WHITE PAPER

Turbo-Charging Continuous Integration and Development Flows with Virtual Prototypes

Author

Sam Tennent
R&D Engineer, Sr. Staff
Synopsys

Overview

As leading-edge semiconductors have evolved into programmable multi-core system-on-chip (SoC) devices, embedded software development has become a dominant factor in project cost and schedule. Virtual prototypes are widely used for writing and debugging embedded code so that the programmers do not have to wait for hardware prototypes. In recent years, the complexity of embedded software has grown tremendously, resulting in larger programming teams and the significant challenge of integrating their code together into working systems. This white paper discusses using a combination of virtual prototypes to replace physical hardware and a continuous integration/continuous deployment (CI/CD) flow not only to "shift left" the initial project schedule but also to enable a faster and more efficient ongoing embedded software development flow with qualified releases and higher code quality.

Virtual Prototypes vs. Physical Hardware

Unlike higher-level applications, embedded software is, by its very nature, close to the SoC hardware. It falls into a category that also includes device drivers, BIOS, boot code, and firmware. It directly monitors and modifies hardware resources such as registers and memories. Embedded programmers have traditionally waited until hardware prototypes were available in the bring-up lab to test the hardware-software interaction. However, physical prototypes cannot be built until all chips are back from the foundry. Starting embedded software development so late in the project delays time to market (TTM) unacceptably. Physical prototypes are costly, limiting the number of programmers who can work in parallel. It is also challenging to maintain physical prototypes and keep them in sync.

Because of these issues, virtual prototypes rather than physical hardware have become the preferred platform for embedded software development. Virtual prototypes are abstract software simulation models for the SoC and hardware system. They contain fast instruction set simulators (ISSs) for the embedded processors, capable of executing the same code that will run in production systems, and can interact with both virtual and real-world I/O devices. Virtual prototypes can be created early in the project, long before any chips are fabricated. This enables true hardware-software co-development and "shifts left" the project schedule dramatically. Any inconsistencies between the design and the embedded code are detected during the pre-silicon phase, greatly reducing the chance of a chip turn.

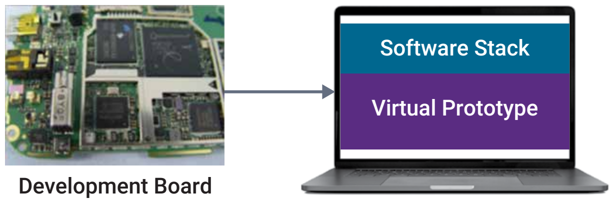


Figure 1: Evolution from physical to virtual prototypes

Virtual prototypes run on standard Linux or Windows platforms so they can be replicated quickly and easily at minimal cost. It is feasible for every embedded programmer to have access to a virtual prototype whenever needed. They are more flexible than physical prototypes since users can test responses to abnormal conditions by injecting virtual faults that would damage hardware. Virtual prototypes also offer better debug capabilities since access to internal behavior is not constrained by packaging or the setup in the bring-up lab. All these capabilities benefit embedded programmers not only in pre-silicon development but also post-silicon, in the ongoing evolution of embedded software. Regression runs of software tests and performance measurements are needed for every new software release. Virtual prototypes are a much better solution than physical hardware for running post-silicon regressions in parallel, testing for responses to faults, and debugging the results.

Continuous Integration and Continuous Deployment

In addition to developing on virtual prototypes, embedded software teams have adopted many best practices used by their programming colleagues. Among these is continuous integration (CI), a significant improvement from traditional software integration, in which programmers checked their code into the main repository only occasionally. Individual code segments often deviated significantly, leading to an integration process that was tedious and hard to debug.

CI is a much better approach to support parallel development by multiple programmers. Code changes are checked-in frequently, usually at least once a day. The system is re-built and tested immediately. There is much less opportunity for divergence, and any integration issues are discovered and debugged quickly during the build and test process. Code check-ins automatically trigger regression testing and often additional analysis such as linting or code coverage. When all tests and analysis complete successfully, the programmers may choose continuous deployment (CD), in which the new software is automatically released to all platforms. CD may be limited to the bring-up lab, or it may update all production platforms in the field, especially for software as a service (SaaS) and cloud applications. This ensures that all users have access to the latest version and keeps them all in sync.

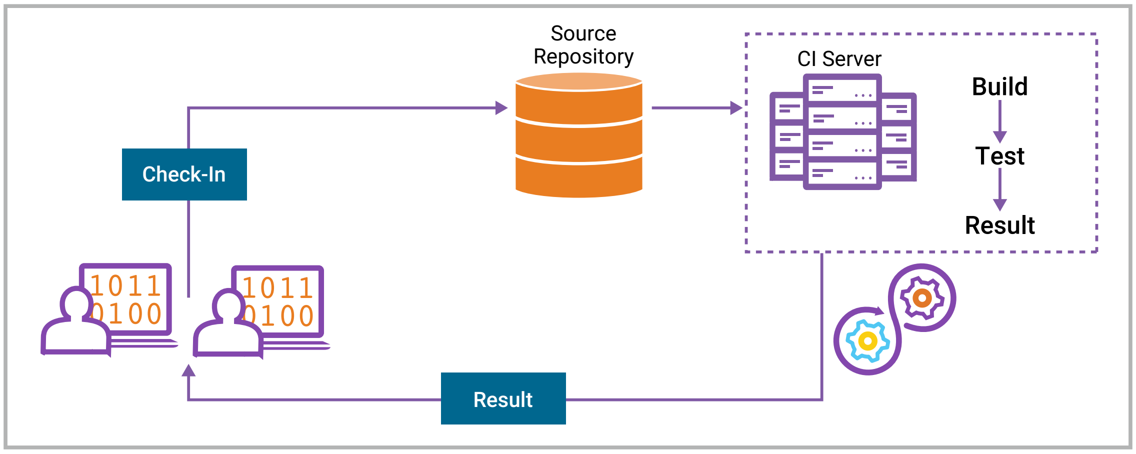


Figure 2: Continuous integration and deployment flow

Growing software content in SoCs, increasing software quality requirements, and ever shorter schedules are pushing embedded programmers to adopt CI widely. Using physical hardware boards to build a CI pipeline for embedded software is costly, cumbersome, and fragile. Instead, virtual prototypes can be used to easily run the automated integration regression tests on standard hardware that is easy and cheap to replicate. Localized CI/CD flows are helpful during pre-silicon development, and they have even more value in post-silicon regression tests as the embedded software evolves.

The Synopsys Solution

Embedded programmers wishing to move to CI/CD flows with virtual prototypes, including running regressions, have a well-proven solution available now. Synopsys Virtualizer™ provides the industry's leading solution for the development and use of virtual prototypes. This solution includes a family of pre-validated Synopsys Virtualizer Development Kits (VDKs) for several widely used processor and microcontroller architectures. Synopsys VDKs contain design-specific virtual prototypes, debug and analysis tools, and sample software. They are assembled and debugged inside the Synopsys Virtualizer integrated development environment (IDE), enabling deep insight into any issues uncovered. They support all types of code development, from applications and operating systems to embedded software.

Synopsys VDKs are frequently integrated into advanced CI/CD flows to create a productive pipeline for embedded software development, test, and debug. Regression testing and deployment benefit from the use of containers, "build once, run anywhere" executable software packages with everything needed to run quickly and reliably in multiple computing environments. Synopsys Virtualizer and Synopsys VDKs provide the capability to create a checkpoint at any time during a test and quickly return to that point in future simulations. This checkpoint/restore capability greatly speeds up test cycles by avoiding redundant work such as long initialization sequences. Synopsys Virtualizer and Synopsys VDKs also support pushbutton test parallelization across many instances of the virtual prototypes. Neither checkpoint/restore nor broad parallelization is possible with physical prototypes, so these capabilities provide even more incentive to adopt virtual prototypes and CI/CD for post-silicon regressions as the embedded software evolves.

Synopsys VDKs provide extensive and unique analysis and debug facilities, and integrate seamlessly with a wide range of leading-edge software development and test technologies, including:

- GitLab and Jenkins DevOps platforms
- Docker containers
- Kubernetes-based container orchestration systems

Example CI/CD Flow with Virtual Prototypes

A recent project demonstrates the value of the Synopsys solution when integrated with several of the platforms mentioned above. The design is a solid-state drive (SSD) using the Nonvolatile Memory Express (NVMe) storage access and transport protocol. Embedded software development is supported by the Synopsys NVMe SSD VDK, which ships to users with Synopsys Virtualizer. As shown in Figure 3, a Synopsys VDK system model is composed of two subsystems connected by a virtual PCI Express (PCIe) Bus. The Root Complex (RC) subsystem acts as a PCIe device and contains the system's applications processor. The Endpoint (EP) subsystem contains an embedded CPU, an NVMe controller, and a NAND flash controller connected to NAND flash models. Together, the two subsystems model an SSD as a PCIe endpoint.

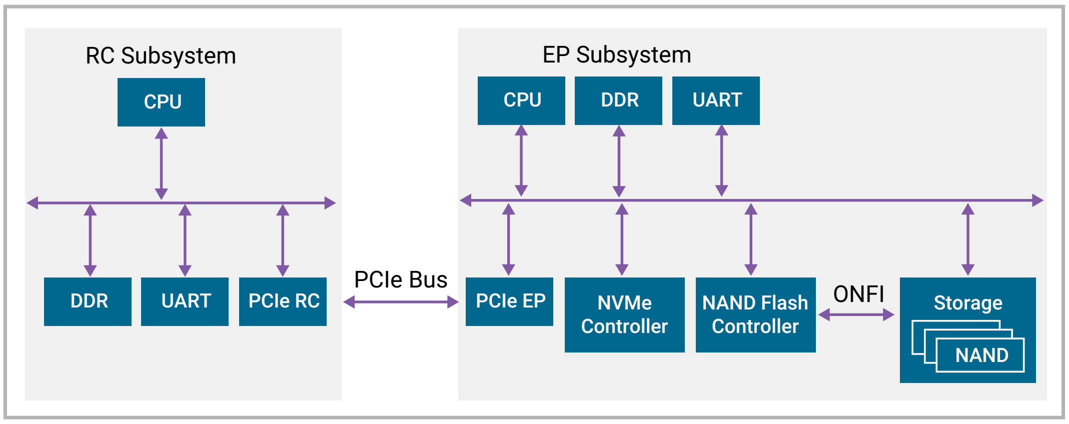


Figure 3: Synopsys NVMe SSD Virtual Development Kit

In terms of software in the virtual prototype, the application CPU in the RC subsystem runs Linux and a Linux application called IOzone that tests the SSD operation. The EP subsystem CPU is running the firmware controlling the SSD. This base metal application is implemented in C++ and carries out all the complex SSD management such as flash page handling, wear levelling, and garbage collection. This software is the focus of development and testing on the virtual prototype. GitLab is used for revision control of all the software and models. The GitLab registry stores the NVMe SSD VDK and the Runtime environment needed to execute it. These tend to remain stable over most of the project, and can be shared across multiple GitLab projects.

The Gitlab repository contains all the SSD firmware source files as well as the test script developed to test the code as it runs on the Synopsys VDK. Any changes in the repository are detected by the built-in GitLab CI/CD capability and used to trigger an automatic re-build and re-test. The regression tests are executed in Docker containers, and orchestration for these containers is provided by a Kubernetes cluster from Rancher. The typical flow for each embedded programmer within this environment is shown in Figure 4.

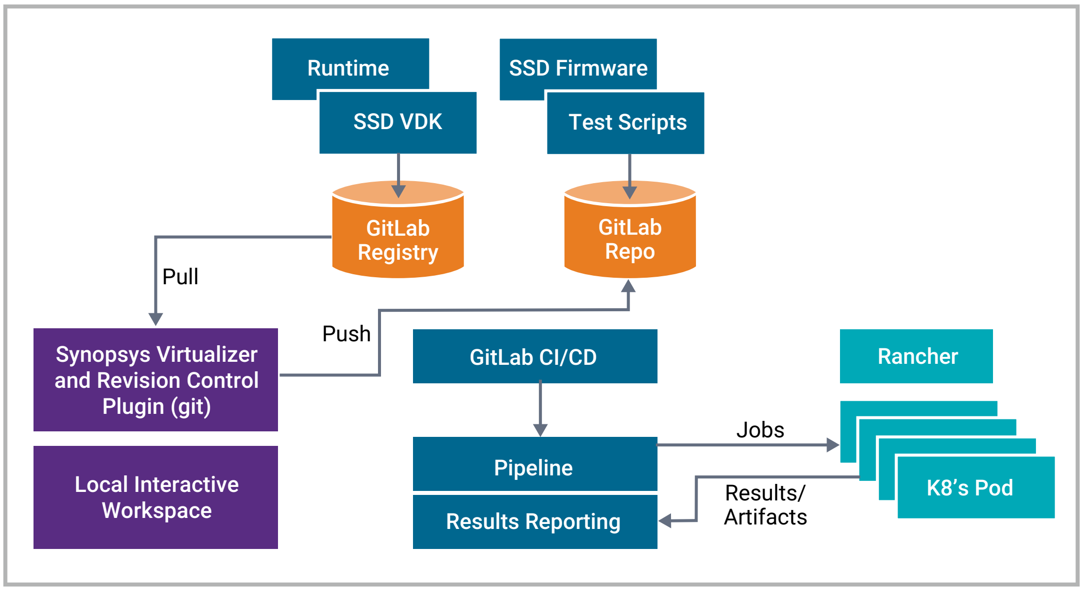


Figure 4: Integration with third-party software in CI/CD flow

The user creates a local development environment by pulling (checking out) the Synopsys VDK and Runtime from the GitLab registry and pulling the source files from the GitLab repository. Users can change and test their own code in their own environment, making sure that their changes work locally. When they are happy with their changes, they can submit them to the central repository by executing a Git push (check-in) command. Any new tests developed can also be pushed so they are run during the CI loops. This triggers a GitLab CI/CD pipeline that will carry out the building of the source code and run the tests to make sure everything is working OK. This tests each user's changed code against changes made by the other developers in the team to ensure that nothing is broken. Typically, the tests pass back results and other artifacts to GitLab, where they can be used and analyzed to produce results back to the team member.

Containerization is an important part of the automation in this flow. Containers provide a portable, packaged compute environment with all dependencies resolved, isolated and abstracted from the host operating system. This ensures that the tests run in a consistent, predictable environment, so that minor differences in machine configuration do not affect test results. Containers are lightweight and agile, supporting scalable and portable testing. Because virtual prototypes are software models, they can be easily containerized. As shown in Figure 5, containers enable a powerful and flexible four-stage testing pipeline: build, test, review, and production.

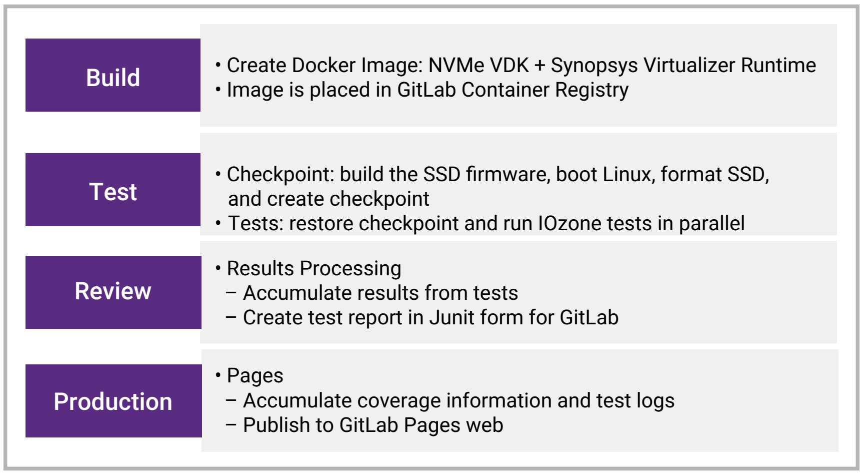

Build
• Create Docker Image: NVMe VDK + Synopsys Virtualizer Runtime
• Image is placed in GitLab Container Registry
Test
• Checkpoint: build the SSD firmware, boot Linux, format SSD, and create checkpoint
• Tests: restore checkpoint and run IOzone tests in parallel
Review
• Results Processing
– Accumulate results from tests
– Create test report in Junit form for GitLab
Production
• Pages
– Accumulate coverage information and test logs
– Publish to GitLab Pages web

Figure 5: Automated CI/CD testing pipeline

In the build stage, the pipeline creates a Docker image with the Synopsys VDK, Synopsys Virtualizer Runtime, and all the dependencies needed to execute it. The resulting image is automatically placed in the GitLab Container Registry. Docker containers derived from this image will be used by most of the subsequent jobs in the pipeline. The test stage starts with creating a checkpoint to be used for the test runs. It builds the SSD firmware and executes it on the EP subsystem processor in the Synopsys VDK. The Linux code running on the RC subsystem boots up and formats the SSD device, creating a usable filesystem on it. The initialization steps are identical for all tests, so creating a checkpoint allows these steps to be skipped for the test runs. Each test runs the IOzone application on top of Linux to exercise the SSD device, and therefore the SSD firmware controlling the device, to measure performance. As shown in Figure 6, containerization allows these tests to be executed in parallel on multiple instances of the Synopsys VDK, covering all combinations of parameters.

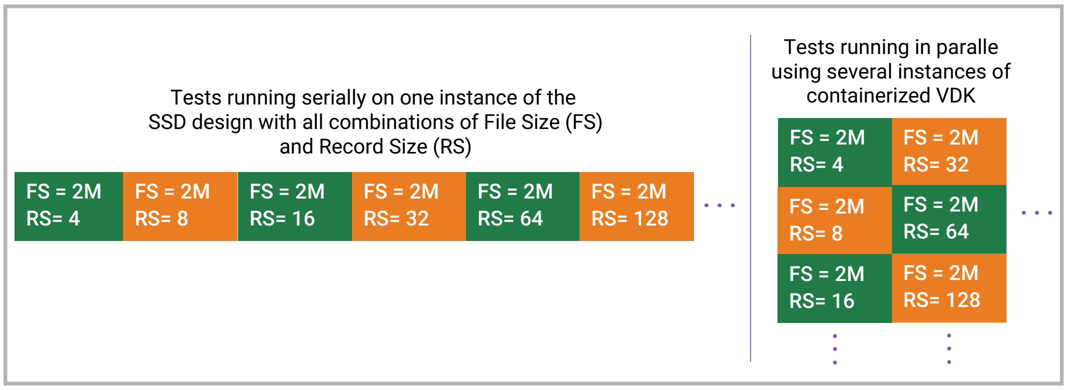
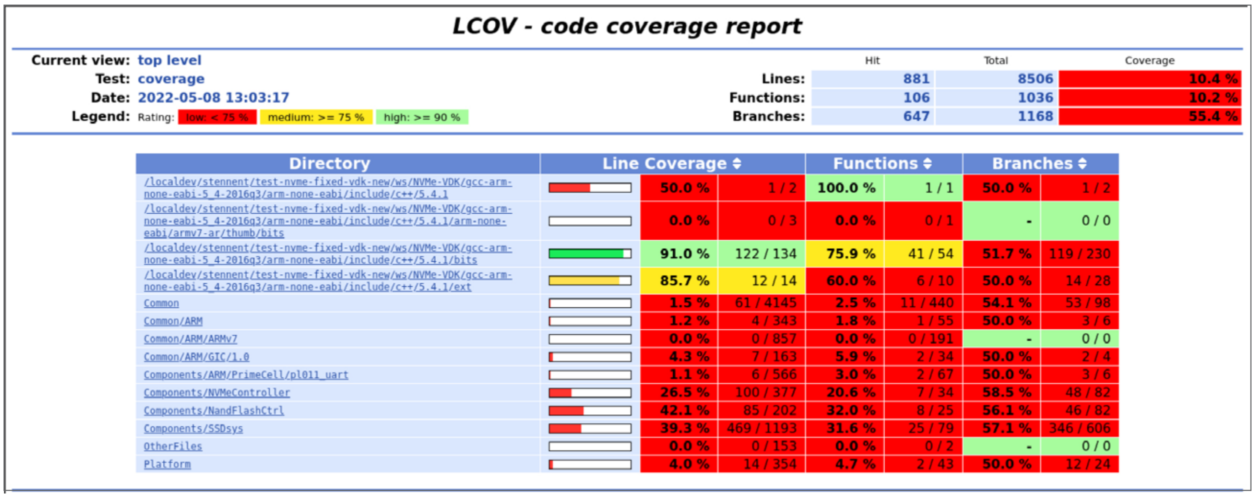

Tests running serially on one instance of the SSD design with all combinations of File Size (FS) and Record Size (RS)
FS = 2M RS= 4
FS = 2M RS= 8
FS = 2M RS= 16
FS = 2M RS= 32
FS = 2M RS= 64
FS = 2M RS= 128
Tests running in paralle using several instances of containerized VDK
FS = 2M RS= 4
FS = 2M RS= 32
FS = 2M RS= 8
FS = 2M RS= 64
FS = 2M RS= 16
FS = 2M RS= 128

Figure 6: Parallelization of IOZone tests

The final two stages in the GitLab pipeline are both concerned with different aspects of results processing. The review stage gathers pass/fail results from the tests and presents these in the GitLab IDE where they can be browsed by the user. The production stage runs a built-in feature of GitLab CI/CD called Pages that allows test results to be presented on a web page. Customized, application-specific measurements and analysis can be extracted from Synopsys Virtualizer using the Generic Data Analysis (GDA) capability. For example, Figure 7 shows the code coverage metrics for the SSD firmware across the IOzone tests and Figure 8 includes the SSD performance as measured by the number of I/O operations per second (IOPS), both as presented on the GitLab Pages site.

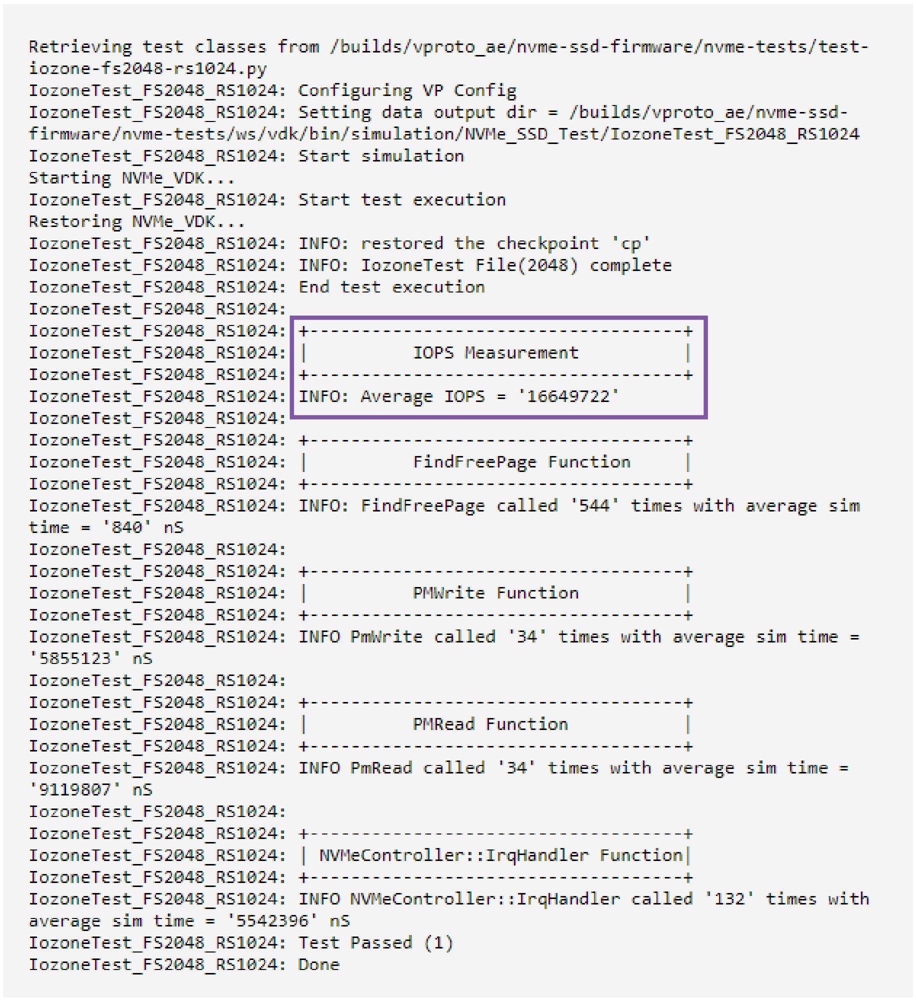


LCOV - code coverage report

				Hit	Total	Coverage
Current view:	top level		**Lines:**	881	8506	10.4 %
Test:	coverage		**Functions:**	106	1036	10.2 %
Date:	2022-05-08 13:03:17		**Branches:**	647	1168	55.4 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %					

Directory	Line Coverage ⬍			Functions ⬍		Branches ⬍	
/localdev/stennent/test-nvme-fixed-vdk-new/ws/NVMe-VDK/gcc-arm-none-eabi-5_4-2016q3/arm-none-eabi/include/c++/5.4.1		50.0 %	1 / 2	100.0 %	1 / 1	50.0 %	1 / 2
/localdev/stennent/test-nvme-fixed-vdk-new/ws/NVMe-VDK/gcc-arm-none-eabi-5_4-2016q3/arm-none-eabi/include/c++/5.4.1/arm-none-eabi/armv7-ar/thumb/bits		0.0 %	0 / 3	0.0 %	0 / 1	-	0 / 0
/localdev/stennent/test-nvme-fixed-vdk-new/ws/NVMe-VDK/gcc-arm-none-eabi-5_4-2016q3/arm-none-eabi/include/c++/5.4.1/bits		91.0 %	122 / 134	75.9 %	41 / 54	51.7 %	119 / 230
/localdev/stennent/test-nvme-fixed-vdk-new/ws/NVMe-VDK/gcc-arm-none-eabi-5_4-2016q3/arm-none-eabi/include/c++/5.4.1/ext		85.7 %	12 / 14	60.0 %	6 / 10	50.0 %	14 / 28
Common		1.5 %	61 / 4145	2.5 %	11 / 440	54.1 %	53 / 98
Common/ARM		1.2 %	4 / 343	1.8 %	1 / 55	50.0 %	3 / 6
Common/ARM/ARMv7		0.0 %	0 / 857	0.0 %	0 / 191	-	0 / 0
Common/ARM/GIC/1.0		4.3 %	7 / 163	5.9 %	2 / 34	50.0 %	2 / 4
Components/ARM/PrimeCell/pl011_uart		1.1 %	6 / 566	3.0 %	2 / 67	50.0 %	3 / 6
Components/NVMeController		26.5 %	100 / 377	20.6 %	7 / 34	58.5 %	48 / 82
Components/NandFlashCtrl		42.1 %	85 / 202	32.0 %	8 / 25	56.1 %	46 / 82
Components/SSDsys		39.3 %	469 / 1193	31.6 %	25 / 79	57.1 %	346 / 606
OtherFiles		0.0 %	0 / 153	0.0 %	0 / 2	-	0 / 0
Platform		4.0 %	14 / 354	4.7 %	2 / 43	50.0 %	12 / 24

Figure 7: SSD firmware code coverage results



```
Retrieving test classes from /builds/vproto_ae/nvme-ssd-firmware/nvme-tests/test-
iozone-fs2048-rs1024.py
IozoneTest_FS2048_RS1024: Configuring VP Config
IozoneTest_FS2048_RS1024: Setting data output dir = /builds/vproto_ae/nvme-ssd-
firmware/nvme-tests/ws/vdk/bin/simulation/NVMe_SSD_Test/IozoneTest_FS2048_RS1024
IozoneTest_FS2048_RS1024: Start simulation
Starting NVMe_VDK...
IozoneTest_FS2048_RS1024: Start test execution
Restoring NVMe_VDK...
IozoneTest_FS2048_RS1024: INFO: restored the checkpoint 'cp'
IozoneTest_FS2048_RS1024: INFO: IozoneTest File(2048) complete
IozoneTest_FS2048_RS1024: End test execution
IozoneTest_FS2048_RS1024:
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: |           IOPS Measurement          |
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: INFO: Average IOPS = '16649722'
IozoneTest_FS2048_RS1024:
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: |           FindFreePage Function     |
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: INFO: FindFreePage called '544' times with average sim
time = '840' nS
IozoneTest_FS2048_RS1024:
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: |           PMWrite Function          |
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: INFO PmWrite called '34' times with average sim time =
'5855123' nS
IozoneTest_FS2048_RS1024:
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: |           PMRead Function           |
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: INFO PmRead called '34' times with average sim time =
'9119807' nS
IozoneTest_FS2048_RS1024:
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: | NVMeController::IrqHandler Function|
IozoneTest_FS2048_RS1024: +-------------------------------------+
IozoneTest_FS2048_RS1024: INFO NVMeController::IrqHandler called '132' times with
average sim time = '5542396' nS
IozoneTest_FS2048_RS1024: Test Passed (1)
IozoneTest_FS2048_RS1024: Done
```

Figure 8: SSD IOPS performance results

Conclusion

Two important innovations in system development, virtual prototypes and containerized CI/CD tests, can be used together very efficiently for embedded software in SoC devices. Virtual prototypes replace physical hardware in the bring-up lab for both pre-silicon and post-silicon embedded code development and test, shortening project schedules while eliminating the cost and hassles of hardware. Virtual prototypes also provide checkpoint/restore, additional debug capabilities, more flexible error injection, and a variety of software and hardware analysis. CI/CD flows speed up embedded software development by minimizing the integration process and automating regression testing.

Using Synopsys Virtualizer virtual prototypes in CI/CD regression flows is a natural combination with many advantages. Embedded programmers get started early with a flexible, scalable approach that runs tests consistently in parallel across multiple test environments and continue to use the same flow for ongoing software regressions. The recent project using the Synopsys NVMe SSD VDK shows how efficient and easy to use this approach can be. The results shown above speak for themselves.

Pub: September 2022