (2) the same parameters can be reused for many different applications. To aid this variability, our build subsystem was divided into three layers (see Figure 2): common, experiment, and application layers.

Common layer contains parameters that are applicable to all benchmarks and all build types. This includes, for example, optimization levels, debugging information (if enabled), common compilation flags and generic compilation targets.

Experiment layer is responsible for parameters of the current build type. For example, if a benchmark suite has to be built by GCC and with enabled AddressSanitizer, the makefiles will set `CC` variable to `gcc` and `CFLAGS` to `-fsanitize=address`. Note that there might be multiple levels of makefiles in this layer: some may set parameters that are applicable to all configurations of a single compiler, and the others will refine them to a concrete configuration.

Finally, the *application* layer defines the structure and the procedure of the build. It specifies the location of source files, lists dependencies, and sets application-specific flags.

The overall build system is structured in such a way that these layers can be replaced independently of each other. Accordingly, any application can be compiled with any of the existing build configurations without additional efforts.

Experiment runners. When an experiment is started via

```
>> fex.py run ...
```

a new instance of the FEX class is created (see Figure 3). This object controls the overall experiment execution. Firstly, it retrieves a configuration file and sets experiment parameters accordingly. Then, it sets environment variables to the necessary values by instantiating child classes of the Environment abstract class. In the end, it instantiates and calls the child of the Runner class that corresponds to the current experiment. This new Runner object will perform the actual experiment.

Since environmental variables can vary in accordance to parameters of the given experiment (e.g., when debug mode is turned on), we define four types of the variables:

1) Default: the default values of environment variables.
2) Updated: the values of this type are appended if the variable exists, and assigned otherwise.
3) Forced: the variables are overwritten regardless of the previous value.
4) Debug: the values are set only in the debug mode.

Note that the order is important here: each next type has higher priority that the previous one. For example, if variable `BIN_PATH` is assigned to `/usr/bin/` among default variables and to `/home/usr/bin/` among the forced ones, the end value will be `/home/usr/bin/`. On top of it, if a user wants to add another type, she can do it simply by writing a subclass of Environment and redefining the `set_variables` function.

The key element of the Runner class is `experiment_loop` function. For each of the execution parameters, it iterates over all their values by going through a series of nested loops, as shown in Figure 4. For example, the outermost loop may go through GCC and Clang compilers, the next one—through Nginx, Apache, and Memcached applications, and so forth. Each of the loops has a hook that can be implemented in a subclass. This way, the overall structure of the experiment stays the same, but the concrete actions can be tailored to the needs of the given experiment. Moreover, if even more parameters would be necessary, the `experiment_loop` can be redefined or extended in a subclass, as VariableInputRunner does in Figure 4.

III. FEX DETAILS AND WORKFLOW

A. Creating new experiments

In this section, we explain how FEX facilitates creation of new experiments and evaluation of new benchmark suites and standalone programs. We also detail the implementation of FEX with the help of its standard directory layout (similar to projects like Jekyll [19], FEX assumes a specific directory tree structure).

One (simplified) example of a directory tree is shown in Figure 5. Here, the end user sets up the environment to evaluate the performance overhead of Google's AddressSanitizer [20] on the Phoenix benchmark suite [3] and on Apache web server [21]. Moreover, for reproducibility she chooses GCC version 6.1 which comes with AddressSanitizer by default.

First, she needs to write installation scripts to install the GCC 6.1 compiler, download input files for the Phoenix benchmark, and install an additional Apache benchmark. For convenience, FEX provides a set of functions for frequently used operations, found in `install/common.sh`, e.g., `download`.

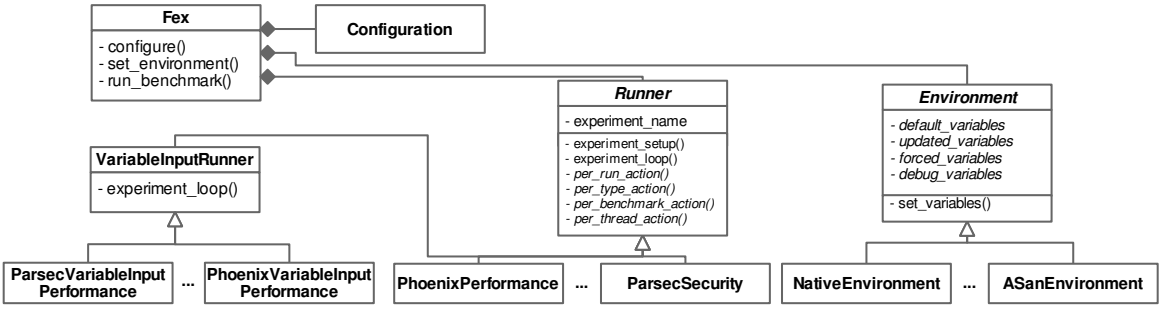

Fig. 3: Class diagram of the infrastructure for running experiments.

```
for each build type:
  self.per_type_action(type)
  for each benchmark:
    self.per_benchmark_action(type, benchmark)
    for each thread count:
      self.per_thread_action(type, benchmark, thread_num)
      for i in range(0, number_of_repetitions):
        self.per_run_action(i)
```

Fig. 4: Experiment loop

Next, the user must create compiler-specific and type-specific makefiles for different experiment variants and put them under `makefiles/`. The compiler-specific `gcc_native.mk` file would look like:

```
include common.mk
CC := gcc
CXX := g++
```

The type-specific file `gcc_asan.mk` would include the previous file and additionally enable AddressSanitizer:

```
include gcc_native.mk
CFLAGS += -fsanitize=address
LDFLAGS += -fsanitize=address
```

After that, the user must put application-specific makefiles and sources of the Phoenix benchmark suite and Apache web server under `src/`. To keep directories clean, standalone programs like Apache are put in a separate subdirectory named `applications/`. In case of Apache, the sources are downloaded from the Internet using an installation script, thus the only file required is a Makefile. In case of Phoenix, all sources are copied in the FEX directory tree for convenience; we only show `histogram` for the sake of clarity. Application-specific makefiles are generally simple and follow this pattern on the `histogram` example:

```
NAME := histogram
SRC := histogram-pthread
include Makefile.$(BUILD_TYPE)      ;; includes type-specific makefile
all: $(BUILD)/$(NAME)               ;; build target
```

Finally, the user describes the experiments themselves. The Phoenix performance-overhead experiment is put under `experiments/phoenix`. The only required file here is `run.py` which describes benchmarks with their command-line arguments to be run and measured. Additionally, each Phoenix benchmark needs a preliminary dry run: this functionality is implemented through a `per_benchmark_action` hook. Note that most of the functionality to build and run benchmarks is actually inherited from the abstract class implemented in `experiments/run.py`.

There are no specific collect and plot scripts for Phoenix. Instead, since the user has no ad-hoc requirements for them, the generic `collect.py` and `plot.py` are re-used. Also note that Figure 5 does not show Apache experiment files for simplicity.

Some variants of the experiment may require setting specific environment variables. For example, AddressSanitizer can be fine-tuned via runtime flags in the `ASAN_OPTIONS` variable. For this, the user shall add this flag in `environment.py`. In addition, the user can modify parameters for collection and plotting of results in a `config.py` file.

In the end, to add a new experiment with new compiler types and new benchmarks, the user needs to create (some of) the following files: (1) installation scripts, (2) compiler- and type-specific makefiles, (3) sources and makefiles for benchmarks, and (4) experiment descriptions. In fact, all of the scripts are already available in the repository of FEX.

Additionally, FEX facilitates creation of tests—short runs of benchmarks with tiny inputs. These tests can be accessed via `-i test` and are useful to check if user-defined makefiles, source files, and other scripts are written correctly. This functionality is implemented inside `run.py` files.

B. Running new experiments

Now that the experiment description is finished, the user can re-build the Docker container using `Dockerfile` and deploy it on a test server. The actual experiment proceeds in two stages as shown in Figure 1.

Inside the container, the user sets up the experiment by invoking the relevant installation scripts:

```
>> fex.py install -n gcc-6.1
>> fex.py install -n phoenix_inputs
>> fex.py install -n apache
```

Next, it is sufficient to call the generic all-in-one "run" command like this:

```
>> fex.py run -n phoenix -t gcc_native gcc_asan
```

This command will build all Phoenix benchmarks using native and AddressSanitizer GCC versions, run them once (with a preliminary dry run), collect statistics from logs, and aggregate and save final data in a CSV table. There are several command-line flags to fine-tune the

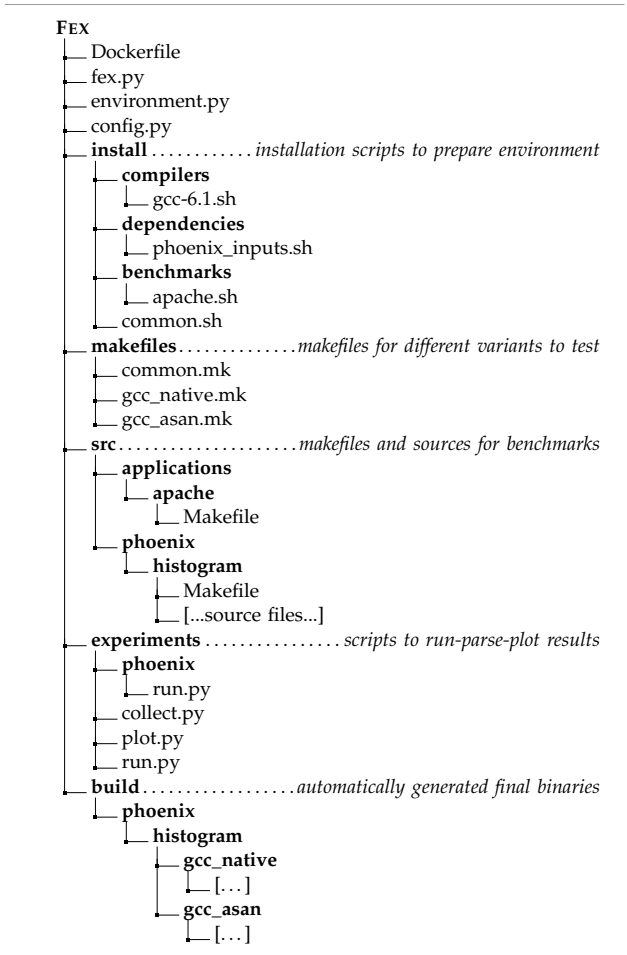

Fig. 5: Example directory tree of FEX.

experiment. The user can specify `-v` for verbose output and `-d` to build debug versions of benchmarks. To increase the number of runs of each benchmark, the user adds `-r 10`. To run benchmarks with different numbers of threads, the `-m 1 2 4` flag must be added. To run only one benchmark from the benchmark suite, the user specifies `-b histogram`.

Note that the final binaries of benchmarks are put under `build/` directory, see Figure 5. Sometimes it is useful to run the binary directly from there, e.g., to debug spurious errors or to perform additional quick measurements.

After the experiment is finished, the user should fetch the final CSV results from the server and run the "plot" command locally:

```
>> fex.py plot -n phoenix -t perf
```

This builds the "perf" (performance overhead barplot) graph and saves it in a PDF file. The examples of such graphs are shown in the next section.

C. Currently supported experiments

During the months of internal use of FEX, we expanded it in several directions. Table I lists the currently supported benchmarks, compilers and compilation types, and experiments.

– Benchmark suites	Phoenix, SPLASH, PARSEC, SPEC CPU2006*
– Add. benchmarks	Apache, Nginx, Memcached, RIPE, micro
– Compilers	GCC, Clang/LLVM
– Types	AddressSanitizer (as example)
– Experiments	Performance and memory overheads, security evaluation
– Tools	perf-stat (generic), perf-stat (memory), time
– Plots	Lineplot, regular barplot, stacked barplot, grouped barplot, stacked-grouped barplot

*Will not be open-sourced as part of FEX due to proprietary license.

TABLE I: Currently supported experiments in FEX.

FEX supports four benchmark suites: Phoenix [3], SPLASH [14], PARSEC [4], and SPEC CPU2006[2] [2]. Additionally, FEX comes with Apache, Memcached, and Nginx programs that showcase real-world usage scenarios. They are installed via scripts and *not* put under `src/`. FEX also provides several statically linked libraries like libevent and OpenSSL, required for at least one of the above benchmarks. Lastly, we wrote a suite of microbenchmarks—e.g., reading from an array—that can be useful for debugging purposes.

FEX provides installation scripts and makefiles for GCC version 6.1 and Clang/LLVM 3.8.0 [22]. It is easy to update these scripts to install newer versions of these compilers. As of type-specific makefiles, the current version of the framework includes only AddressSanitizer as an example.

The list of supported experiments includes (1) performance- and memory-overhead experiments as well as variable-inputs experiments of Phoenix, PARSEC, and SPEC, and (2) throughput-latency and security experiments of Apache, Nginx, and Memcached.

For plotting, FEX provides the following generic plots: barplot (e.g., for performance and memory overheads), lineplot (for multithreading overheads), stacked barplot, grouped barplot, and stacked-and-grouped barplot (for complicated statistics such as cache misses at different levels).

IV. CASE STUDIES

In this section, we evaluate extensibility and ease of use of FEX on a set of benchmarks. To showcase the required end-user effort, we considered the following scenario: a researcher wants to compare performance of Clang compiler against GCC using SPLASH-3 [14] benchmark suite and Nginx web server [15], as well as the security guarantees provided by the two compilers using the RIPE testbed [16].

The presented evaluation of effort was done by an experienced user—we wrote all the extensions on our own.

A. Multithreaded Benchmark Suite: SPLASH-3

SPLASH-3 benchmark suite is used to evaluate parallel applications on large-scale NUMA architectures

[2]SPEC CPU cannot be made publicly available and will not be open-sourced as part of FEX

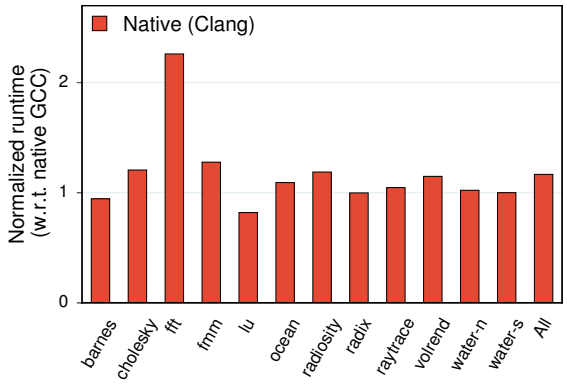

Fig. 6: Example of Clang-GCC comparison produced by FEX and tested on SPLASH-3.

[14]. To include it in FEX, the first step was to add the source code of SPLASH-3 itself. This required:

- Changes in the build system of the suite: renaming of the variables, restructuring of directories, and removing unnecessary build targets—194 LoC in total. Note that most of the changes can be done by automatic renaming and do not take much time. Also, the resulting build system became smaller and more generic, since many variables and build targets are now defined globally (167 LoC deleted).
- An installation script to download input files (5 LoC).
- A Runner subclass to control the experiment (36 LoC) and `collect.py` script to process the final results (9 LoC).

Next, Clang must be added as a build type (GCC is supplied with the framework). It required writing an installation script for Clang and all its dependencies (50 LoC if built from sources) and a compiler-specific Makefile (6 LoC).

Finally, the results had to be represented as a plot of slowdown (speedup) of Clang versions over GCC ones. The natural choice for this type of data is a barplot since it can clearly depict overheads of several applications. FEX already has the functionality for building such plots, therefore the effort was minimal—only 26 LoC in `plot.py` script. The total effort summed up to 326 LoC or approximately 5 man-hours of work.

The experiment was run with the following command:

```
>> fex.py run -n splash -t gcc_native clang_native
```

As a result, it produced the plot shown in Figure 6. From this plot the researcher might deduct, for example, that the given version of Clang has slightly worse performance than GCC and it is especially bad with operations on matrices, as represented by FFT.

B. Real-world Application: Nginx Web-server

To evaluate the effort of adding a standalone application, we integrated Nginx web server into FEX [15].

First, we did not put Nginx's sources under the `src/` directory, but instead wrote an installation script (9 LoC). Next, we created a performance experiment: we wrote a specialized `collect.py` script to collect

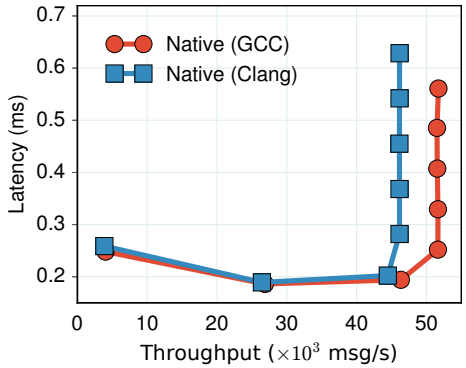

Fig. 7: Example throughput-latency plot of Nginx produced by FEX. Remote clients fetch a 2K static web-page over a 1Gb network.

throughput and latency statistics (14 LoC), a `plot.py` script to adjust the appearance of a throughput-latency plot (34 LoC), and a `run.py` script to pre-configure the server side, start a client on a separate machine via SSH, wait for the experiment to finish, and fetch the logs (89 LoC). Finally, we created a makefile with configuration options to build Nginx (20 LoC). The whole effort was 166 LoC—mostly due to a complicated running scenario with a remote client—or approximately two man-hours.

We ran the Nginx experiment like this:

```
>> fex.py run -n nginx -t gcc_native clang_native
```

The resulting throughput-latency measurements are shown in Figure 7. In our hypothetical study this plot would support the previous observations: the Clang version has worse throughput than GCC.

C. Security Benchmark: RIPE

To highlight that FEX supports types of experiments other than performance ones, we experimented with RIPE security testbed [16]. At its core, RIPE is a C program that tries to attack itself in a variety of ways (with 850 possible attacks in total).

As a first step, we put sources of RIPE—two source and two header files—together with a simple Makefile under `src/`. We did not change the source code of RIPE, and our resultant Makefile was 14 LoC. Next, we created an experiment. The `run.py` script (44 LoC) simply calls a script to run security tests, shipped together with RIPE. The `collect.py` script extracts RIPE-specific statistics from the final log (17 LoC). Note that for this security experiment, we do not need any plot. In the end, the effort took 75 LoC and less than one hour of work.

The experiment was run with the following command:

```
>> fex.py run -n ripe -t gcc_native clang_native
```

This produced the aggregated results shown in Table II. It is interesting to note that even under our "insecure" configuration (Ubuntu 16.04 with disabled ASLR and building with disabled stack canaries and enabled executable stack), only a handful of attacks were successful: through the shellcode that creates a dummy file and through return-into-libc. Another interesting result is that Clang

Compiler	Successful	Failed
Native (GCC)	64	786
Native (Clang)	38	812

TABLE II: RIPE security benchmark results produced by FEX. Columns 2 and 3 show the number of successful and failed attacks respectively.

has almost 2× less successful attacks: Clang prevents indirect attacks via buffers in BSS and Data segments due to a smarter layout of objects in these segments.

V. RELATED WORK

Benchmark suites. Systems researchers developed a plethora of benchmark suites, varying in their age, targeting, and diversity. Dhrystone is a 30-year-old but still wildly used set of synthetic integer benchmarks [23]. Because of its age and code atypical for modern programs, its substitute Coremark suite was developed in 2009 [24]. However, both these suites are targeted for embedded systems and have a limited diversity of included programs. Recently, new benchmark suites were released, covering more scenarios and stressing particular parts of systems (floating-point operations, instruction/data cache pressure, etc.) [2, 14, 25, 26]. For example, MiBench is a comprehensive set of 35 embedded applications targeting areas such as networking, security, automotive, and telecommunications [27].

Aside from the high number of included programs and their diversity, benchmark suites are characterized by their targeting. LINPACK [28] is targeted for vectorizable computations, MediaBench [29]—for media applications, BioPerf [30]—for bioinformatics, MineBench [31]—for data mining area, HPC Challenge [32] and NAS [33]—for high-performance computing.

The focus of this work are benchmarks to analyze the impact of static and dynamic instrumentation techniques. Thus, benchmarks that test the whole hardware/software stack (Phoronix [34]) or large-scale systems (YCSB [35] and CloudSuite [8]) are not in the scope of FEX.

Orthogonally to our work, recent research efforts concentrate on evaluation of diversity and redundancy of benchmark suites. For example, several studies analyzed the redundancy of SPEC and PARSEC benchmark suites, i.e., what is the minimum number of programs and inputs needed to obtain statistically significant results [36, 37]. Some papers compare different benchmark suites, e.g., PARSEC and SPLASH [6, 38].

Tools and methodologies for performance measurements. To our knowledge, FEX is the only meta-framework to combine several benchmark suites. However, other research tools exist that strive to provide more stable, reproducible, and statistically significant results of performance evaluations.

It is widely known that *measurement bias* can perturb evaluation and lead to incorrect performance results [5]. Additionally, abnormal behavior called *workload flurries* is frequently observed in real workloads (which are later used as inputs in benchmarks) and can thus lead to instable results [39]. One solution to this problem is "shaking" the input workload to achieve a better distribution of results [40]. We believe this can be seamlessly integrated in FEX.

Stabilizer is a tool to achieve statistically sound performance evaluation by re-randomizing the memory layout—which turns out to be the leading factor of measurement bias—to achieve normal distribution of results [41]. Coz is another tool for better performance measurements [42]. Its main focus is on highlighting performance bottlenecks in complex software with the help of *causal profiling*, by virtually speeding up separate parts of code. Stabilizer was evaluated only on SPEC CPU2006, and Coz—only on PARSEC. Both tools could benefit from FEX: they could be plugged into FEX for quick evaluation on other benchmark suites.

Kalibera and Jones provide not a tool but a set of guidelines to perform statistically sound experiments with the minimum number of repetitions for each benchmark [43]. The users are encouraged to follow these guidelines when performing their experiments with FEX.

Another related category of measurement tools is comprised of profilers [44, 45] and tracers [46, 47]. These tools are mainly used for measuring various runtime parameters and analyzing performance bottlenecks. Some of them (e.g., `perf` [44]) provide basic results processing, such as calculation of mean values and standard deviations over several runs. Yet, all of them are single-application single-configuration tools, i.e., they are not capable of aggregating and comparing measurement results of multiple benchmarks and/or different build configurations. Moreover, control of experiment procedure is out of their scope. Therefore, we consider Fex to be orthogonal and complementary to this class of tools.

VI. CONCLUSION AND FUTURE WORK

FEX started as a small set of frequently reused scripts and quickly matured in a full-fledged evaluation framework. During months of internal usage (e.g., in our evaluation of Intel MPX [48] and SGXBounds [1]), we constantly refactored and expanded the framework to accommodate our growing needs, until the point we decided it can be useful for others. The end result is FEX, a software system evaluation framework that is extensible, practical, and reproducible.

Currently, FEX has a number of limitations. The framework provides no statistical analysis functionality (except basic statistics such as standard deviation). We plan to integrate statistical numpy/scipy Python packages in the framework to allow for advanced statistical methods and hypothesis testing.

We would like to combine FEX with a continuous integration system (e.g., Jenkins) to facilitate Evaluation-Driven Development (similar to Test-Driven Development). Furthermore, we wish to support a graphic user interface, since an ability to observe intermediate results will simplify and shorten the process of setting up and debugging experiments.

FEX supports only single-machine experiments. We are investigating ways to build distributed experiments, e.g., using the Fabric library.

Finally, since we rely on the Docker infrastructure, we do not guarantee reproducibility on levels below user space, i.e., on different hardware setups or across different kernel versions. However, FEX outputs various environment details, so that the complete experimental setup is stored in the log file.

FEX is available at https://github.com/tudinfse/fex.

REFERENCES

[1] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBounds: Memory Safety for Shielded Execution," in *EuroSys*, 2017.

[2] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *HPCA*, 2007.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[5] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009.

[6] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors," in *WWC*, 2008.

[7] V. Saxena, Y. Sabharwal, and P. Bhatotia, "Performance evaluation and optimization of random memory access on multicores with high productivity," in *HiPC*, 2010.

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ASPLOS*, 2012.

[9] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, 2016.

[10] G. R. Brammer, R. W. Crosby, S. J. Matthews, and T. L. Williams, "Paper mâché: Creating dynamic reproducible science," *Procedia Computer Science*, 2011.

[11] S. Perianayagam, G. R. Andrews, and J. H. Hartman, "Rex: A toolset for reproducing software experiments," in *BIBM*, 2010.

[12] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014.

[13] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS OS Review*, 2015.

[14] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *ISPASS*, 2016.

[15] "nginx: The Architecture of Open Source Applications," http://www.aosabook.org/en/nginx.html, 2016, accessed: Oct, 2016.

[16] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *ACSAC*, 2011.

[17] W. McKinney, "pandas: a foundational python library for data analysis and statistics," *Python for High Performance and Scientific Computing*, 2011.

[18] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science Engineering*, 2007.

[19] Jekyll, "Directory structure – Jekyll," https://jekyllrb.com/docs/structure/, accessed: Dec, 2016.

[20] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *ATC*, 2012.

[21] "Apache HTTP server project," http://httpd.apache.org/, 2016, accessed: Oct, 2016.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004.

[23] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of ACM*, 1984.

[24] "EEMBC – CoreMark – Processor Benchmark," http://www.eembc.org/coremark/, 2016, accessed: Nov, 2016.

[25] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *MoBS*, 2009.

[26] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System," in *WWC*, 2009.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *WWC*, 2001.

[28] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, 2003.

[29] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *MICRO*, 1997.

[30] D. Bader, Y. Li, T. Li, and V. Sachdeva, "BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications," in *WWC*, 2005.

[31] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *WWC*, 2006.

[32] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) Benchmark Suite," in *SC*, 2006.

[33] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks—Summary and Preliminary Results," in *SC*, 1991.

[34] "Phoronix test suite," http://www.phoronix-test-suite.com/, 2016, accessed: Nov, 2016.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*, 2010.

[36] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ISCA*, 2007.

[37] C. Bienia and K. Li, "Fidelity and scaling of the PARSEC benchmark inputs," in *WWC*, 2010.

[38] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterization of SPLASH-2 and PARSEC," in *WWC*, 2009.

[39] D. Tsafrir and D. G. Feitelson, "Instability in parallel job scheduling simulation: The role of workload flurries," in *IPDPS*, 2006.

[40] D. Tsafrir, K. Ouaknine, and D. G. Feitelson, "Reducing performance evaluation sensitivity and variability by input shaking," in *MASCOTS*, 2007.

[41] C. Curtsinger and E. D. Berger, "Stabilizer: Statistically sound performance evaluation," in *ASPLOS*, 2013.

[42] C. Curtsinger and E. Berger, "Coz: Finding code that counts with causal profiling," in *SOSP*, 2015.

[43] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *ISMM*, 2013.

[44] "perf: Linux profiling with performance counters," https://perf.wiki.kernel.org, 2017, accessed: Apr, 2017.

[45] J. Reinders, *VTune performance analyzer essentials*. Intel Press, 2005.

[46] R. McDougall, J. Mauro, and B. Gregg, *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall PTR, 2006.

[47] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *OLS*, 2006.

[48] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches," *arXiv:1702.00719*, 2017.