Continuous Integration in SoC Design: Challenges and Solutions

Liu Wei, Li Jianjun, Yang Liangfeng, Li Liang
SDTECH Information Technology, Beijing, China
{liuwei, jianjun.li, feng, liang.li}@sudoinfotech.com

***Abstract*-This paper introduces the implementation of Continuous Integration (CI) in large-scale System-on-Chip (SoC) design. We discuss the challenges encountered, our solutions, and the resulting automated CI system for SoC design and verification code. Our approach leverages both the modern continuous integration practices in software development and the high-performance computing (HPC) environments and existing regression scripts commonly used in chip design and verification, enabling the implementation of a user-friendly yet powerful continuous integration system.**

I. INTRODUCTION

Continuous Integration (CI), as a practice that allows developers to frequently integrate code changes into a shared code repository and perform automated builds and tests, has gradually been applied in the hardware development field. Although many companies have introduced continuous integration into their chip (System-on-Chip, SoC) design and verification workflows, implementing an efficient hardware continuous integration system still faces many challenges due to several differences between hardware and software development.

These challenges are mainly reflected in the following aspects: First, chip design and verification rely on job scheduling systems in high-performance computing (HPC) environments to handle build and simulation work, and this unique infrastructure and workflow are not compatible with mature continuous integration tools. A common approach to address this issue is to configure additional dedicated computing resources for the continuous integration system. However, this solution faces a dilemma: if the additional resources are limited, it becomes difficult to run automated tests on a large scale in parallel; if a large amount of computing resources is added, it leads to excessive costs. To resolve this dilemma, there have been some attempts in the industry, some running tests once at scheduled times, some combining multiple merges to run tests once, and some saving code merge events to a queue and then running jobs one by one from the queue, but these solutions more or less have problems such as poor real-time performance and system complexity. Second, many modern continuous integration tools are deeply integrated with Git or even developed directly based on Git. If Git is not used, some basic components need to be developed independently. We choose to adopt popular tools from the software field as much as possible instead of reinventing the wheel, which not only allows us to continuously follow the latest concepts and methods in the field of continuous integration but also minimizes custom development and maintenance costs, avoiding the need for dedicated maintenance personnel. However, using Git for chip design code management brings new problems, including performance degradation in large code repositories, binary file merge conflicts, and large file management challenges. These problems do not exist or are not obvious when using perforce or subversion. For example, perforce is very efficient and powerful in supporting large repositories and large files. Both perforce and subversion support binary file locking.

To implement a continuous integration system that is both easy to use and powerful, we connected the continuous integration system with the HPC environment(cluster of special compute nodes and storage to allow jobs to run in parallel), allowing the use of HPC computing resources in continuous integration, and efficiently utilizing them: In the early stages of chip development, when regression testing has not yet begun, HPC resources are not fully utilized, while the CI system requires a large amount of HPC. As development progresses, code merges decrease, CI demands decrease, but large-scale regression testing begins to start up, requiring more HPC resources. This complementary scheduling makes the implementation of CI possible without additional HPC resources - even if the continuous integration system is designed to perform automatic testing before and after each code merge. This way, the continuous integration system will be simpler. To use Git smoothly for code management in large SoC projects, we upgraded our IT infrastructure, changed the topology of code storage and access, and implemented a series of special Git configurations and methods, which will be detailed in subsequent sections.

II. CI SYSTEM ARCHITECTURE

To implement CI for SoC design verification, first, there must be a set of build/simulation scripts that can provide parameterized commands to start the build and/or simulation of different tests. The CI system will call this script to implement the automatic testing function. For example, running sim -b a_test will compile the design and verification code corresponding to a_test, generate an executable file, and sim -s a_test can run this simulation. If it ends normally, the simulation passes, and the exit code of this command is 0; an exit code of non-0 indicates task failure. The CI system will determine whether the test passes based on this exit code and display it in the user interface, thereby deciding whether the code should be merged. Most companies may already have similar tools, generally called regression scripts.

On this basis, the following components are also needed:

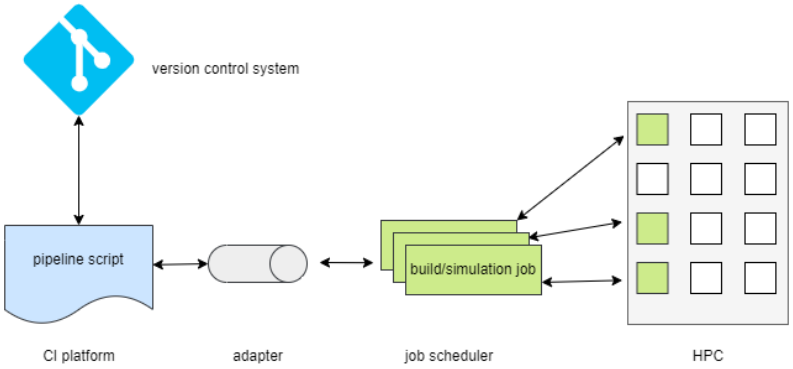

Figure 1. Basic components of CI flow.

Moving forward, I will outline the capabilities of each component and suggest some viable tools for each. In the subsequent chapter, I will introduce our choices, as well as more details.

Version Control System (VCS), storing chip design code, verification test cases, and configuration files. Webhook trigger: automatically triggers CI process after code submission; branch strategy management: supports feature/main branch workflow, implements code merging; code review integration. Such as GitLab, Bitbucket, Gitea, Azure repo, etc. (This article only focuses on workflows using git for code management, non-git version control systems are not listed).

Continuous Integration Platform (CI Platform), providing a pipeline script file for users to implement, executing tests pipelines. Job monitor: real-time monitoring of verification task status, displaying the process of command execution in real-time on the webpage, including complete output; pipeline execution: CI includes an agent (or runner) that executes commands in the pipeline script, generally needing to be installed on a separate host, to clone the current branch's code to be verified after the automatic test is triggered, and execute commands in the pipeline (such as sim -b a_test); report results: notify VCS whether it can be merged, and promptly notify relevant personnel of verification results. Continuous integration platforms include: Gitlab CI, Bamboo, Buildboot, Jenkins, Azure pipeline, etc.

Job Scheduling System, each pipeline will produce multiple jobs that need to be run in parallel, these jobs will be handed over to the job scheduling system, managing and distributing to HPC, such as Slurm, LSF, PBS, etc.

Command Adapter, connecting the CI system and job scheduling system, handling the gap between the CI system and HPC system, including storage structure, user permissions, etc., and converting CI platform commands (such as sim -b a_test) into commands required by the job scheduling system (such as srun -n 1 sim -b a_test). Such as Jacamar CI (for Gitlab), or develop your own based on the CI system and job scheduling system used.

HPC, using the original existing HPC.

This way, the basic workflow is as follows:

1. Developers submit code to the version control system
2. The Webhook of the version control system triggers the CI pipeline
3. The CI platform parses the configuration, prepares the verification environment, and the CI runner/agent clones the code of the current branch
4. The command adapter converts test tasks into HPC jobs
5. The scheduling system allocates jobs to appropriate compute nodes
6. HPC executes verification tasks and returns results
7. Results are returned through various levels back to the CI platform

As described in the above process, each pipeline run needs to clone the code once, so the response speed of git commands is very critical. In many IC design companies, users' working directories are mounted network drives through NFS or other network file systems, rather than real local hard drives.

Using Git to clone code or execute any git commands in such working directories will have serious performance issues, so if Git is used for version management, it must be ensured that users' Git repositories are stored on the local hard drive of the host executing Git commands. To meet Git's requirement for local hard drives, we upgraded the IT infrastructure: two servers are needed, one for installing the version control system (such as Gitlab), and one for installing the CI runner (such as Gitlab Runner), command adapter (such as Jacamar CI) and job scheduling system, with the final commands executed on the HPC of the server cluster. So our hardware system became like this:

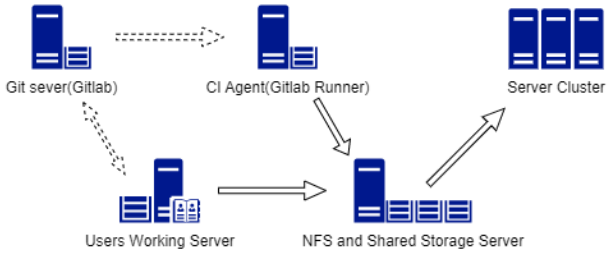

Figure 2. Hardware topology for CI flow.

The Git server and CI runner, as well as the user work server, all have their own local storage for storing Git repositories locally. Among them, the local storage of the CI runner and user work server needs to be shared with other public storage file servers through network file protocols such as NFS to form the shared file system of the server cluster, because the server cluster needs to use the code of the CI runner and user work server for build and simulation, so they need to be mounted on the server cluster, as shown by the solid arrows in the figure 2. The dashed arrows in the figure represent directly using Git clone or push code through Ethernet to synchronize between local Git repositories and the Git server. In this way, it not only utilizes the high-performance computing capabilities of the server cluster but also meets the requirement of Git repositories for local hard disk storage.

III. CI PROCESS TO ENSURE MAIN BRANCH IS ALWAYS AVAILABLE

Our main goal is to ensure that the main line code always has no basic errors and maintains a known good state. To achieve this, our strategy is to automatically run a set of tests before each code merge, only merging into the main line if all tests pass, and then running this set of automatic tests again after merging into the main line to ensure the correctness of the code after merging. This set of tests is called CI tests, covering multiple build and simulations at IP, subsystem, and SoC levels to ensure the basic functionality of all components. Multiple users can trigger CI tests simultaneously, and each CI test contains multiple build and simulation tasks that need to be executed in parallel, thus requiring a large amount of computing resources.

The biggest benefit of executing CI tests every time code is merged into the main line is the rapid discovery of errors. Developers can fix errors while their memory of the errors is still fresh, reducing the workload of task switching and review. At the same time, most bugs are fixed before they spread to others, thus reducing the workload of more people debugging due to bugs spreading to other engineers. To automatically implement this process, we designed the following workflow:

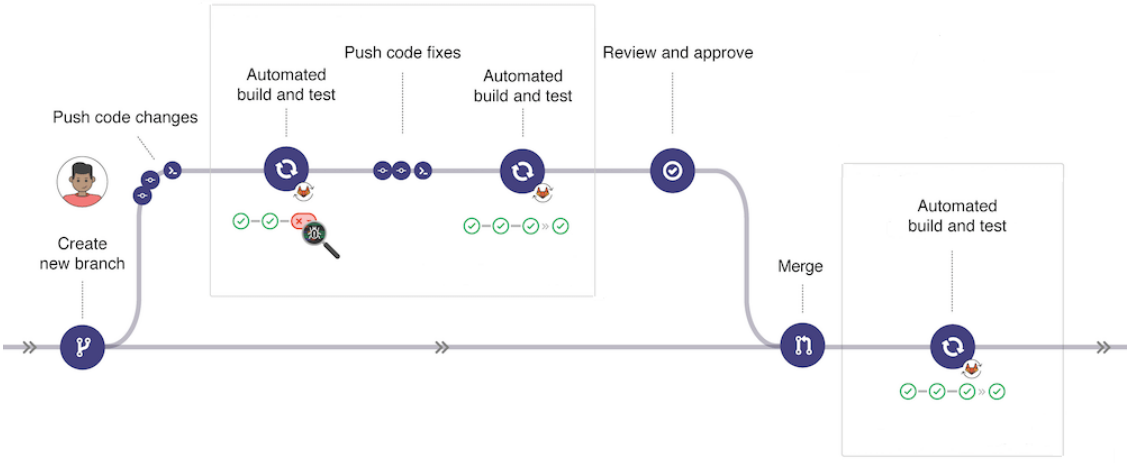

Figure 3. Workflow for chip design and verify based Git.

All work starts from the main branch and ends by returning to the main branch. The main branch does not accept code pushes by default, only accepting merges.

Before RTL freeze, only one long-term branch, main, is maintained. Whether design engineers or verification engineers, each time a feature is added or a bug is fixed, a new branch needs to be created. This branch is only for one person's use, and to complete and merge into the main branch in a timely manner, the planned workload for each branch should not be too large.

After completing work locally, push to your own remote branch (can push multiple times), then open a Merge request on the webpage, which automatically triggers CI tests. CI test jobs include two types, one is build, and the other is simulation. One build may correspond to more than one simulation, verifying different functions through different simulation options. At the beginning of the test, all builds work will be carried out in parallel, each build will have its own separate directory, and if this build corresponds to multiple simulations, new sub-directories will be created within the directory for each simulation. Build will use HPC through the job scheduling system. As soon as the current build ends, all simulations corresponding to the current build begin in parallel, also using HPC through the job scheduling system. Each build and simulation is treated as a separate job, and the output results will be returned in real-time to the webpage of the git server (such as Gitlab) for easy debugging. After the job ends, the execution results (Pass or Fail) of all jobs will be summarized together. If even one does not pass, the current branch cannot be merged into the main branch. At this time, the author needs to find the reason and push the code again, otherwise, it cannot be merged into the main branch. CI tests are automatically triggered after pushing the code again. It should be noted that before pushing code, developers should perform a git rebase main operation locally, so that push after merging the latest main branch code, to maximize avoiding automatic test failures after code merging.

CI tests are automatically triggered again after the code is merged into the main branch because merging may cause tests that passed on one's own branch to fail, although the probability of this happening is small. At the same time, doing this can also promptly discover problems in the main branch and promptly understand which merge brought about the problem.

To speed up turnaround, automatically triggered tests do not dump waveforms, but for some failed tests, it may be difficult to analyze the reason just through logs, and waveform dumps are needed. In CI tests, including when the job scheduling system submits, they are executed with the identity of the person who triggered the current continuous integration pipeline, and the generated files' owner is also the current trigger. This way, if the automatic test fails, the trigger can go to the directory of the failed automatic test case according to the directory displayed in the log, and use the build target files already generated by the automatic test to start simulations with dump parameters, thus saving the work and time of rebuild. Using the identity of the trigger to submit jobs through the job scheduling system has another advantage: large chip design companies generally allocate different HPC resources to different teams and limit the number of jobs running simultaneously. Submitting with the trigger's identity can keep the use of HPC computing resources within the company's policy requirements.

In chip verification tests, some variables in some tests will obtain values randomly, which may be different modes, switches, etc. This way, tests may pass once with the same code but fail another time (because it randomly got another mode, etc.). Thus, tests triggered by merge requests may pass, but tests triggered after merging into main may fail again. To ensure the reliability of CI test results, specified random seeds were used in all CI test cases.

In actual work, some multi-person collaboration problems may be encountered. For example, a feature design has been implemented, and there is a CI test that automatically tests it every time the code is merged. Now the DE (Design Engineer) needs to change this design, and if the verification code is not modified accordingly, the test will inevitably fail, and the DE has no obligation to modify the verification code. The automatic test fails, so the design code cannot be merged. At this time, a Git feature called cherry-pick needs to be used. As long as the DE pushes code to his remote branch, DV (Design Verification Engineer) can get the DE's code through cherry-pick on DV’s local branch, thereby merging the DE’s code into main. Of course, Git will smartly handle different code authors and modification records, and there won't be problems like the DE's code being modified by DV.

IV. CI IMPLEMENTATION BASED ON GITLAB

After comparing multiple CI tools, we chose Gitlab as our git server, Gitlab CI as the CI platform, and Jacamar CI as the adapter. We didn't choose the commonly used Jenkins regression manager tool mainly because it's more complex and even requires dedicated maintenance personnel. Jenkins uses Groovy (Jenkinsfile) to write pipelines, while Gitlab CI, Azure DevOps, GitHub, etc., all use YAML, making Gitlab CI easier to use and maintain. Recent discussions among developers on platforms like Reddit suggest that GitLab CI has been increasingly admired as a CI/CD tool in recent years [4][5]. Gitlab CI and Gitlab as a whole have very well-integrated functions, requiring no development on our part.

Additionally, Gitlab has an open-source third-party adapter, Jacamar CI, which solves the issue of connecting with HPC and supports multiple job scheduling systems such as Cobalt(qsub), LSF(bsub), PBS(qsub), and Slurm(sbatch). These components can help us implement an easy-to-use automatic testing process with very little code, and they are all open-source and free (we use the community version of Gitlab), which can save a lot of costs for the company.

The data flow of the entire system is shown in figure 4.

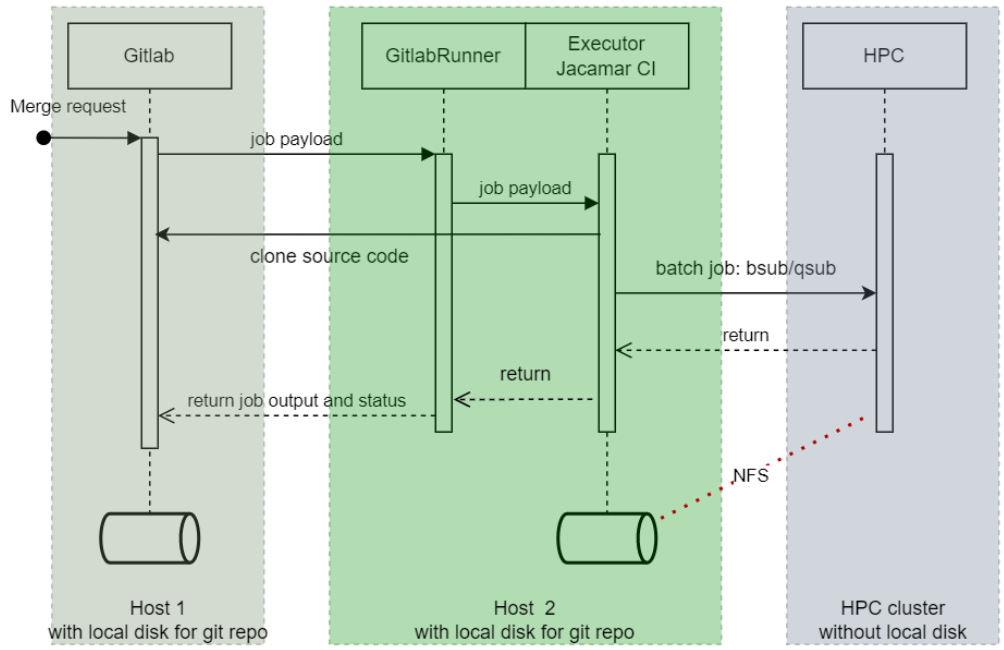

Figure 4. CI Implementation and date flow.

After users push code from their local repository to the remote branch, triggering a Merge request, Gitlab will automatically notify the Gitlab Runner to clone the code of the trigger's branch. The cloning location is configured by us in Jacamar CI, a directory on the Runner's local disk named with the current pipeline's ID. Jacamar CI will submit jobs to HPC, and HPC will run our modified regression script. The parameters of the regression script specify whether it's a build or simulation, which test it is. The command line output results of build or simulation will be returned to Gitlab's webpage in real-time for easy debugging. After the regression script execution is complete, if the exit code returned by the script is non-zero, it indicates an error occurred during the run, and Gitlab will display the current test as Fail. If the script's exit code is 0, it indicates the current test Pass. After a pipeline is successfully executed, the cloned code and build artifacts produced by CI tests are no longer needed, so we have a timed script executed as the root user that deletes directories older than 24 hours every day at noon. Additionally, because our Gitlab webpage displays complete logs, continuously accumulating log files will occupy a lot of storage, which also needs to be deleted periodically.

A. Customize a regression script and implement a CI script

Our regression script is a Python-developed tool used for build and simulation test cases. It can regress many test cases at once, but in the CI process, it is used to run tests for individual cases. For example, if we need to build and run a test case named example_test_a, the command is sim -t example_test_a. If the build and simulation pass, the exit code is 0; otherwise, it's non-zero. GitLab CI relies on this return code to determine whether the current test has passed or failed. Before simulating a test, we must first compile to generate an executable file, which we call a build. This build has its own filelist and unique UVM_TEST, as well as all options needed for the build. Suppose there is a build named example_build. Under this build, multiple tests can be generated by setting different simulation parameters (sim options). Let's assume that in addition to example_test_a, there is another test called example_test_b. Our regression tool supports executing build and sim (simulation) separately. We can compile and generate executable files using sim -b example_build, and simulate based on the previously compiled executable file using sim -t example_test_a -so. Different simulation options are managed through yaml files, with each test corresponding to different simulation options. After running the regression script, it will automatically parse the simulation options in the corresponding yaml file to generate simulation commands. Build options are also stored in yaml files. Figure 5 is a simple yaml file showing one build and two tests.

```yaml
example_build:
    extend: prj_base_build
    type: build
    tb_path: subsys/example_tb
    compile_option:
        -timescale=1ns/1fs: +
        +define+EXAMPLE_MACRO: +
example_test_a:
    extend: example_build
    type: test
    uvm_test: example_test
    sim_option:
        +topology=a: +
    tag:
        - ci
        - nightly
example_test_b:
    extend: example_build
    type: test
    uvm_test: example_test
    sim_option:
        +topology=b: +
    tag:
        - ci
        - nightly
```

Figure 5. YAML used by regression script

The tb_path key-value pair determines the folder of the current build, which contains the testbench file, design and verification filelist for the current module, sequence for the current module, testcases file, and so on. Each module has such a build, and when the regression script is run, it creates subdirectories for each build, and within the build directory, it creates simulation directories for each test, allowing for parallel builds and parallel simulations. In this example, there is a tag key-value pair. When the script is used for regression, the command sim -tag sanity will compile and simulate in parallel all cases that contain the sanity tag (the CI process does not use this command). Each build and simulation need to be submitted separately to the HPC, and the simulation job must wait for its corresponding build to succeed before it can be submitted, so it is necessary to generate dependencies between builds and simulations. For this, we added a command sim -genci -tag ci to the regression script. This command generates build and simulation commands for all test cases that include ci in their tags, and outputs them in the format of GitLab continuous integration scripts, appearing as individual jobs with built-in dependencies. We use GitLab CI's parent-child pipeline mechanism, running the sim -genci -tag ci command in the parent pipeline to generate a child pipeline script containing build and sim commands, and then completing the CI tests in the child pipeline. As show in figure 6.

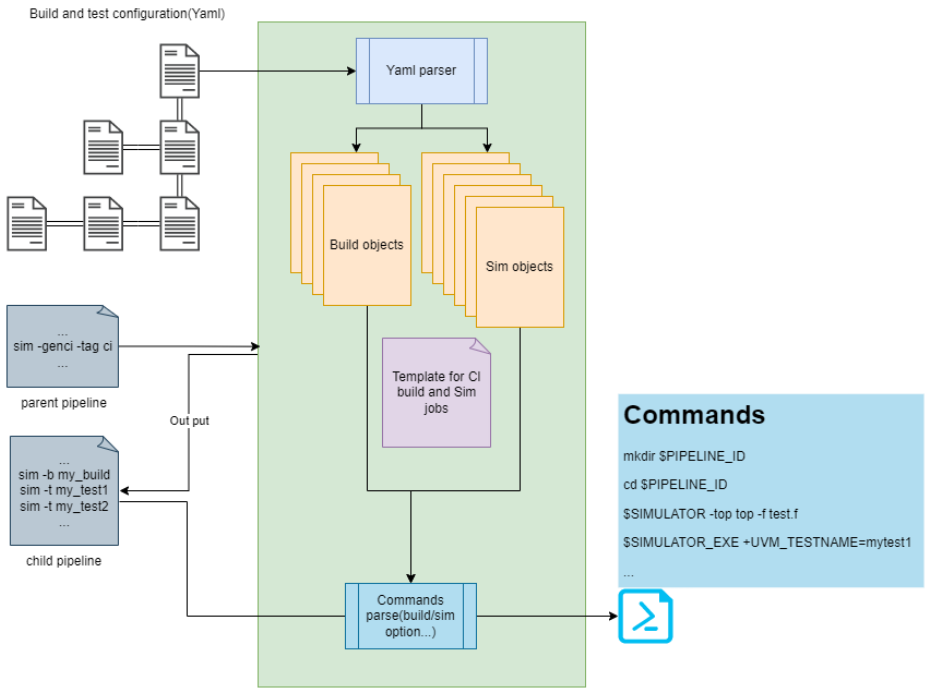

Figure 6. Gitlab CI and enhanced regression script implementation.

B. Gitlab Runner and Jacamar CI Configuration

Gitlab Runner provides a configuration called custom executor, which allows users to implement their own executor. We took advantage of this feature and specified Jacamar CI as the executor in the Runner's configuration file /etc/gitlab-runner/config.toml:

In the Jacamar CI configuration file, setting custom_build_dir to true allows the CI test, when run, to obtain the pipeline ID from the current pipeline via ${CI_PIPELINE_ID}, create a new directory, clone the code, and simulate. The executor is used to specify the job scheduling system, and setting group_permissions to true allows users in the same Linux group to view each other's logs and waveforms.

V. CHALLENGES OF USING GIT IN SOC PROJECTS

Without any adjustments, simply using git directly can indeed encounter problems:

1. As the number of files in the repository increases, the execution of git commands will become very slow.

2. Git requires checking out the entire repository before modifying and committing, but actually some people's work only involves one or two directories.
3. Git manages large files, and the historical versions of large files will always exist in the git repository. After modification and submission, the size of the repository grows exponentially.
4. Git's merge cannot resolve conflicts of binary files, and it does not have a locking feature like svn.

The first problem has caused us a lot of trouble for a long time. The main reason has been mentioned before. That is, the location of the Git repository needs to be switched from NFS to a local hard disk. In fact, we have completely lost the feeling of command latency after making this one adjustment. If the repository is still large and there are performance problems, you can use git's scalar [1] and sparse checkout to further optimize. It will be further explained below.

The second problem indeed existed in the historical version of git, but git 2.38 and later support the scalar command, which supports sparse checkout and can set the directory to be checked out, so other directories will not appear on their own disk. Moreover, scalar will obtain the storage objects of remote branches regularly in the background. It is almost real-time when users use commands such as git fetch or git pull. Scalar is a series of upgrades developed by Microsoft for managing large repositories with git. It was born out of Microsoft's production solutions for many years and was merged into the official git repository in version 2.38. It is very simple to use. Just replace the git clone command with scalar clone.

The third and fourth issues can be perfectly resolved using the same tool that comes with git. Git LFS, LFS allows large files to be stored in a proprietary directory specified by GitLab. In the git code repository, it only stores an index of a large file and checks out a version that needs to be used only when it is used.

Popular monorope recommends managing all project-related code in one git repository, but it is more appropriate for third-party libraries used in IC design to be placed outside the repository to reduce the size of the repository, such as network disks mounted by NFS. At the same time, maintain a filelist containing the version path of the third-party library in the Git repository. After the third-party library is updated, you can see the version change history in the Git repository.

As for the merge conflict of binary files, LFS supports locking and unlocking, which is very convenient to use.

VI. IMPLEMENTATION IN A REAL PROJECT

The CI pipeline consisted of 19 different testbench builds and 23 simulation jobs, completing within an hour, while traditional manual integration and testing in similar projects typically required one day per cycle. At the peak of development, up to 40 pipelines were triggered daily, with 7 to 8 pipelines running simultaneously.

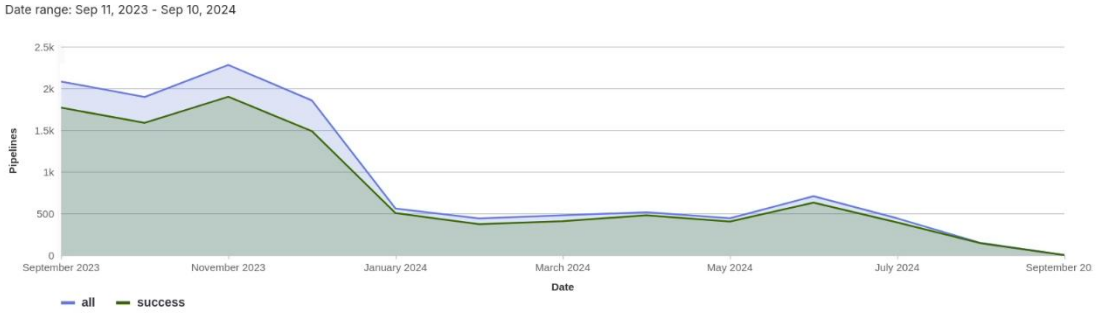

Figure 7. Pipelines charts last year.

Throughout the 15-month development period, our team of 41 developers performed 3,303 merges, triggering 8,213 pipelines, with 6,743 successful completions and 1,344 failures. The CI system demonstrated significant improvements in bug detection and resolution efficiency: developers typically identified root causes within 5 minutes of a CI failure, compared to over 30 minutes in traditional flows, with some cases extending beyond 24 hours.

Based on historical data from previous projects using traditional methods, where each integration issue consumed an average of 2.5 hours, the 1,344 early-caught failures represented over 3,360 hours of saved engineering time.

VII. CONCLUSION

We utilized Gitlab CI, combined with our custom regression scripts and configurations, leveraging the existing HPC system, to implement a lightweight and efficient CI system. Unlike traditional CI systems

that require extensive custom setups or additional resources, our approach uniquely optimizes the use of existing HPC infrastructure, making it particularly suitable for large-scale chip design projects.

This system demonstrates the following advantages:

1. Rapid error detection, allowing developers to fix issues while their memory of the code is fresh.
2. Reduced bug propagation, as most bugs are fixed before spreading to other team members.
3. Maintaining the integrity of the main branch, ensuring that the main branch is always in good condition: all basic feature tests (CI test) have always passed.
4. Optimized HPC utilization, eliminating the need for additional hardware investments.

This system significantly enhances our development process, enabling faster iterations and more reliable outcomes. It facilitates early detection of issues and reduces the overhead of manual testing and integration.

REFERENCES

[1] Derrick Stolee and Victoria Dye, "The Story of Scalar," [Online]. Available: https://github.blog/open-source/git/the-story-of-scalar. [Accessed: Dec. 3, 2024].

[2] GitLab, "GitLab Documentation," [Online]. Available: https://docs.gitlab.com. [Accessed: Dec. 3, 2024].

[3] ECP-CI, "Jacamar CI Documentation," [Online]. Available: https://ecp-ci.gitlab.io/index.html. [Accessed: Dec. 3, 2024].

[4] Reddit users, " Choosing a CI/CD tool for your product,"[Online]. Available: https://www.reddit.com/r/devops/comments/11cd32e/choosinga_cicd_tool_for_your_product/. [Accessed: Dec. 3, 2024].

[5] Reddit users, "Which CI tool? " [Online]. Available: https://www.reddit.com/r/devops/comments/1de43le/which_ci_tool/. [Accessed: Dec. 3, 2024].