Extending FreeCompilerCamp.org as an Online Self-Learning Platform for Compiler Development

Justin Gosselin
Lawrence Livermore National Laboratory
Livermore, California, USA
University of Central Florida
Orlando, Florida, USA
jpgosselin@knights.ucf.edu

Anjia Wang
Lawrence Livermore National Laboratory
Livermore, California, USA
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
awang15@uncc.edu

Peter Pirkelbauer
Lawrence Livermore National Laboratory
Livermore, California, USA
University of Central Florida
Orlando, Florida, USA
pirkelbauer2@llnl.gov

Chunhua Liao
Lawrence Livermore National Laboratory
Livermore, California, USA
liao6@llnl.gov

Yonghong Yan
University of North Carolina at Charlotte
Charlotte, North Carolina, USA
yyan7@uncc.edu

Damian Dechev
University of Central Florida
Orlando, Florida, USA
dechev@cs.ucf.edu

***Abstract*—Compilers and compiler-based tools have become increasingly critical for optimizing high-performance computing workloads; however, compiler development remains difficult and time consuming due to the complex nature of compilers. FreeCompilerCamp.org is an online training framework for compiler development that allows users to complete hands-on tutorials with a Linux environment that is directly embedded in the web browser. It provides an effective and convenient training platform for both new and experienced compiler developers. In this paper, we present our enhancements to the framework to support self-evaluation and learning outcome feedback for trainees. We extend FreeCompilerCamp to support a fully contained self-learning environment with exercises and examinations providing immediate and automatic feedback via server-side grading. We achieve this through two forms of evaluation: open-book practicals and closed-book exams. To facilitate learning, we design several new tutorials and improve the framework to support both CPU and GPU servers and docker images, optimize resource utilization, and enhance usability. Our extended platform, FreeCompilerCamp v1.1, follows the same extensibility design goals as the original to allow for new practicals and exams, providing an effective method to reduce the barrier of entry to compiler development.**

***Index Terms*—high-performance computing, compilers, educational technology, electronic learning, training**

I. Introduction

Compilers, responsible for transforming high-level source code to executable programs via an intermediary low-level language, play an essential role in computing. In HPC, compilers and source-to-source translators are an enabling technology for program analysis, optimization, and parallel programming model research (*e.g.*, [1]–[3]).

Developing these tools, however, is not an easy task. Compilers are inherently complex and are composed of many different inter-dependent phases. To learn the skillset needed for compiler and tool development requires significant time and effort for developers. While many compiler frameworks are open source, such as Clang/LLVM and ROSE, documentation is often sparse, and the learning curve is high. It is also challenging for beginners to install and configure these compilers in the proper environment, especially in the HPC field where GPUs and other heterogeneous systems are employed.

Thus, there is a need for a self-contained learning environment for tool developers to master the skills needed for their use cases, such as parallelization with OpenMP, GPU offloading, loop optimizations, and other analysis and optimization tools for HPC applications. Similar platforms and environments exist for programming education [4]–[6] and are often cloud-based online services for optimal accessibility and flexibility, requiring no installation or configuration. FreeCompilerCamp.org [7] is a free cloud-based online platform that provides hands-on tutorials for Clang/LLVM and ROSE compiler development. Users can interact with a Linux sandbox environment within the web browser, with compilers already pre-installed. While the original version of FreeCompilerCamp is an efficient resource for beginner tool developers, it is incomplete as a self-learning environment as it is lacking diversity in HPC tutorial content and student evaluation; prior to this work, there are no exercises or self-tests for students to check their understanding.

In this work, we use FreeCompilerCamp as a basis for creating a fully contained self-learning platform for compiler development that we dub FreeCompilerCamp v1.1. The contributions of this paper are as follows.

- We construct several design principles and constraints for supporting student evaluation via "closed-book" exams. Exams involve completing a larger task in a restricted environment with a basic set of tools, such as a code text editor. This encourages students to draw on their knowledge and and think critically.
- We additionally design "open-book" practicals that are similar to lab exercises and provide significant guidance for the student, including allowing full terminal access. This provides students with an opportunity to check their

understanding with simple hands-on exercises.

- We design a methodology of handling student submission of practicals and exams on the server, grading them and providing immediate feedback based on test cases provided by the author.
- We provide additional enhancements to the system to facilitate learning and ease-of-use, including supporting multiple terminal servers and docker images, particularly those supporting GPUs; optimizing resource utilization; adding more HPC tutorials, especially those on source-to-source translation with ROSE; and so on.
- We release all of the new features and improvements mentioned above on GitHub[1] so anyone who is interested in our work can deploy similar online learning systems.

As HPC workloads continue to move toward exascale computing, compiler development will become increasingly necessary and complex. We believe our extended learning platform to be an effective method of reducing the barrier of entry to this area and educating current and future practitioners.

II. Background

In this section, we provide some background on compilers using two example compilers, the goals of FreeCompilerCamp, and its limitation that we improve upon.

A. Compilers

At a high-level, compilers perform a series of complex steps to convert human readable source code into efficient machine readable executables. Within the HPC community, there are many efforts to optimize code running on supercomputers, either by relying on vendor and third-party compilers such as the GNU Compiler Collection [8] or by creating source-to-source translator tools for custom domain-level optimizations [9]. Although compiler-based tools will continue to see increased use in HPC workloads, their barrier of entry remains high and will likely remain so due to the inherent nature of compiler development. While documentation, manuals, and APIs exist, these are often verbose and challenging to comprehend as a layperson.

LLVM is a compiler framework developed to address existing problems in other compilers, such as complicated structure and aging code that is hard to maintain [10]. Clang is the native C/C++ front-end of LLVM, responsible for building the intermediate representation (IR). Using the Clang Abstract Syntax Tree (AST) and LLVM IR, developers can build cross-platform tools for code generation, transformation, optimization, and program analysis. LLVM is widely used in the HPC community due to its performance and flexible design; many vendor compilers and projects have been built on Clang/LLVM [11]–[13]. While LLVM has many online tutorials [14]–[16], the API evolves rapidly, and it is difficult to follow tutorials as they quickly become out of date, even official tutorials.

ROSE is a source-to-source compiler framework developed at Lawrence Livermore National Laboratory (LLNL) [17], with support for C/C++, FORTRAN, Java, and UPC applications [18]. ROSE can perform source-to-source transformations, optimizations, and program analyses by creating a unified mutable AST for supported languages. Built-in APIs can be used to modify and traverse the AST. ROSE also offers common analysis frameworks, including control flow graph and dataflow. Advanced developers can easily build their own tools on top of ROSE, such as producing CUDA source code from an AST generated from C source code [19]. However, the powerful capability of ROSE also makes it complex for beginners due to its extensive API and infrastructure.

[1]Our extended platform is freely available and can be obtained at https://github.com/freeCompilerCamp

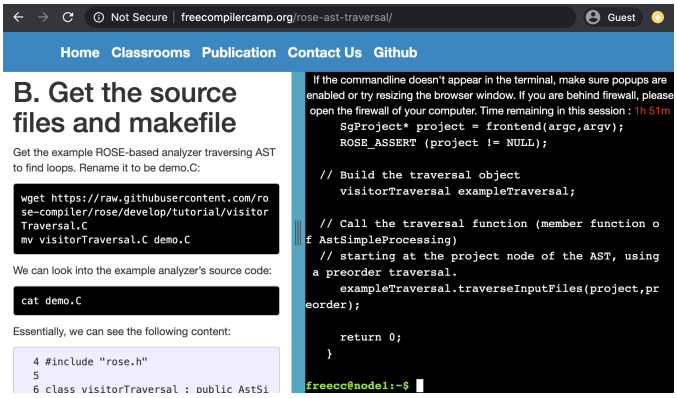

Fig. 1. An example of a tutorial in FreeCompilerCamp. The left-hand side contains the tutorial content, while the right-hand side contains the sandbox terminal. In this case, the user is going through a tutorial on traversing the ROSE AST and is viewing the source code of a sample traversal translator.

B. FreeCompilerCamp.org

FreeCompilerCamp [7], shown in Fig. 1, is a free and open-source online training platform for compiler-based development, focusing particularly on Clang/LLVM and ROSE. The infrastructure is cloud-based and uses Docker container technology to deliver a private Linux terminal pre-configured with Clang/LLVM and ROSE directly in the browser, shown on the right-hand side of Fig. 1. Users can freely read through tutorial content created by professionals, shown on the left-hand side the figure, and practice compiler development skills in the terminal, following the learning-by-doing principle of experiential learning [20]. This integrated online environment relieves users from the burden of configuring complex software development installations.

However, the initial release of FreeCompilerCamp (v1.0) is lacking in its features and content. There is no form of student evaluation in the system. Students only go through tutorial content without any "check-your-understanding"-like exercises or examinations, decreasing knowledge retention in the long term and weakening the platform as a self-learning environment.

III. Student Evaluation Design

Experiential learning theory is based on the philosophy that experience should play a large role in the learning process

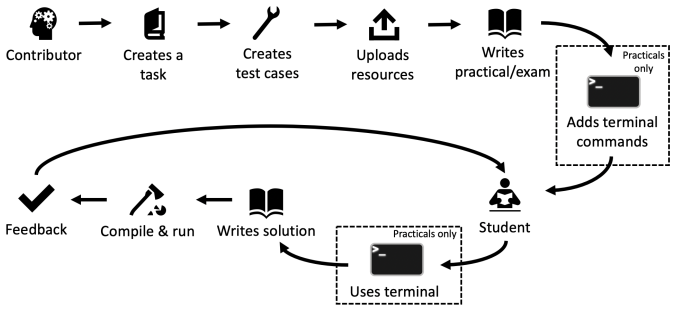

Fig. 2. Typical workflow in creating and using a practical or exam. Dashed boxes include those steps where the terminal is used; i.e., those that belong to practicals only. In exams, the terminal environment is replaced with a code text editor.

[20]–[22]. In particular, two principles of experiential learning are rooted in the design of FreeCompilerCamp:

1) The learning process should include elements of hands-on learning through experience, while also encouraging students to consider their own prior knowledge and experience.
2) Students should receive feedback during the learning process in order to engage them effectively. This is limited in the original state of FreeCompilerCamp as there is no form of student evaluation.

We strengthen FreeCompilerCamp on both of these design principles with two methods of student evaluation: "open-book" practicals and "closed-book" exams. Table I summarizes the differences between the two methods. In both cases, the tasks to be completed by the student generally involve implementing some compiler-based tool. The main difference we highlight is the user interface: for practicals, the student has access to the full sandbox terminal environment; for exams, the student has access to a code editor only. This allows practicals to act as quick, hands-on exercises and exams to provide a less guided environment, encouraging critical thinking. Although practicals and exams can be used independently, they can also be combined to act as a two-phase submission and feedback process similar to Schordan et al.'s approach to porting computer science courses to online courses [23].

Fig. 2 shows the typical workflow in creating a practical or exam with essential components and student interactions. Although the steps are nearly identical, the design considerations contained within each step differ. We discuss these design principles in the following subsections for both types of evaluation.

A. Open-book Practicals

The first step in Fig. 2 is creating the task as a contributor. In FreeCompilerCamp v1.1, practicals refer to small exercises that are completed by the student with access to the terminal, hence the title "open-book". These are similar to lab practicals in an in-person class setting, where the intent is to provide a hands-on exercise for the student to check their understanding. Practical tasks are small in scope with an expected length of 15-25 minutes and are matched with one related tutorial. For example, related to the ROSE compiler, a tutorial may focus on code analysis of OpenMP programs by traversing the AST for certain OpenMP constructs, such as `parallel` directives. A practical related to this tutorial, then, could give a student the task to perform a similar traversal but count the number of OpenMP `parallel for` directives.

TABLE I
COMPARISON OF OPEN-BOOK PRACTICALS AND CLOSED-BOOK EXAMS

	Practicals	**Exams**
User Interface	Terminal	Code editor
Expected Length	15-25 min.	30-45 min.
Evaluation Feedback	Basic (e.g., pass/fail)	Numeric score
Compilation Feedback	Full through terminal	Errors only
Test Case Design	Basic (1 or 2)	Thorough (3+)
Test Case Visibility	Open to students	Optional
Tutorial Relation	One tutorial	One or more tutorials

The next step is to provide some test cases for the student's implementation to be evaluated on. In practicals, test cases are simple with basic feedback, such as pass or fail; there should not be more than two test cases per practical to maintain simplicity. Continuing with our example of finding OpenMP directives, a simple test case would be some C input code containing two such directives. Contributors have the freedom to choose whether or not test cases are automated. If so, one strategy is to provide a `make check` target in the makefile that can be run to automatically check whether or not a test case passed or failed. Non-automated test cases can also be made, such as simply informing the student that the expected number of `parallel for` directives is 2. We intentionally leave the design of test cases and their evaluation to the contributor as the terminal is freely available for the student to use.

Finally, the contributor then uploads all resources needed for the practical to a hosted endpoint, such as GitHub, and writes the practical in Markdown to be integrated into FreeCompilerCamp, similar to writing tutorials. The hosted endpoint is used for the student to obtain any necessary skeleton code, makefiles, test cases, etc. When writing a practical, contributors must ensure two conditions:

1) There are clickable terminal commands included in the practical that automatically download all resources needed from the hosted endpoint (e.g., with `wget` or `curl`).
2) The test cases are shown in the practical and their expected output is made available. This ensures that the student has a goal to work toward and provides guidance, a major design criterion among practicals.

On the student side, students navigate to the practical page on the site and write their solution in the terminal. As the terminal is available, students can freely test their implementation with their own test cases, view source code of other translators, and use the debugger, emphasizing learning-by-doing. Finally, once the student is ready for evaluation, they compile and run their implementation on the provided test

```c
#define num_steps 2000000
#include <stdio.h>
double approx_pi()
{
  double pi = 0.0;
  double x, interval_width;
  interval_width = 1.0/(double)num_steps;

  #pragma omp parallel for reduction(+:pi)
       private(x)
  for (int i = 0; i < num_steps; i++) {
    x = (i+ 0.5) * interval_width;
    pi += 1.0 / (x*x + 1.0);
  }
  pi = pi * 4.0 * interval_width;
  return pi;
}
```

(a) An example test case (input OpenMP code) for an exam

```python
import subprocess

tests = 5
expected = [1, 2, 1, 0, 2]

# Run the submission executable on each test input code
results = [subprocess.check_output(["./submission",
      "test{}.c".format(i)]) for i in range(tests)]

correct = 0
for i in range(tests):
   if (int(results[i]) == expected[i]): correct += 1

score = (float(correct) / tests) * 100

print("Test Cases Passed: {} out of {} //// SCORE: {}%"
   .format(correct, tests, score))
```

(b) An example evaluation script in Python that runs a test suite and grades it

Fig. 3. A test case and evaluation script corresponding to an exam where the task is to count the number of OpenMP loops with `reduction` clauses. The evaluation script is run by a `make check` target that the server runs automatically. The score response is returned back to the student.

cases using the terminal. At this point, they receive feedback on the success or failure of the test cases and can freely revise as they choose, indicated by the loop in Fig. 2.

Educators interested in adding GPU support to a practical or tutorial can easily do so provided that FreeCompilerCamp now allows tutorial-specific docker images, including GPU-enabled ones. Contributors and educators need only specify an appropriate GPU-enabled Docker image in their tutorial Markdown file, thereby enabling GPU support in the terminal environments serviced to students in a tutorial or practical. This allows educators to design tasks with GPU topics in mind; for example, tasks involving playing OpenMP GPU offloading with Clang/LLVM or CUDA code generation using ROSE. More information about how we enable GPU-based dockers can be found in Section V-A.

B. Closed-book Exams

"Closed-book" exams refer to exercises where the terminal is replaced with a code editor supporting syntax highlighting. Here, the focus is on creating a confined development environment with only basic tools for the student, encouraging them to draw on all they have learned in past tutorials and practicals. Exams follow the same flow as in Fig. 2 minus the dashed boxes and contain tasks that are larger and more significant in scope than practicals, with an expected completion time of 30-45 minutes. Additionally, they may be matched to more than one tutorial. An example of an exam task that is an extension of the previous example practical is to traverse the ROSE AST to count instances of the OpenMP `reduction` clause. Such a code analysis can be useful in ascertaining opportunities for GPU offloading.

Exam test cases are similar to those in practicals; however, contributors are free to create more than two test cases to complete a thorough test suite (at least three are required). There are additional differences, including the following requirements for exam test cases:

1) Exam test cases *must* be automated via a `make check` target in order to support our system's server-side grading component.
2) Additionally, test cases must report a numeric score based on the pass/fail ratio of each test case in the suite. One strategy here is to create a Python or shell script that automatically runs the student's submission with each test case, and call this script within the makefile's `make check` target.

An example of an exam test case is shown in Fig. 3a corresponding to the example above. In this case, the test case contains a basic OpenMP implementation of approximating π. In this case, there is one `reduction` clause in the input code. We thus expect the student's submission to print "1" for this test case, indicating their implementation has found one reduction. Fig. 3b shows an example of a Python script that automatically evaluates the student's submission on a test suite. Line 4 describes the expected result for each test case as an array, and line 7 runs the submission executable on each test case input code file, the result of which is stored into an array. The two arrays are then compared and the score is calculated and output. Note that this script would be called in the `make check` target, which would return the result output by the Python script; e.g.,

```
RESULT = $(shell python eval_test.py)
check: submission
   @echo $(RESULT)
```

The output from the `make check` rule is returned from the server to the student. This methodology allows the system to grade virtually any test suite provided by the contributor.

Just like practicals, contributors should upload their makefile and test cases, as well as any skeleton code they wish to provide to the student, to a hosted endpoint. For exams, the hosted endpoint must support Git as this is used on the server to obtain the resources. They may then write the exam

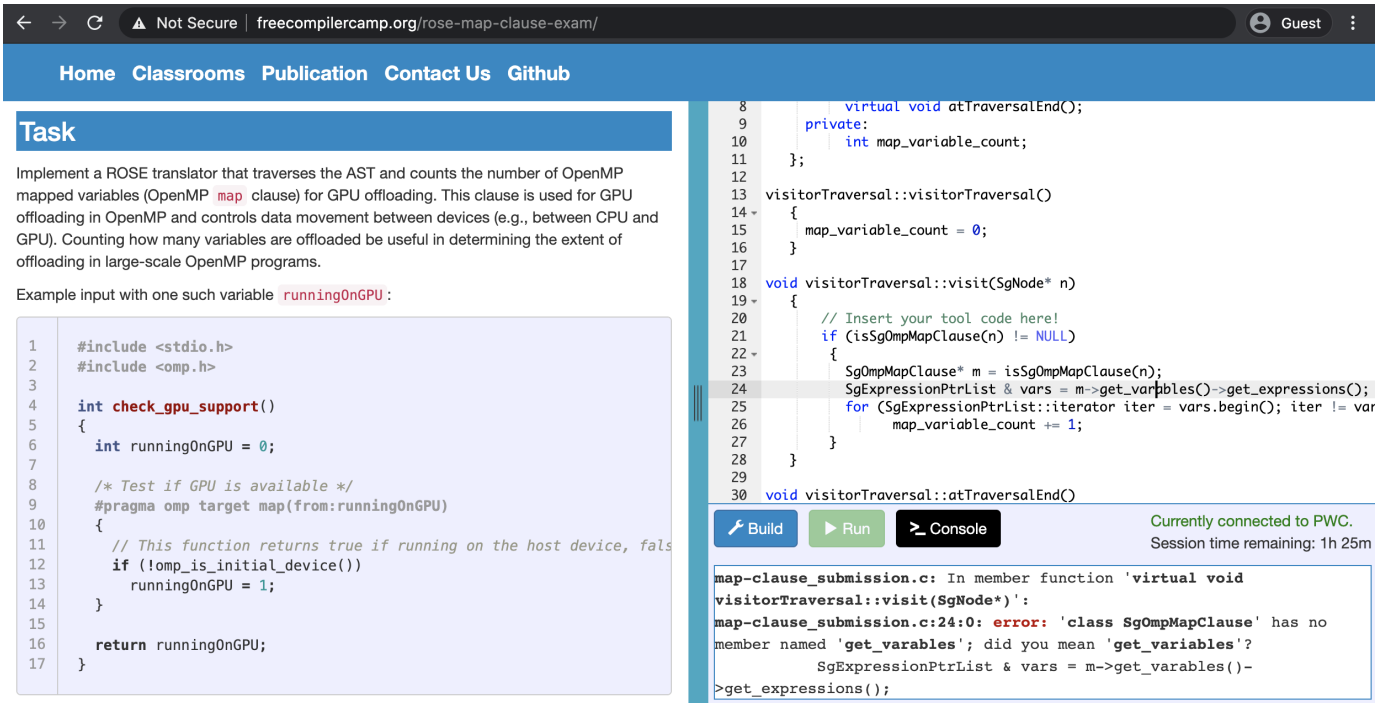

Fig. 4. An example of a closed-book exam in FreeCompilerCamp v1.1. In this case, the student is given the task to traverse the ROSE AST for OpenMP `map` directives and count how many mapped variables exist in the input code. Note there is no terminal on the right-hand side; rather, a code editor appears with a console window showing any build errors.

in Markdown the same way as practicals and tutorials, with GPU support available in the same fashion. Note, however, contributors are *not* required to reveal the test cases in the exam task description, a design choice we make in order support the possibility for exams to act as critical thinking catalysts for students.

Students completing an exam have access to the code text editor and three buttons: build, run, and view console, as shown in Fig. 4. This figure shows an exam related to the OpenMP `map` clause used for CPU/GPU data transfer, with some sample input code showing a simple check for GPU support in OpenMP. The student is given the task to count the number of OpenMP mapped variables, which may be useful in determining the extent of offloading in larger-scale programs. The build button will compile their implementation and any errors will be reported in the console view, similar to the approach taken in other online programming training frameworks such as LeetCode [5]. In the example in Fig. 4, the student has made a typo and the corresponding build error is shown. The run button tells the server to automatically run the `make check` target on the code. Finally, the numeric score will be reported back to the student from the server response from the response of the `make check` rule.

IV. Integration Into FreeCompilerCamp

In this section, we describe the technical integration of supporting student evaluation in FreeCompilerCamp v1.1. Practicals require no additional system features beyond what is already included in the original FreeCompilerCamp. We instead focus our discussion on exams.

A. Technical Background

FreeCompilerCamp is composed of two parts: the front-end website that users access (hosted at Github Pages and generated with Jekyll [24]), and the back-end Play-With-Compiler engine, written in Go, that manages terminal instances through Docker. Docker [25] is a container service that allows for the creation of lightweight packages, called containers, that can be easily deployed with pre-installed and configured resources, called images.

Play-With-Compiler (PWC) is based on Play-With-Docker (PWD), a sandbox platform for Docker tutorials. PWC and PWD use Docker-in-Docker technology to run a parent Docker container that spawns isolated child containers. This ensures server-side security as arbitrary user code runs in isolated containers. Users connecting to FreeCompilerCamp have an instance spawned by the parent container, giving them their own private terminal sandbox instance. To ensure resources are kept limited, child containers are automatically destroyed after a configurable amount of time or when the user leaves the website, whichever comes first. HTTP requests are used to interface between the front-end website and the back-end PWC engine, such as when to spawn a container. The back-end is abstracted away from the front-end; contributors need only write their tutorial, practical, and exam content using the Markdown lightweight markup language.

B. Integration of Exams

Fig. 4 shows an example of a closed-book exam in our extended platform. Our work in integration can be divided into two areas: the front-end, that we see in Fig. 4, and the back-end, which handles server-side uploading of submission and their grading.

1) Front-end integration: As the front-end of FreeCompilerCamp is built on Jekyll, contributors adding a new exam simply create a new Markdown file and freely add their task and details. We add a new layout specifically for exams that constructs the view in Fig. 4. We use Ace [26], a JavaScript library for code editors, to construct the code editor shown in the right-hand side of Fig. 4 and specify syntax highlighting for C/C++ as these are the languages currently used in Clang/LLVM and ROSE tutorials.

On the bottom of the right-hand side panel, we see a status bar containing various information about the current exam, including a connection indication and session time remaining. On the left-hand side, we have three buttons for code submission: build, which will upload the code to the serving container instance and compile it; run, which will run the `make check` target and grade the submission; and console, which shows or hides the console window. The run button is only enabled if the current code has been compiled successfully. The console window shows the status of compilation, including any errors, and shows the result of the run. In Fig. 4, the user has made a typo and the compiler error is returned to the user. Clicking the build and run buttons sends HTTP requests to the PWC backend, which are handled by the serving container.

2) Back-end integration: All compilation and test case evaluation is performed in the user's private container instead of the master Docker container, ensuring security of the server when uploading user code and isolating the container from others. Normally, PWC/PWD *requires* the terminal to be embedded in the browser in order to be spawned, as it creates and binds a websocket as part of its JavaScript API. Since exams are closed-book, we work around this by creating a wrapper around the API that uses the same endpoints but without websockets, allowing us to spawn containers "in the background" when the user opens an exam page. We additionally create several new endpoints and their request handlers in PWC to support submitting and grading exam submissions.

Once the child container has been successfully spawned on page load, the build button is enabled on the front-end, allowing the student to compile their code when ready. The first step in the compile HTTP request is to obtain all resources from the Git endpoint configured by the administrator via `git clone`. The next step is to then upload the code as a C or C++ file to a set path on the container. Finally, the code is compiled and evaluated using the `make check` target.

Compilation and evaluation are performed by using the `execRequest` structure in PWD, which encapsulates a terminal command and its parameters in a JSON object. We then use a custom function that uses the Docker Engine API to execute a command on a given container *and* capture its standard output, functionality that has been added in FreeCompilerCamp v1.1's back-end. The output of the command, such as the output of a `make` command, is then stored and copied to a string. Finally, we setup an HTTP response, and in the case of compilation, we search the output string for any keywords indicating compilation failure, including "error" and "stop". If errors exist, an HTTP 502 (Bad Gateway) response is sent to the client; else, an HTTP 200 (OK) is sent. The client receives this response and handles it appropriately in the console window as shown in the compilation error message in Fig. 4. A similar process is performed for evaluation using the `make check` command.

V. Platform Enhancements

To better serve the purpose of creating an integrated evaluation system, FreeCompilerCamp.org has been upgraded in the aspects of deployment flexibility, resource utilization, and usability. We have also added new tutorials and documentation.

A. Multiple PWC Servers and Docker Images

By default, PWC/PWD only connects to a single host machine to provide sandboxes. The host machine only starts instances based on a single Docker image, forcing all of the tutorials in FreeCompilerCamp to share the same server and docker image. However, this is not sufficient, as different tutorials may require different hardware configurations. One example is developing compilers for GPU optimizations, which need a GPU machine and Docker images supporting GPUs. Administrators may also use mixed private internal local servers and public cloud servers in their tutorials. Additionally, a single server is not scalable.

In FreeCompilerCamp v1.1, we allow multiple PWC servers to be configured. Once administrators have deployed the servers, tutorial writers can then specify the PWC server address using the `pwc` Markdown tag, such as `pwc:http://lab.yourServer.org`. The tutorial, practical, or exam will then connect to this specified PWC server.

Additionally, we provide GPU support for containers serviced to students. PWC/PWD is built upon an older official Go SDK of Docker, which has not been updated since 2017. Other than the security concerns, it does not support many features in the recent Docker releases, such as GPU support. To address this issue, we adopt an alternative Go library named Moby as the engine of PWC [27]. Using the new attribute `DeviceRequests` in the configuration, we enable the GPUs in the Docker container. CUDA toolkit 10.1 and Clang/LLVM 10.0 with GPU offloading support are preinstalled in the Docker image. It allows users to compile and run their HPC programs directly on FreeCompilerCamp as on a local machine without any other extra steps.

Moreover, tutorials may also need different software environments, such as various builds of ROSE and LLVM. For example, the compiler in one sandbox might intentionally include a bug to support a tutorial focusing on how to fix the

bug, while others tutorials may require a fully functional compiler. Conflicts could occur if the required versions are mixed together, and it increases the docker image size significantly.

To address this concern, we add support for different Docker images on any deployed server. Administrators can create different Dockerfiles, build the images, and add them to the pool of available images by adding the image name to the `AvailableDinDInstanceImages` variable in the `api.go` file. Tutorials can then specify an available docker image from the pool with Markdown tag `image` to create the sandbox, such as `image: freecompilercamp/pwc:rose`.

By supporting multiple PWC servers and docker images, instructors can choose the hardware/software combination that best fits a given tutorial. These two extensions significantly enhance flexibility and service scalability.

B. Resource Utilization

In the original PWD and PWC framework, there is no restriction on the resources that a sandbox can use. Any docker container can use 100% of the CPU and memory, which is not necessary and may severely slow down the host under extreme condition. Enabling resource allowance per user can reduce the effect of unusual usage of the sandbox to a minimum level and save operation costs.

To implement such a restriction in FreeCompilerCamp v1.1, we use the `Resources.Memory` and `Recources.CPUQuota` configurations from the Docker API. These configurations enable administrators to set the maximum memory usage and CPU time percentage for an individual container. On FreeCompilerCamp.org, up to 4GB RAM and 20% CPU time are allowed in a sandbox. While each user still sees all the resources on the host, such as 32GB memory and 8 virtual CPUs, he or she can never exceed the utilization limits.

C. Usability Improvement

Some students may not be able to access the terminal environment when reading a tutorial, or choose not to use it. In this case, it is imperative to ensure that the tutorial can be successfully completed without having to rely on output obtained from commands issued in the terminal. To better improve usability when reading tutorials, we provide collapsible code snippets to allow these users to view source code without the terminal, but also allow those using the terminal to skip it by leaving it collapsed. Additionally, we enhance code snippets to support automatic line numbers to improve readability and ease of reference.

In the tutorials, users may produce some output files, such as DOT graphs and PDFs, which can be further reviewed. Therefore, we add options at the front-end to let the user easily obtain generated files. Corresponding HTTP request handlers are available on the back-end side to transfer the files properly.

Finally, the original FreeCompilerCamp only supported vim as its editor. We add additional editors including nano and emacs so that users can have more choices based on their preferences.

D. Content and Documentation

Prior to this work, FreeCompilerCamp was limited on content. There were only five ROSE tutorials, three of which were on introductory concepts. We improve FreeCompilerCamp by adding nine new ROSE tutorials focusing on different aspects of the compiler. In particular, we add tutorials on debugging ROSE translators; working with complex data types; interfacing with Clang/LLVM; and several tutorials on program transformation, including AST modification, inlining, outlining, and loop optimization. There are now 14 ROSE tutorials organized into three sections titled Getting Started, Program Transformation and Optimization, and Parallelism.

We also add six new Clang/LLVM tutorials related to Clang AST manipulation, libtooling of Clang, AST error checking, LLVM IR basics, how to modify LLVM IR, and how to use IRBuilder to create a function. In total, there are 15 Clang/LLVM tutorials organized into three sections titled Clang, OpenMP, and LLVM.

Finally, we update and add new documentation on how to deploy and configure FreeCompilerCamp for educators who may not have a development background, especially for the deployment of practicals and exams.

VI. Related Work

There are many online educational platforms and are generally referred to as massive online open courses (MOOCs). Prominent examples include Coursera [28], edX [29], Khan Academy [30], and Udemy [31]. Research has shown MOOCs to be effective in higher education and training [32]–[35]. Computer science has found particular interest in using MOOCs for education with platforms such as FreeCodeCamp [4], LeetCode [5], and Codecademy [6]. Universities have also begun to offer content of their classes online, such as MIT Opencourseware [36], which has an abundance of computer science topics available. Despite the availability of these MOOCs and platforms, content on hands-on compiler development remains sparse.

Many of these platforms include forms of evaluation through examinations or exercises. FreeCodeCamp, for example, provides users with programming exercises to learn web development by creating HTML components. LeetCode, designed for algorithms and interview question training, follows a similar format as ours to provide a code text editor and allows students to submit their solutions to be graded server-side. Compiler development is more challenging to evaluate as tools can become complex, especially in the context of their input and output. We rely on professionals in the field to design effective test cases as part of their contribution to new exams.

VII. Conclusions and Future Work

In this paper, we have discussed our efforts in extending FreeCompilerCamp as a contained self-learning environment by adding an evaluation component through check-your-understanding practicals and closed-book exams. We have also added more content, with various new features to enable

flexible deployment, efficient resource utilization, and better usability.

Our goal is to provide an effective method of training practitioners on compiler development, which has become increasingly necessary in the HPC realm. Rather than having to rely on verbose documentation and manuals, HPC educators can use our platform to easily train students and practitioners with hands-on tutorials, while students can check their understanding with exercises and exams. The modularity of our platform also allows educators and students to easily find tutorials in specific areas of their interest, such as OpenMP, ROSE, Clang/LLVM, GPU offloading, and so on. We provide various pre-configured resources, including two different compiler frameworks and GPU server support, allowing for a wide range of topics to be covered with our platform. Moreover, we welcome professionals and educators to add new tutorials, practicals, and exams, and we invite educators to deploy their own versions of our platform. Overall, educators and practitioners can use our platform as a means of training to enable further progress on future compiler-based tools to increase productivity and performance in HPC workloads.

In the future, we plan to incorporate an account system to track student grades and progress. This will allow us to create a certification system whereby evidence of knowledge can be issued. We also intend to provide additional feedback for closed-book tests, including detailed results on test cases after submission. Additionally, we plan to deploy FreeCompilerCamp v1.1 to classrooms and workshops and conduct a user study to determine the effectiveness of the platform. Finally, we will continue to add new tutorials and evaluate other compiler frameworks to include in the platform.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and partially supported by the U.S. Dept. of Energy, Office of Science, ASCR SC-21), under contract DE-AC02-06CH11357. IM Release Number: LLNL-CONF-813550.

REFERENCES

[1] P. Pirkelbauer, P. Lin, T. Vanderbruggen, and C. Liao, "Xplacer: Automatic analysis of data access patterns on heterogeneous cpu/gpu systems," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 997–1007.

[2] H. Ahmed, A. Skjellum, P. Bangalore, and P. Pirkelbauer, "Transforming blocking mpi collectives to non-blocking and persistent operations," in *Proceedings of the 24th European MPI Users' Group Meeting*, ser. EuroMPI '17. New York, NY, USA: Association for Computing Machinery, 2017.

[3] A. Wang, Y. Shi, X. Yi, Y. Yan, C. Liao, and B. R. de Supinski, "Ompparser: A standalone and unified OpenMP parser," in *OpenMP: Conquering the Full Hardware Spectrum*, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds. Cham: Springer International Publishing, 2019, pp. 140–152.

[4] "FreeCodeCamp," https://www.freecodecamp.org/, accessed: 2020-08-11.

[5] "LeetCode," https://leetcode.com/, accessed: 2020-08-09.

[6] "Codecademy," https://www.codecademy.com/, accessed: 2020-08-11.

[7] A. Wang, A. Mishra, C. Liao, Y. Yan, and B. Chapman, "FreeCompilerCamp.org: Training for OpenMP compiler development from cloud," *Journal of computational science education*, vol. 11, no. 1, 2020.

[8] "GCC, The GNU Compiler Collection," https://gcc.gnu.org/, accessed: 2020-08-13.

[9] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[11] AMD. (2020) AMD optimizing C/C++ compiler. [Online]. Available: https://developer.amd.com/amd-aocc/

[12] LLVM. (2020) Projects built with LLVM. [Online]. Available: https://llvm.org/ProjectsWithLLVM/

[13] IBM. (2020) IBM XL C/C++ compiler. [Online]. Available: https://www.ibm.com/us-en/marketplace/xl-cpp-linux-compiler-power

[14] LLVM. (2020) Getting started/tutorials. [Online]. Available: https://llvm.org/docs/GettingStartedTutorials.html

[15] IBM. (2020) Create a working compiler with the LLVM framework. [Online]. Available: https://www.ibm.com/developerworks/opensource/library/os-createcompilerllvm1/index.html

[16] A. Sampson. (2015) LLVM for grad students. [Online]. Available: https://www.cs.cornell.edu/~asampson/blog/llvm.html

[17] (2020) ROSE documentation. [Online]. Available: http://rosecompiler.org/ROSE_HTML_Reference/index.html

[18] C. Liao, D. J. Quinlan, T. Panas, and B. R. De Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *International Workshop on OpenMP*. Springer, 2010, pp. 15–28.

[19] C. Liao, Y. Yan, B. R. de Supinski, D. J. Quinlan, and B. Chapman, "Early experiences with the OpenMP accelerator model," in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer, 2013, pp. 84–98.

[20] A. Y. Kolb and D. A. Kolb, "Learning styles and learning spaces: Enhancing experiential learning in higher education," *Academy of management learning & education*, vol. 4, no. 2, pp. 193–212, 2005.

[21] U. Baasanjav, "Incorporating the experiential learning cycle into online classes," *Journal of Online Learning and Teaching*, vol. 9, no. 4, p. 575, 2013.

[22] D. A. Kolb, *Experiential learning: Experience as the source of learning and development*. FT press, 2014.

[23] M. Schordan, C. Kollmitzer, R. Pucher, and G. Brezowar, "How to convert a traditional course in programming into an online course," in *EDULEARN10 Proceedings*, ser. 2nd International Conference on Education and New Learning Technologies. IATED, 5-7 July, 2010 2010, pp. 517–522.

[24] "Jekyll," https://jekyllrb.com/, accessed: 2020-08-13.

[25] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[26] "Ace," https://ace.c9.io/, accessed: 2020-08-13.

[27] "Moby Project," https://mobyproject.org/, accessed: 2020-10-06.

[28] "Coursera," https://www.coursera.org/, accessed: 2020-08-11.

[29] "edx," https://www.edx.org/, accessed: 2020-08-11.

[30] "Khan academy," https://www.khanacademy.org/, accessed: 2020-08-11.

[31] "Udemy," https://www.udemy.com/, accessed: 2020-08-11.

[32] K. S. Hone and G. R. El Said, "Exploring the factors affecting mooc retention: A survey study," *Computers & Education*, vol. 98, pp. 157–168, 2016.

[33] G. Christensen, A. Steinmetz, B. Alcorn, A. Bennett, D. Woods, and E. Emanuel, "The MOOC phenomenon: who takes massive open online courses and why?" *Available at SSRN 2350964*, 2013.

[34] D. O. Bruff, D. H. Fisher, K. E. McEwen, and B. E. Smith, "Wrapping a MOOC: Student perceptions of an experiment in blended learning," *Journal of Online Learning and Teaching*, vol. 9, no. 2, p. 187, 2013.

[35] A. M. F. Yousef, M. A. Chatti, U. Schroeder, and M. Wosnitza, "What drives a successful MOOC? an empirical examination of criteria to assure design quality of moocs," in *2014 IEEE 14th International Conference on Advanced Learning Technologies*. IEEE, 2014, pp. 44–48.

[36] "MIT opencourseware," https://ocw.mit.edu/index.htm, accessed: 2020-08-11.

APPENDIX
ARTIFACT DESCRIPTION APPENDIX

In this appendix, we provide a step-by-step guide on installing and configuring FreeCompilerCamp v1.1 as a personal instance of the platform. We additionally discuss how to add a new tutorial, practical, and exam. Readers interested in a more detailed installation guide are encouraged to reference our GitHub repository (see footnote 1). Note that our website can be freely browsed at FreeCompilerCamp.org without any installation.

A. Prerequisites

- Install the following prerequisite software:
  - Docker engine and docker-compose
  - Golang 1.14+
- Please refer to external guides on the installation process of these resources.

B. Deploying PWC

- Initialize the Docker swarm with `sudo docker swarm init`
- Clone the PWC git repository available at https://github.com/freeCompilerCamp/play-with-compiler
- Pull the relevant Docker images from our Dockerhub repository[2] with `docker pull`
  - Our extended platform supports various images of ROSE and Clang/LLVM. For example, to obtain the ROSE image for closed-book exams, we can use `docker pull freecompilercamp/pwc:rose-exam`.
  - The default image is freecompilercamp/pwc:full, which contains both ROSE and Clang/LLVM. It needs to be pulled to be used.
- Setup the Go environment by downloading dependencies with `go mod vendor` while in the PWC top-level directory.

C. Configuring PWC

- Navigate to the `config/config.go` file in PWC.
- Change the line containing `PlaygroundDomain` to the domain that is to be used for the PWC engine. It must end in port 5010; e.g., `localhost:5010`.
- Start PWC with `docker-compose up -d` while in the top-level PWC directory.
  - This will start three services: `l2` (backend), `haproxy` (frontend), and `pwd` (Play-with-Docker).
  - Verify that the services have successfully started with `docker ps`.
- To stop PWC, use `docker-compose down --volumes`.

[2]Our Dockerhub repository is publicly available, where all of our supported images can be found: https://hub.docker.com/r/freecompilercamp/pwc

D. Deploying Front-end Website

- Clone the FreeCompilerCamp git repository available at https://github.com/freeCompilerCamp/freecompilercamp.github.io
- Navigate to the `_config.yml` file and change the `url` tag to the domain that students will access. Additionally, change the `pwdurl` to the domain of the PWC engine setup in Section C.
- To deploy on a web server, build the site with `jekyll build` at the top-level FreeCompilerCamp directory, and move the content of the generated `_site` folder to the web server. To use localhost, issue `jekyll serve`, which will run a local server on port 4000. GitHub Pages can also be used.
- The site can now be accessed via the URL setup above and PWC will spawn container instances for each tutorial, practical, or exam opened. Verify that a terminal appears for these pages.

E. Creating a Tutorial

- Navigate to the `_posts` folder and create a new Markdown file in the format `yyyy-mm-dd-tutorial-name.markdown`.
- At the top of the file, specify the layout as `post`; i.e., `layout: post`. Follow the same format to add author, date, and title tags. To specify a server or image, use the `pwc` and `image` tags, respectively.
- Write the tutorial freely in Markdown. See the styling guide on our GitHub repository for more information on how to create special code snippets.
- Open the existing ROSE or LLVM landing pages and add the new tutorial to the existing list to link it to the landing page.

F. Creating a Practical

- Create a new Markdown file in the same format as a tutorial, and link it to the landing page. Use `post` as the layout.
- Write the practical task. Ensure that the task is well articulated, understandable, and follows the design criterion outlined in this paper.
- Write the test cases and their expected output *in the practical content* to ensure the student can view them, a requirement of practicals.
- Upload all required files (Makefile, skeleton code, test cases, etc.) to a hosted endpoint, such as GitHub, that can be used with the `wget`, `curl`, or similar commands.
- Create clickable code snippets with ```` ```.term1 <commands> ``` ```` to include commands for the student to download all required resources.
- Use clickable code snippets to include commands for running the evaluation, if using automatic test cases.

G. Creating an Exam

- Create a new Markdown file in the same format as a tutorial, and link it to the landing page. Use `exam` as the layout.
- Specify the following additional Markdown tags based on the exam preferences:
  - `exam_name: <exam_name>`
  - `exam_language: c or c++ or cpp`
  - `image: freecompilercamp/pwc:rose-exam or freecompilercamp/pwc:llvm-exam`
- Write the exam task and test cases and ensure they follow the design criterion outlined in this paper.
- Create a Makefile that contains a compilation target to compile the solution and a `make check` to evaluate it.
  - Student submissions will *always* be uploaded to the container as `<exam_name>_submission.c or <exam_name>_submission.cpp` depending on the language tag, and both Makefile commands will only be run in the same directory containing the source file.
- Write the `make check` rule in the Makefile.
  - External scripts that grade the submission can be called from within the Makefile; e.g., the Python example in this paper.
  - The rule should output a numeric score.
- Upload all resources to a hosted git repository. Please follow the guidelines below.
  - The repository must have a directory named `exams` at the top-level directory of the repository.
  - The `exams` directory must have subdirectories that have the same name as the name provided in the `exam_name` tag. For example an exam named `complex-types` should have a directory with the same name in the `exams` directory. Place all resources needed for the exam in this subdirectory.
  - We recommend using GitHub. The repository can contain other directories and files so long as the two points above are followed.
- Navigate to the `config.go` file in PWC.
  - Update the `RoseExamEndpoint` and/or `LLVMExamEndpoint` to be the top-level link (the cloneable link) of the git repository from the previous step.
  - It should be possible to call `git clone` with this link.
- Navigate to the `_config.yml` file in FreeCompilerCamp.
  - Update the same variables to be the same link as in the previous step.
- Test the deployed exam and ensure that the submission compiles and runs the evaluation.