Toward Well-Provenanced Computer System Benchmarking: An Update

Samuel K. Gutiérrez and Howard P. Pritchard
High-Performance Computing Division
Los Alamos National Laboratory
samuel@lanl.gov

September 24, 2021

Abstract

This report provides an update on the current design, implementation, and usage of *bueno*, a Python framework enabled by container technology that supports gradations of reproducibility for *well-provenanced benchmarking* of sequential and parallel programs. The ultimate goal of the bueno project is to provide convenient access to mechanisms that aid in the automated generation, collection, and dissemination of data relevant for experimental reproducibility in computer system benchmarking.

1 Introduction

Computer system benchmarking provides a means to compare or assess the performance of hardware or software against a point of reference. Because of some of the reasons discussed later in this report, achieving experimental reproducibility in this domain is challenging. To aid in this, we are developing an extensible software framework named *bueno* that helps support what we call *well-provenanced computer system benchmarking*, or *well-provenanced benchmarking* for short. In this context, a well-provenanced benchmark maintains to the extent possible the minimal required set of data needed to share, replicate, and revisit a prior result up to a given standard. To that end, the bueno project aims to provide convenient access to mechanisms that aid in the automated capturing of relevant *features* that define a sufficiently precise experiment. In the rest of this report, we provide additional motivation, followed by a summary of bueno's current software architecture, feature set, usage, and methodology toward this goal.

2 Motivation

Experimental reproducibility is a crucial component of the scientific process. Capturing the relevant features that define a sufficiently precise experiment is a difficult task. This difficulty is mostly due to the diversity and non-trivial interplay among computer platforms, system software, and programs of interest. To illustrate this claim, consider the interconnected relationships formed among the components shown in Figure 1. Here, we define an experiment as the Cartesian product of a given software stack and its configuration. The elements shown in Figure 1 are described as follows:

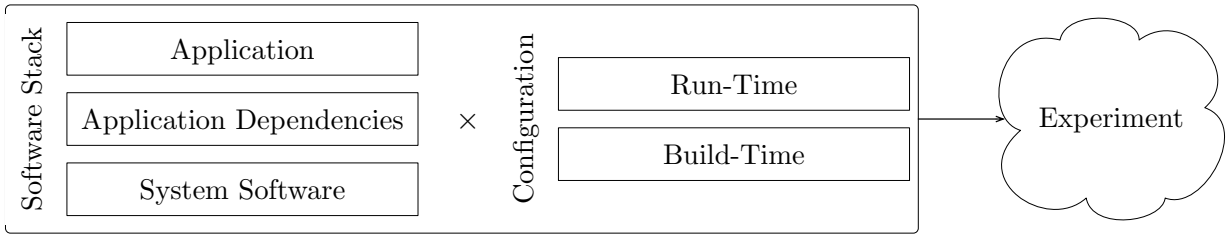

Figure 1: The high-level makeup of a computer system benchmarking experiment.

- **System Software**: the operating system (OS), compilers, middleware, runtimes, and services used by an application or its software dependencies. Examples include Linux, the GNU Compiler Collection (GCC), Message Passing Interface (MPI) libraries, and OpenMP.
- **Application Dependencies**: the software used by the application driver program, including linked software libraries and stand-alone executables. Examples include mathematical libraries, data analysis tools, and their respective software dependencies.
- **Application**: the driver program used to conduct a computer system benchmark, including sequential and parallel programs with and without external software dependencies. Examples include micro-benchmarks, proxy applications, and full applications.
- **Build-Time Configuration**: the collection of parameters used to build an application and its dependencies. This includes preprocessor, compile, and link directives that have an appreciable effect on the generated object files and resulting executables. Examples include whole program optimization (WPO) and link-time optimization (LTO) levels, which may vary across components in the software stack.
- **Run-Time Configuration**: the collection of parameters used at run-time that have an appreciable effect on the behavior of any software component used during a computer system benchmark. Examples include application inputs and environmental controls.

In summary, contemporary computing environments are complex. Experiments may have complicated software dependencies with non-trivial interactions, so capturing relevant experimental characteristics is burdensome without automation. Next, we describe the infrastructure that implements our graded approach toward solving some of the challenges noted previously in this report.

3 Software Overview

In this section, we begin with an overview of bueno's core software architecture. We then provide a brief introduction to container technology and motivate its use in bueno. Finally, we summarize bueno's command-line interface (CLI) and module services.

3.1 Core Framework

bueno is an open-source [1] (BSD-3) software framework written in type-annotated Python 3. Its internal software architecture is straightforward and organized into three major components: `core`, `public`,

[1] https://github.com/lanl/bueno

and `service`. The `core` component implements the infrastructure used internally within bueno. The `public` component, detailed in Section 3.4, implements a collection of Python modules made available to Python programs executed under bueno's supervision. We call these programs *bueno run scripts*, and they are the programmable interface that drives the bueno framework. Finally, the `service` component, detailed in Section 3.3, implements the infrastructure made accessible through a CLI that carries out a specific task, for example, executing a run script.

3.2 Use of Container Technology

Container technology [5] has garnered attention recently, especially in cloud and high-performance computing (HPC) environments. This attention is well-deserved, as this approach has demonstrated broad utility in software development and deployment tasks. In the context of container-enabled experimental reproducibility [1, 2], we note the following properties that serve our work's ultimate goal:

- **Data Encapsulation**: Containers offer nearly complete encapsulation of a given software stack [3]. This capability allows researchers the ability to revisit a prior experimental configuration. An example is conducting a post-mortem analysis of saved binary files to understand better the performance characteristics of a previous benchmarking result.
- **Low-Overhead Execution**: The use of a containerized software stack introduces little to no appreciable overhead compared to its non-containerized analog [3].
- **Separation of Concerns**: Container images can be built, shared, and later augmented to create a new *base image*, which in turn can be shared and augmented further. This process allows for a separation of concerns among a potentially large conglomerate of multidisciplinary expertise.

In summary, container technology is a promising avenue for capturing relevant features that define a sufficiently precise experiment. For this reason, bueno implements features that make use of containers to improve the likelihood of experimental reproducibility. Currently, bueno supports unprivileged container activation through Charliecloud [4].

3.3 Command Line Interface Services

CLI services are currently made available through bueno's `run` command. The `run` service coordinates container image activation and the execution of bueno run scripts, a programmatic description of the steps required to conduct a computer system benchmarking experiment. Currently, there are two image activators implemented in bueno: `charliecloud` and `none`. The former uses Charliecloud to activate a given container image. The latter is a pass-through to the host, which offers a lower degree of reproducibility when compared to the former.

3.4 Module Services

Because of the diversity among computer platforms, system software, and programs of interest, program execution and subsequent analysis of their generated outputs are expressed through Python programs executed by bueno's `run` service. A collection of Python utility modules is made available to these programs to aid in conducting benchmarking activities. Example functionality includes command dispatch to the host or container, logging, metadata asset agglomeration, concise expression of structured experimental inputs, and programmable pre- and post-experiment actions.

3.5 Hello, Container!: A Minimal Example

In this section, we provide an example in the style of a *"Hello, World!"* program to demonstrate the relationship among bueno CLI services, module services, and run scripts. For comprehensive information, please consult the resources provided at the end of this section.

Executing a bueno run script is a straightforward process that requires specifying an image activator, a container image (when relevant), and a run script. In the example shown in Listing 3.1, we see Charliecloud is used to activate the container defined by the image tarball named `container.tar.gz`. In this example, once bueno stages the image onto a potentially distributed collection of computer resources, it calls the `main` function located in `hello.py` (Listing 3.2). The experiment is conducted based on the program's instructions, ultimately returning control to bueno after `main` returns. Finally, bueno stores relevant data to an output directory and then terminates.

```
$ bueno run -a charliecloud -i ~/container.tar.gz -p ./hello.py
```

Listing 3.1: An example `run` service invocation of `hello.py`.

```
from bueno.public import container
from bueno.public import experiment
from bueno.public import host

def main(argv):
    experiment.name('hello')
    container.run('echo "Hello from a container!"')
    host.run('echo "Hello from the host!"')
```

Listing 3.2: Source code of the `hello.py` run script.

For more information on bueno's use in practice, including source code and online documentation, please consult the information linked below:

- **bueno Source Code**: https://github.com/lanl/bueno
- **Example Run Scripts**: https://github.com/lanl/bueno-run-proxies
- **Online Documentation**: https://lanl.github.io/bueno

4 Concluding Remarks

The bueno project aims to provide convenient access to mechanisms that aid in the automated generation, collection, and dissemination of data relevant for experimental reproducibility in computer system benchmarking. Our approach sets out to support gradations of reproducibility when confronted with production computing realities. At Los Alamos National Laboratory, we are working closely with friendly

testers to use bueno for automated performance regression testing of production HPC workloads. This work is in its infancy but appears promising.

5 Acknowledgement

Work supported by the Advanced Simulation and Computing program of the U.S. Department of Energy's NNSA and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Los Alamos National Laboratory is operated by Triad National Security, LLC, under contract number 89233218CNA000001 for the Department of Energy's National Nuclear Security Administration (NNSA).

References

[1] Richard S. Canon and Andrew Younge. "A Case for Portability and Reproducibility of HPC Containers". In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, pages 49–54.

[2] Ivo Jimenez, Carlos Maltzahn, Adam Moody, Kathryn Mohror, Jay Lofstead, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. "The Role of Container Technology in Reproducible Computer Systems Research". In *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pages 379–385.

[3] Reid Priedhorsky, R. Shane Canon, Timothy Randles, and Andrew J. Younge. "Minimizing Privilege for Building HPC Containers". In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM/IEEE. 2021. DOI: 10.1145/3458817.3476187.

[4] Reid Priedhorsky and Tim Randles. "Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC". In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pages 1–10.

[5] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors". In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 2007, pages 275–287.