Encouraging Compiler Optimization Practice for Undergraduate Students through Competition

Yu Zhang
yuzhang@ustc.edu.cn
University of Science and Technology of China
Hefei, China

Chunming Hu
hucm@buaa.edu.cn
Beihang University
Beijing, China

Mingliang Zeng
zengmingliang1998@gmail.com
University of Science and Technology of China
Hefei, China

Yitong Huang
hyt@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, China

Wenguang Chen
cwg@tsinghua.edu.cn
Tsinghua University
Beijing, China

Yuanwei Wang
wangyw20@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

ABSTRACT

AI and other emerging applications demand domain-specific architectures which require compiler techniques such as back-end generation for different architectures and optimizations. However, traditional undergraduate compiler courses emphasize the front-end, while code generation and optimization are rarely involved. To motivate universities to have more industry-friendly compiler courses, we have designed a national compiler design competition for undergraduates to include compiler techniques beyond parsing. Moreover, we provided reliable, continuous cloud storage and an online evaluation platform for distributed competitors. In the 9-week Competition in 2020, each team (up to 4 students) was required to implement a compiler for a given `SysY` language and given target hardware (Raspberry Pi 4B). The performance was evaluated by executing code generated by the compiler on real hardware. Finally, 21 of 72 teams successfully passed all functional test cases; 12 of 21 teams implemented optimizations showing significant speedup over `gcc -O0`; furthermore, compilers of the top 3 teams performed better than `gcc -O2` on the given 10 performance test cases. Some advanced optimization techniques, such as multithreading and SIMD, were used by some teams. This paper summarizes the competition and further thoughts on compiler courses.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Social and professional topics** → *Computational science and engineering education.*

KEYWORDS

compiler design; competition; online judge system; optimization

ACM Reference Format:
Yu Zhang, Chunming Hu, Mingliang Zeng, Yitong Huang, Wenguang Chen, and Yuanwei Wang. 2021. Encouraging Compiler Optimization Practice for Undergraduate Students through Competition. In *26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2021), June 26–July 1, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3430665.3456354

1 INTRODUCTION

Compilers are important core infrastructures for building other software. In recent years, with the rise of AI and other emerging applications, more domain specific architectures such as GPUs, TPUs or FPGAs are more prevalent. As the complexity and diversity of applications and underlying architectures are increasing, it becomes more and more difficult to generate efficient code. In the end, compilers play an increasingly important role in making programs run efficiently on the given customized platform [1].

Traditional undergraduate compiler courses, at least in China, mainly focus on the front-end. However, compiler techniques needed by the industry are not just parsing, but using the compiler to generate code that can run on real hardware, optimize the code to eliminate redundancy, and analyze the code to find potential vulnerabilities. In addition to the gap in compiler knowledge pointed out by Radermacher *et al.* [6, 7], graduating students lacked skills in different fields, including technical skills (design, testing and configuration management tools, *etc.*), personal skills (oral and written communication, teamwork, *etc.*) and professional qualities (ethics *etc.*). Computer educators have the responsibility to make course changes to better prepare students for their future careers.

In June 2012, the Teaching Steering Committee for Computer Majors of the Ministry of Education (TSC-CS-MOE) in China established a special project to deal with the computer education reform for systemic ability training. After years of hard work, the project team has formed a teaching and experimental integrated system for the core computer system courses. In order to further cultivate the innovation ability of undergraduate students as well as the above technical skills, personal skills and professional quality, the TSC-CS-MOE decided to promote professional construction through subject competitions. In 2017, the first National Undergraduate Student Computer System Ability Competition (CPU Design Competition) was held, with 70 teams from 45 universities participating. From the first to the third, the scale of the competition has increased to 68 universities distributed in 22 provinces. In 2020, the individual competition was added, and the team/individual competitors involved 91 universities in 28 provinces. Through the competition, micro-architectures designed by teams have become increasingly complex, greatly improving the teaching level and students' abilities.

Motivated by the effectiveness of the CPU Design Competition, we came up with the idea of a compiler design competition in 2018. After a year of deliberation and many substantive discussions since the second half of 2019, we prepared and launched the 2020 National Undergraduate Student Compiler Design Competition in China (abbreviated as the 2020 Competition). The competition required each enrolled team to implement an optimizing compiler from a given `SysY` language to a given target hardware (Raspberry Pi 4B), and to use self-developed test suite to evaluate the function and performance of the compiler on the provided online judge (OJ) platform. In the 2020 Competition, 72 teams from 47 universities signed up, and 21 of the 72 teams successfully passed all functional test cases; 12 of 21 teams implemented optimizations showing significant speedup over `gcc -O0`; furthermore, compilers of top 3 teams performed better than `gcc -O2` on the given 10 performance test cases. Some advanced optimization techniques can be seen in the student compilers, such as multithreading and SIMD, showing the value of the competition to tap the potential of students.

2 DESIGN OF THE COMPETITION

2.1 Design Goal

The goal is to encourage undergraduates to learn and practice compiler techniques beyond parsing. To achieve the goal, we have carefully designed the competition: each team is required to write an optimizing compiler for a certain language from scratch, which can run on real hardware. We explain this decision as follows:

1) *Writing the compiler from scratch.* Instead of adding optimization passes in an existing compiler infrastructure, which may be used in some courses, we asked teams to write their compilers from scratch. The major point for this decision is because the design of Intermediate Representation (IR) is the core of the compiler design. Writing a pass on an existing compiler would require the use of fixed IR, which would not stimulate students to understand the connection/dependency between IR and optimizations.

2) *Generating executables on real hardware.* Virtual machines or simulators are also extensively used in compiler courses due to their good portability and independence of real hardware. Although they are fine for front-end dominated courses, they are obviously not proper execution environments for evaluating compiler optimizations. They are normally very slow and cannot provide reliable performance metrics for compiler optimizations. In addition, they are particularly poor to evaluate architecture-dependent optimizations, such as register allocation and instruction scheduling. Therefore, we chose to evaluate compilers on real hardware in the competition.

3) *Language features: simple but suitable for mining optimizations.* We have added substantial complexity to the compiler due to the previous two decisions, so we need to make the language simpler to make the total time spent by undergraduate students in the competition manageable. We introduced a simple subset of C-like for the competition so that students can quickly understand based on their own C language foundation. Although the language for the competition is simple, it needs to be able to explore different compiler optimizations. Therefore, the language we chose for the competition only has primitive types `int` and multi-dimensional arrays of `int`.

2.2 Content of the Competition

According to the above goals, we designed `SysY` language for the competition and chose real hardware that supports RISC (Reduced Instruction Set Computing) as the target platform.

2.2.1 `SysY` *Language.* To save students from developing the preprocessor, each `SysY` program is stored in a single file with the extension `sy`. As mentioned before, `SysY` was designed as a simple C-like subset with restricting types to only include 32-bit `int` and multi-dimensional integer arrays stored in row-first order. The control flow structure of `SysY` is similar to C, but there is no `for` statement. `SysY` supports basic arithmetic operators except ++ and −−, as well as all relational and logical operators in C. The precedence and associativity of its operators as well as calculation rules (including short-circuit logical operations) are consistent with the C language. The function in `SysY` can take parameters or not. The parameter type can be `int` or an integer array type, but the return type can only be `int` or `void`. When the parameter is of type `int`, it is passed by value; but when the parameter is of array type, what is actually passed is the starting address of the array, and only the length of the first dimension of the formal parameter can be vacant.

`SysY` *Runtime.* `SysY` itself does not provide constructs for input/output (I/O). The I/O and timing functions are provided in `SysY` runtime. As shown in Fig. 1, `SysY` library functions can be called within functions in the `SysY` program, *e.g.,* the `get*/put*` I/O functions and `starttime-stoptime` pair timing functions. It should be noted that parameter types of some `SysY` library function will exceed data types in `SysY`, *e.g.,* the format string in `putf` function.

Potential Optimization Space of the `SysY` *Program.* `SysY` allows constants, variables and function parameters of multi-dimensional integer array types, making it easy to express various algorithms related to integers (including the vector, matrix and even tensor) and to support testing with various scale data sets. Large-scale or multi-dimensional arrays can be used to evaluate the data storage layout; the combination of arrays and loops can be used to evaluate the addressing optimization, access locality, vectorization and even parallelization. Function calls can be used to evaluate interprocedural optimizations such as function inline and CPS (continuation-passing style) optimization. Although there is no `for` loop, it can be represented by `while` and the loop can be further used to evaluate loop identification and loop optimizations.

2.2.2 Target Hardware Selection. We chose real hardware supporting RISC, because the RISC system was designed to improve performance with a smaller instruction set and simpler instructions. Therefore, the focus of the compiler is how to generate fewer

```
int a[5][5]={1,2,3,4,5};
int func(int a[][5]){
    int i=0, j=0, sum=0;
    starttime();
    while(i<5){
        while(j<5){
            sum=sum+a[i][j];
            j=j+1;
        }
        i=i+1;
    }
    stoptime();
    return sum;
}
int main(){
    putarray(25, a); putch(10);
    putint(func(a)); putch(10);
    putf("Get %d elements",
        getarray(a[0]));
    return 0;
}
```

Figure 1: A SysY Program Example.

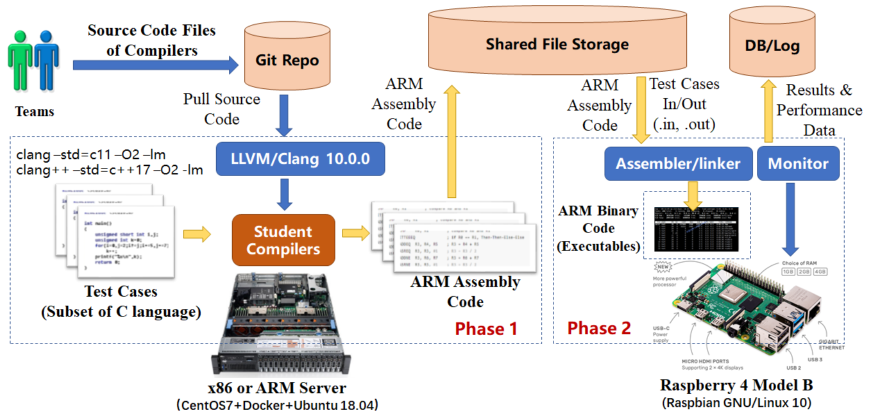

Figure 2: Workflow of the Online Judge System.

instructions for a given SysY program, and secondly select low-overhead instructions instead of high-overhead ones. Another consideration is price. We hope to use low-end hardware to allow students to practice the construction of the compiler. This will make it easier for more universities to purchase experimental equipment at a low cost, and drive the practice of compiler courses through the competition, thereby promoting teaching.

Based on the above considerations, we at first tried to use the RISC-V Dev Board of less than 100 RMB, *e.g.,* GD32 VF103 [3], but due to its low configuration (up to 128 KB on-chip Flash memory and 32KB SRAM memory), it was later discarded due to the following problems: 1) unable to run full Linux, 2) unable to test large data sets, 3) only through USB DFU (Device Firmware Upgrade) tool to update programs, not supporting online judgement. We then chose Raspberry Pi 4B [2] with a price of about 300~400 RMB, which has mature tool chain. The selected target hardware has an ARM Cortex-A72 CPU and 2GB LPDDR4 SDRAM memory, which can meet the running and testing needs of the competition, and also support updating and automatic evaluation through the network.

2.2.3 The SysY *Compiler.* Each team needed to develop its own student compiler to compile .sy file from scratch and generate the ARM assembly code related to the target hardware, then call the existing assembler and linker to generate the ARM executable file.

It should be emphasized that since SysY runtime contains features beyond SysY language, such as string parameters in putf function, the student compiler must not only handle the features of SysY language itself, but also correctly handle SysY runtime calls.

2.3 Online Judge System and Platform

2.3.1 Demand Characteristics of the OJ System. The OJ system for the compiler design competition needs to provide a safe, reliable and continuous cloud storage and evaluation platform for distributed competitors. Traditional programming-oriented OJ systems often use virtual machines (VM) or simulators for evaluation, while the OJ system here needs to evaluate the performance of the compiler on real hardware. The main reasons are as follows: 1) Only measurement on real hardware can provide reliable performance metrics for optimizations. 2) VMs and simulators perform particularly poor when evaluating architecture-dependent optimizations (such as register allocation and instruction scheduling).

2.3.2 Structure of the OJ System. The workflow of the OJ system is shown in Fig. 2, which is mainly divided into two phases. One is to use the compiler submitted by a team to generate the ARM assembly file for each test case, and the other is to schedule and evaluate the generated ARM assembly file on real hardware.

In the first phase, the teams first upload the source code of their compilers to the Git repository. Once a team explicitly submits the compiler to be evaluated to the platform, the OJ system will pull the code from the repository and build it using LLVM/Clang to generate the student compiler. Next, the OJ system invokes the student compiler to compile each test case and generate the corresponding ARM assembly file, which will be further uploaded to the Shared File Storage. This phase can be carried out on an x86 or ARM server.

In the second phase, our OJ system downloads all ARM assembly files to be evaluated from the Shared File Storage to real hardware, converts them into ARM binary code after assembling and linking, and then evaluates the executables on the real hardware after injecting the test data set. Meanwhile, the monitor in the OJ system outputs the results (including performance data) to the Log Server.

2.3.3 Innovations of the OJ System. To sum up, the proposed OJ system has the following innovations. First, we extend the programming-oriented OJ to the comprehensive project evaluation of any custom evaluators, such as the automatic evaluation of compilation, artificial intelligence and other disciplines. Second, the OJ system integrates the collaborative development functions of Git and performs automatic evaluation, making it easy to quantify the contributions of team members. Third, the OJ system realizes highly extensible distributed cluster management and task scheduling, and the test platform can be real machines (Raspberry Pi 4B) or cloud resources for remote scheduling. Fourth, the stability guarantee mechanism is designed for the test platform, including cleaning abnormal processes and data, the exclusivity of the ongoing evaluation tasks to Raspberry Pi, frequency locking to avoid fluctuations, *etc.*

2.4 Test Suite

2.4.1 Overview of the Test Suite. The test suite we designed for the competition includes two parts. The first part is the set of functional test cases, which mainly examines the correctness of the submitted compilers. The second part is the set of performance test cases, which mainly examines the performance of the submitted compilers. In the preliminary competition, there were 110 functional test cases, 80 of which were public, and there were 5 groups of performance test cases, 2 of which were public. Each group of performance test cases had 3 different input test points. In the finals, another 5 groups of hidden performance test cases were added, and the hidden performance test cases used in the preliminary round were also disclosed. No new functional test cases were added in the finals. This is because the competition pays more attention to the performance of the compilers designed by the participating teams.

2.4.2 The Performance Test Cases. The set of performance test cases aims to provide imperfect codes that implement some generic kernels to inspect the data flow analysis and optimization capabilities of student compilers. Table 1 lists the performance test cases of the ten generic kernels, and each kernel test case has three input data sets of different sizes. The table also lists the running time of the performance test cases on the target Raspberry Pi 4B. All test cases were compiled by `gcc` with `-O0` or `-O2` option.

Table 1: The description of the performance test cases

Name	LOC	Description	Data Set	Runtime (s) (using gcc)		
				-O0	-O2	Speedup
`mm1`	90	Matrix multiplication between two $n \times n$ matrices	n=500, nnz$_{A,B}$=70%	6.33	1.03	6.11
`mm2`			n=500, nnz$_{A,B}$=50%	5.34	0.87	6.12
`mm3`			n=500, nnz$_{A,B}$=30%	3.89	0.66	5.92
`spmv1`	51	Sparse matrix-vector multiplication $A_{n \times n}x$ nnz$_A$ = m	n=50000, m=10^6	2.73	0.74	3.71
`spmv2`			n=10^5, m=5×10^5	2.19	0.50	4.42
`spmv3`			n=1000, m=10^6	1.99	0.46	4.91
`bitset1`	67	Process n randomly generated bitset operations	n=10^7	2.53	0.51	4.91
`bitset2`			n=2×10^5	2.19	0.50	4.42
`bitset3`			n=4×10^7	5.81	1.64	3.54
`mv1`	71	Calculate $A^{100}x$, A is a matrix of $n \times n$ and x is a vector	n=2000 nnz$_A$=80%	4.40	2.30	1.92
`mv2`			n=2000 nnz$_A$=10%	2.36	0.93	2.54
`mv3`			n=2000 nnz$_A$=50%	2.45	1.33	1.84
`sort1`	71	Radix sort an array of length n	n=10^6	1.24	0.23	5.46
`sort2`			n=3×10^7	14.68	4.28	3.43
`sort3`			n=5×10^6	5.24	0.86	6.11
`conv1`	107	m $c \times c$ conv operations on a matrix of $n \times n$	c=5,n=200,m=40	23.56	15.93	1.48
`conv2`			c=3,n=1000,m=20	35.68	22.93	1.56
`conv3`			c=5,n=5000,m=4	26.68	15.20	1.76
`fft1`	121	Use FFT to calculate multiplication of p-order polynomial	p=50000	22.74	6.57	3.46
`fft2`			p=100000	47.88	13.82	3.46
`fft3`			p=150000	46.38	13.53	3.43
`median1`	121	Use quick sort to calculate median of an array of length n	n=49999	8.28	2.52	3.28
`median2`			n=99999	7.69	0.016	473
`median3`			n=100001	50.01	11.38	4.39
`transpose1`	51	m matrix transpose operations on an array of length n	n=10^7,m=30	8.08	3.46	2.33
`transpose2`			n=11741730,m=40	11.33	3.27	3.47
`transpose3`			n=9072000,m=-100	18.20	11.30	1.61
`shuffle1`	51	n key-value pairs and m queries for each key, calculate the sum of values with the same key	n=10^5,m=10^5	7.93	9.79	0.81
`shuffle2`			n=m=10^5	10.73	10.86	0.99
`shuffle3`			n=m=5×10^6	6.20	5.36	1.16

nnz: number of non-zero elements FFT: Fast Fourier Transformation

Next, we briefly describe the design ideas of these performance test cases. First, there are lots of duplicated or even meaningless calculations in almost all of these kernel codes. Data flow analysis technology will help eliminate these redundant codes and achieve a pretty good speedup ratio in these test cases. Second, the designed kernel codes also provide a stage for exploring conventional loop optimizations. For example, the `bitset` kernel uses a constant-ranged loop to calculate $2^n (0 < n \leq 31)$ when invoked, which is the main load of this kernel. Simple loop unrolling optimization will lead to almost 2× speedup in this kernel test cases. When implementing the `SpMV` kernel, we split the inner loop into two loops of the same range with no data-dependence. A loop fusion optimization would work pretty well. Finally, we provide opportunities for competitors who implement the most complex optimization method IPA (interprocedural analysis). IPA causes a speedup of up to 100× for one of the `median` kernel data sets.

3 THE 2020 COMPETITION AND RESULTS

This section introduces the organization, results, effectiveness of the competition, and the questionnaire survey of participating teams.

3.1 Organization of the 2020 Competition

Team Composition. The participants were required to join as a team for the Competition. Each team member should be a full-time undergraduate from the same regular college or university, and each team had a maximum of 4 members and at most 2 instructors. There should be no more than 2 teams from the same college/university.

Schedule of the Competition. Table 2 shows the schedule of the competition. Due to COVID-19, the team registration, 2 rounds of competition, final defense and award ceremony were all online. After each team successfully registered, the competition committee would mail a Raspberry Pi for their use. The competition also held two online trainings, in which experts from education and industry were invited to give reports on compilation and programming languages, and the competition committee interpreted the competition.

Table 2: Schedule of the 2020 Competition

Event	Start time	End time
Registration	2020.05.01	2020.05.30
First Training	2020.05.16	-
Raspberry Pi Distribution	2020.06.16	-
Preliminary	2020.06.06 08:00	2020.08.06 08:00
Second Training	2020.08.08	-
Post the Finalists	2020.08.10	-
Finals: Online Answer	2020.08.19 08:00	2020.08.20 18:00
Finals: Defense	2020.08.21 08:00	2020.08.21 15:00
Finals: Award Ceremony	2020.08.21 16:30	2020.08.21 18:00

Competition Resources and Website. The competition had an online test platform and notification website (https://compiler.educg.net/), which also had a ranking of the competition. Simultaneously, the competition provided a GitLab code hosting platform(https://gitlab.eduxiji.net) for teams to develop and submit compiler code.

3.2 Preliminary and Final Results

In the 2020 Competition, totally 72 teams from 47 universities signed up for the competition, and built their own compilers from scratch.

Git Submission Statistics. Fig. 3 shows the daily number of Git submissions in the 2020 competition. Although submissions were allowed from May 11th, the total number of submissions in the first

5.5 weeks was very small, about 180. This is because the competitors were busy attending classes or developing graduation projects. In the 9 weeks since June 19, all teams generated 5,443 submissions to the OJ platform, including 4,079 submissions during the 7-week of the preliminary round, showing the high interests of this new competition. In particular, on July 30th, the number of submissions was 463, reaching the highest daily peak since the competition.

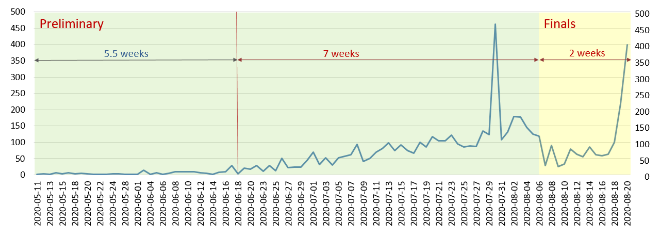

Figure 3: Submission statistics of the 2020 Competition

Preliminary Results. Among all 72 enrolled teams, 40 teams submitted their own compilers to the platform for testing and 31 of them achieved effective execution results. All teams submitted 2541 tests, of which 920 submissions got accepted (*i.e.,* passed all test cases), and the pass rate was 36.2%. 21 teams passed the functional tests with full marks, accounting for 29.2%. 13 teams received effective performance test points, accounting for 18%.

Final Results. There were 21 teams from 15 universities successfully advancing to the finals. During the 36-hour finals, the 21 teams completed a total of 1754 submissions, of which 1,425 were in the debugging test channel of the hidden test cases released in the finals. Of the total 329 standard test submissions in the finals, 259 submissions passed all test cases. The result of the finals is remarkably improved compared with the preliminary. Take test case `bitset3` as an example, the fastest result in the preliminary was 4.34s, the slowest was 66.76s and the average was 19.93s. However, the fastest result in the finals for this test case was 0.9s, the slowest was 43.78s and the average was 12.40s. 3 teams' compilers outperformed `gcc -O2` on the given 10 performance test cases.

Final Defense. Due to COVID-19, all 21 teams remotely joined the online video meeting to conduct their defenses, while the 10-member review expert group gathered physically. When each team was defending, no other teams were allowed to be present. Each team was required to give a 7-minute presentation to quickly introduce their own team and the characteristics of their compiler design, including the overall structure of the compiler, IR design and key innovations in the optimization technology implemented. The expert group asked questions for each team, the Q&A last 8 minutes in general, but some last up to 16 minutes. Many advanced optimization techniques such as SSA, multithreading, SIMD, and interprocedural optimizations (IPO) have been seen in the design of student compilers, which are usually not within the scope of undergraduate compiler courses. The results show that all teams have an extensive understanding of the basic principles, architectures and optimization techniques of building a modern compiler for a given language and a specific hardware platform.

3.3 Effectiveness of the Competition

In the competition, 21 of 72 teams passed all functional tests and entered the finals. 18 of 21 teams implemented optimizations showing significant speedup over gcc -O0. What's more, top 6 teams achieved better performance on the given 5 performance test cases in public than gcc -O2. Considering the limited time and the impact of COVID-19, this result was really impressive. We would also like to highlight a few interesting points in this competition:

1) The champion team used a three-level IR, enabling them to perform vectorization and thread-level parallelism on the code. Their compiler achieved a speedup of up to 4.08× than `gcc -O2` on 5 performance test cases in public.

2) Although most of the teams were junior students who had just finished the compiler course, two of the 21 final teams consisted of sophomore students who had never learned a compiler course.

3) SSA was widely used in almost all final teams, partially attributed to the wide adoption of LLVM.

4) Some optimizations were designed for test cases and lacked of systematic treatment.

5) Many teams felt that this competition was the first time they turned the theories learned in their compiler courses into real compilers. This will not only help them know more about the compiler, but also improve their skills in general problem solving.

In summary, the competition result is far beyond our expectation. We hope that the competition would encourage compiler course teachers to adopt more optimization-related content into their courses, and motivate students to learn and practice compiler techniques. We plan to improve the test cases into more realistic and complete ones in the next year's competition.

3.4 Experience and suggestions from Students

In order to have a deeper understanding of the situation and feelings of the participating teams, we made a questionnaire and further collected 54 questionnaires from 31 teams.

Table 3: Some statistical information of the questionnaire

(a) Grade Distribution

Grade	Freshman	Sophomore	Junior	Senior
Percentage	0%	44.4%	46.3%	9.3%

(b) Competition satisfaction distribution

Score	Very Bad	Poor	Fair	Good	Excellent
Percentage	0%	3.7%	29.6%	55.6%	11.1%

(c) Students who learn compiler knowledge through the competition

Knowledge	Parsing	Build AST/IR	Build SSA	Write IR Pass	
				Intraprocedural	Interprocedural
Percentage	40.7%	46.3%	35.2%	44.4%	29.6%

Knowledge	Register Allocation	Inst.Selection/Schedule	ARM Assembly
Percentage	66.7%	46.3%	61.1%

Table 3 shows the grade distribution of the participating students and their overall evaluation of the competition. It can be seen that over 90% of the students are sophomores and juniors, and 66.7% of the students were satisfied with the competition experience.

3.4.1 *Characteristics of Student Compilers.*

Through the questionnaire, we learned about the technical characteristics of the student compiler developed by the teams.

1) *Parsing method.* 44.44% used Flex + Bison to generate parser, and 42.59% directly wrote a recursive descent parser. The latest ANTLR parser generator [5] was rarely used, indicating that very few universities introduce new parsing techniques. It is worth noting that 2 teams implemented a parser generator on their own.

2) *IR design.* 25.93% designed hierarchical IR and 51.85% used flat IR. 40.74% of teams were inspired by LLVM IR, indicating that the LLVM open source compiler infrastructure [4] has had a certain influence among students. It is noticeable that one team imitated the Maple IR of the Ark compiler that was open sourced in 2019.

3) *Back-end design.* 33.33% introduced Machine IR. LLVM TableGen has not been imitated since solving the task of this competition does not require such complicated design. Most teams had their own way to implement code generator.

3.4.2 What did participants learn from this competition. The questionnaire also investigated the compiler knowledge of participants before and after the competition to find out what they learned.

Before the competition, 64.81% of participants took the compiler course, among them 51.43% were taught IR optimization approaches. However, only 14.29% of compiler courses involved optimizations that were more complex than dead code elimination (DCE) and constant propagation, so the IPO was rarely mentioned.

Most compiler courses included experiments: 85.71% constructed lexer and parser, 42.86% built AST and IR, 28.57% involved IR optimizations, and 37.14% included code generation. Both SSA form and register allocation were not required in any course.

Table 3(c) shows the percentage of students who learned various compiler knowledge through the competition. Almost all participants agreed that they learned a lot from this competition. We also found that some participants felt that learning from the competition was not efficient. They sometimes went in the wrong way, wasted much time, or got stuck on some frustrating bugs.

3.4.3 Difficulties and Suggestions. Through the questionnaire survey, we found that the hardest part of the compiler implementation thought by students was register allocation. They often took a week to a month to complete. In the final student compilers, the register allocation implementations were still buggy, causing at least half of the failure cases. Among the teams that realized register allocation, most of them chose graph coloring approach to solve register allocation problem, which required building interference graph based on live range of variables. After further investigation, we found reasons that made register allocation hard to implement:

1) *Locating bug is hard*: Students often mistook register allocation bugs for other bugs. For example, when they found that disabling an IR pass did not trigger the bug, they mistakenly believed that the pass had an error, which was actually caused by register allocation. This misjudgment would lead to a lot of wasted debugging time.

2) *Debugging is hard*: Even if the bug was known in register allocator part, it was not easy to find out which variable was incorrectly colored and which edge in interference graph was missing. This required using GDB to trace the generated ARM program, which was not easy for participants who were not familiar with GDB.

3) *Constructing a failing test case is hard*: Some teams managed to pass all public test cases, but failed on some hidden test cases. They had to try to construct test cases by themselves to reproduce the bug for debugging, but these attempts often failed. This was because the trigger conditions of such bugs were usually complicated and strict. Intentionally written test cases would probably not reproduce such bugs, only massive test cases can coincidentally trigger them.

Another reason for difficulty in debugging was the gap between the development platform (x86) and the evaluation platform (ARM), which bothered inexperienced teams. To improve experiences of participants, we should consider: 1) provide more technical training and tutorials for inexperienced teams; 2) add more functionality test cases and provide fuzz testing tools for SysY compiler; 3) provide functional evaluation tools (based on QEMU) that can run on x86.

4 RETHINKING THE COMPILER COURSE

With competition and questionnaire survey analysis, we rethink the question: what should we teach and practice in the compiler course? We believe that: 1) The front-end materials should be greatly reduced, and the introduction to modern parsing techniques such as ANTLR should be added. 2) IR plays a key role in modern compiler design. New IR concepts, such as multi-level IR and SSA, should be introduced and explained. 3) Code generation and register allocation deserve more time in courses to allow students understand and write efficient back-ends for different processors. 4) It is necessary to improve the oversimplification of register allocation and code generation in existing textbooks, and further combine examples to introduce relevant principles used by modern compilers.

Are the above suggestions realistic in undergraduate compiler courses? We believe they should be learned through labs. We suggest that labs need to include writing data-flow passes and lowering passes between levels of IR. For code generation and register allocation, some code reading exercises can help students understand the register allocation and code generation techniques in the industry compiler, without too much programming efforts. LLVM table-gen lab can be an optional one for back-end portability.

In order to allow more students and universities to grasp these contents, it is very important to carefully design a set of lecture materials and labs. In addition, the use of new features of C++ and debugging techniques such as GDB also requires tutorials.

5 CONCLUSION

In the paper, we systematically summarize the first National University Student Compiler Design Competition in China. To our surprise, 72 teams from 47 universities signed up for the first competition, slightly higher than the 70 teams from 45 universities participating in the first CPU design competition. Although there were various problems caused by inadequate consideration and lack of experience in the competition, the participating teams have shown great enthusiasm and actively cooperated to make all aspects of the competition progress in an orderly manner. The performance of the compilers developed by some teams was also surprising, showing that the potential of the students should not be underestimated. The results of the competition and questionnaire analysis provide guidance for better preparation for the next year competition and also the improvement of the compiler course in some universities.

Acknowledgment This work was partially funded by New Engineering Research and Practice Projects of Chinese Ministry of Education (No. E-JSJRJ20201301), USTC Undergraduate Quality Engineering Projects (No.2020xylkc001) and Anhui Provincial Major Teaching and Research Projects (No.2017jyxm0005).

REFERENCES

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *CGO 2019 (CGO 2019)*. IEEE Press, 193–205.

[2] Raspberry Pi Foundation. cited March 2020. Raspberry Pi 4 Tech Specs. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[3] GigaDevice. cited Jan 2020. GD32 RISC-V Microcontrollers. https://www.gigadevice.com/products/microcontrollers/gd32/risc-v/.

[4] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04*. Palo Alto, California. http://llvm.org/

[5] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *OOPSLA '14*. ACM, New York, NY, USA, 579–598. https://doi.org/10.1145/2660193.2660202

[6] Alex Radermacher and Gursimran Walia. 2013. Gaps Between Industry Expectations and the Abilities of Graduates. In *44th SIGCSE*. ACM, New York, NY, USA, 525–530. https://doi.org/10.1145/2445196.2445351

[7] Alex Radermacher, Gursimran Walia, and Dean Knudson. 2014. Investigating the Skill Gap Between Graduating Students and Industry Expectations. In *36th ICSE Companion*. ACM, New York, NY, USA, 291–300. https://doi.org/10.1145/2591062.2591159