Virtual to the (Near) End – Using Virtual Platforms for Continuous Integration

Jakob Engblom
Wind River
Torshavnsgatan 27
16440 KISTA, Sweden
jakob.engblom@windriver.com

ABSTRACT

Continuous integration (CI) is a hot topic in software development today. CI is a critical enabler for Agile methods and higher software development velocity and productivity. However, adopting the practice of Continuous Integration can be difficult, especially when developing software for embedded systems. Practices such as Agile and Continuous Integration are designed to enable engineers to constantly improve and update their products. However, these processes can break down without access to the target system, a way to collaborate with other teams and team members, and the ability to automate tests. This paper outlines how simulation can enable teams to more effectively manage their integration and test practice, using virtual platforms as a key part of the test setup and simulation as a key part of the test strategy.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]; I.6.3 [**Simulation Output Analysis**];

General Terms

ECONOMICS, RELIABILITY, VERIFICATION

Keywords

Virtual platform, simulation, simulated hardware, transaction-level simulation, TLM, Continuous integration, Agile, simulator integration

1. INTRODUCTION

Continuous integration (CI) is an important component of modern Agile software engineering practice [8]. While the details of CI differ depending on whom you ask, a key part is that rather than waiting until the last minute to integrate all the many different pieces of code in a system, integration and most importantly integration testing is performed as early as possible, as soon as code is ready to run. You cannot really do Agile software development fully unless you have automated builds, automated tests, and automated successive integration – continuous integration. Embedded software developers are actively embracing Agile practices, but are often blocked from doing it fully due to the issues inherent in working with embedded hardware.

A properly implemented and employed CI system shortens the lead time from coding to deployed products, and increases the overall quality of the code and the system being shipped. With CI, errors are found faster which leads to lower cost for fixing the errors, and less risk of showstopper integration issues when it is time to ship the product. In CI, each piece of code added to a system should be tested as soon as possible and as quickly as possible, to make sure that feedback reaches the developers while the new code is still fresh in their mind. The most common technique is to build and test as part of the check-in cycle for all code, which puts access to test systems on the critical path for developers.

Testing soon and testing quickly is logistically simple for IT applications where any standard computer or cloud computing instance can be used for testing. However, for embedded systems and distributed systems, it can be very difficult to do continuous integration and quick automated testing. The problem is that running code on an embedded system typically requires a particular type of board or even multiple boards. If multiple boards are involved, they need to be connected in the correct way, and the connections between them configured appropriately. There is also a need for some kind of environment – an embedded system rarely operates in isolation, it is rather a system that is deeply embedded in its environment, and depends on having the environment in order to do anything useful. Thus, CI for embedded systems tends to be more difficult to achieve, due to the dependency on particular hardware the dependence on external inputs and outputs.

Using simulation for the computer as well as the environment portion of the embedded system offers a potential solution that allows for true automated and continuous integration even for embedded software developers. In our experience, we can achieve this by using high-speed transaction-level (TLM) virtual platforms (VP), along with models of networks, and simulators for the physical world that the embedded system interacts with.

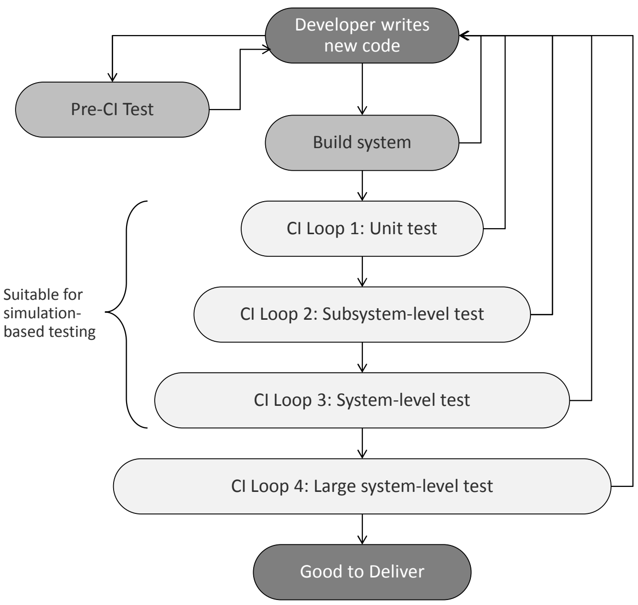
Using VPs for this application means that VPs will be used throughout the product life cycle – it is virtual all the way from the first design simulations to the final test, shipping, and post-delivery maintenance of the system. It is virtual from start to end, with physical hardware used alongside the virtual systems for testing, once hardware is available.

2. CONTINUOUS INTEGRATION

A continuous integration setup is fundamentally an automatic test framework, where code is successively integrated into larger and larger subsystems. As shown in Figure 1, the CI setup typically consists of a number of CI loops, each loop including a larger and

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2747948

larger subset of the system – both in terms of hardware and software. Each CI loop implies integration more of the system software, and running tests that cover a larger part of the system.



Developer writes new code
Pre-CI Test
Build system
CI Loop 1: Unit test
CI Loop 2: Subsystem-level test
CI Loop 3: System-level test
CI Loop 4: Large system-level test
Good to Deliver
Suitable for simulation-based testing

Figure 1: Continuous Integration Concept

The CI system is typically started when the code is checked in by developers. Since code needs to have some basic level of quality before being checked into a development branch or trunk, there is normally a separate *pre-CI test* phase where developers test their code manually or informally to make sure it is basically sound. Once the code seems to work, it is checked in, and automatically submitted to the build system and the CI system.

It is critical to perform testing at multiple levels of integration, since each level tends to catch different types of bugs [3][5][9]. Just doing system-level end-to-end testing on a completely integrated system will miss large classes of errors that are easy to find with more fine-grained tests. Running unit tests are necessary to ensure system-level quality, but not sufficient. Integration testing will reveal many types of issues that are not found in unit tests.

Each successive CI loop covers a larger scope and takes more time to run. The first-level loops should ideally complete in a few minutes, to provide very quick developer feedback. At the tail of the process, the largest loops can run for days or even weeks, essentially being the final testing before delivery.

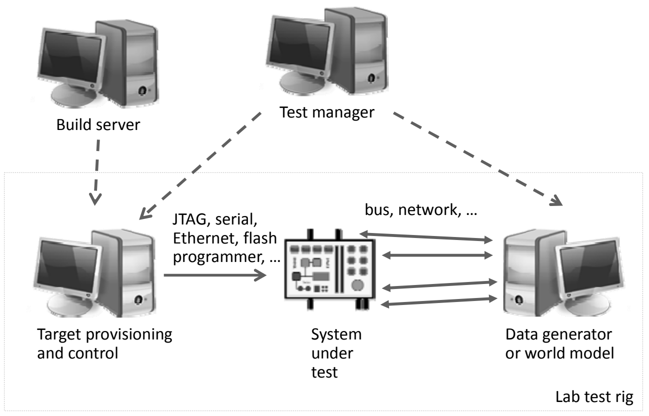
In our experience, simulation can be used for all but the last and largest test loops. In the end, you have to *test what you ship and ship what you test*, and that means that you have to test the system on the hardware that will be shipping. However – that is the last thing that happens before release, and most testing up to that point can be done using simulation.

It is important to note that CI cannot necessarily be applied to any arbitrary existing software stack – in most cases that we have seen, the software architecture has had to be changed to facilitate CI and Agile practices. A key requirement for success is that it is possible to build and integrate parts of a system, as well as ensuring that subsets of the entire system can be tested in isolation. Unit tests and subsystem tests have to be defined, if they do not already exist. Automated testing using simulation can be achieved for almost any system, but continuous integration means more than that. It is a higher level of testing and integration sophistication.

3. HARDWARE-BASED CI

The standard way to do testing and CI for embedded systems is to use hardware. Since you have to at some point test on hardware, a hardware setup is essentially mandatory. However, it is often difficult to set up. As shown in Figure 2, a hardware test setup often consists of a board under test, a master PC that loads software onto the board and runs it, and a test data PC equipped with interfaces such as AFDX, ARINC 429, MIL-STD-1553, CAN, Ethernet, FlexRay, and other buses and networks used to communicate with the real target board.

To test the embedded software on the system under test, it is necessary to have input data to communicate to the target. That is the job of the test data PC in Figure 2. The input data can come from recordings of real-world inputs, from manually written files of input data, or from models that run in real-time. While the test data generator is shown as a PC in Figure 2, it can also be specialized test hardware, in particular for high-performance systems where the data volumes needed are huge and latency requirements are tight.



Build server
Test manager
JTAG, serial, Ethernet, flash programmer, …
bus, network, …
Target provisioning and control
System under test
Data generator or world model
Lab test rig

Figure 2: Hardware Test Rig

The master PC is responsible for managing the target system, including loading software on it, resetting it, and starting target software. The PCs directly connected to the target system are often managed by some central testing system.

Hardware test setups are necessary for doing tests on the hardware, and are universally used for at least final integration testing. However, access to hardware test setups is typically limited since there are not that many setups to go around. Furthermore, they can be difficult to automate and configure quickly enough for small CI loops. In practice, hardware can be so difficult to set up, control, and fully automate that many companies have given up on using it for CI entirely. Instead, testing on hardware is done only quite late in the process using a mostly complete system, essentially going straight to the largest CI loop without using the smaller ones. This brings with it the well-known effect that defects are expensive to fix, since they are found late in the process.

To work around the inconvenience and lack of access to hardware, companies have tried various solutions. Unit testing on can be performed on development boards using the same architecture as the target board, as long as tests do not depend on accessing application-specific hardware. Stubs can be used to imitate the rest of the systems. This gets around the need to have real target boards, but at the cost that it is not really running the final integrated software stack. Once it is time to do integration tests, the actual target hardware is needed.

Another common solution is to develop an *API-based* or *shim-layer-based* simulator. In such a setup, the software is compiled to run on a Windows- or Linux-based PC, and the target hardware and operating system is represented by a set of API calls that can be used on both the target and the host. This provides an environment where application code can run, but it will not be compiled with the real target compiler, it will not be integrated in the same way that software is for the real system, and it will not run the real operating system kernel. They are most useful for pre-CI testing, in practice.

In summary, hardware solutions are sufficiently difficult to use and integrate that they prevent a continuous integration flow that is as smooth and efficient as that experienced by general IT companies.

4. VIRTUAL PLATFORM-BASED CI

To get around the problems caused by hardware, companies have turned to using virtual platforms to run the code. Testing can be performed using standard PCs and servers, reducing the reliance on hardware and expanding the access to hardware virtually. The setups look like that illustrated in Figure 3. The PCs servicing and controlling the target board are replaced with simulation modules. The target board is replaced with a virtual platform.

The target software running on the simulated hardware boards will have to include not only low-level firmware and boot loaders, but also hypervisors, operating systems, drivers, middleware, and applications. To achieve this, you need a fast model of the full target hardware system, and the only way to achieve this is to use a TLM model of the target system, such as Simics [1], SystemC TLM-2 [2], and Qemu.

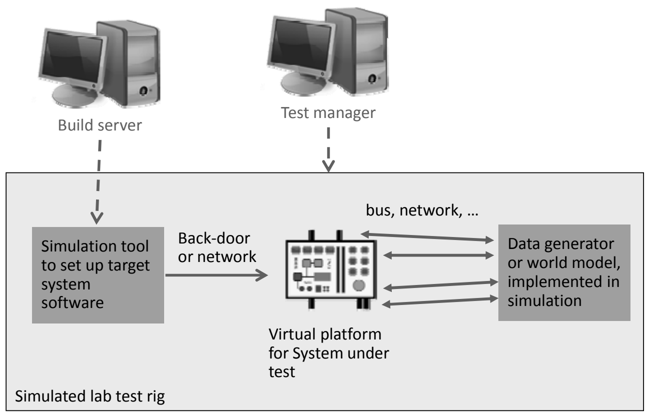

Build server
Test manager
bus, network, …
Simulation tool to set up target system software
Back-door or network
Data generator or world model, implemented in simulation
Virtual platform for System under test
Simulated lab test rig

Figure 3: Virtual Platform Test Rig

It is important to note that a virtual platform model does not have to correspond to the *complete* physical hardware system to be useful. Rather, the most common way to enable CI using simulation is to create a set of virtual platform configurations that are useful for particular classes of test cases, and that do not necessarily all include the entirety of the hardware system. If some piece of hardware is not actually being used for a particular test case, it can be skipped or replaced by a dummy in the model, reducing the work needed to build the model and the execution power needed to run it. A simulation setup must always be designed with the use case in mind.

Figure 3 includes the simulation of the environment on the right-hand side. This is a very important aspect of embedded systems testing. The specific environment differs from system to system; for a mobile phone or base station, the environment is a cellular network. For an automotive control system, it could be a hybrid drive train. For a satellite, it could be the position of the earth, sun, and stars, alongside traffic from mission control. Regardless of the details, the environment needs to be brought into the CI system at some point and in some way. Figure 4 shows how a virtual platform is typically integrated with other simulators to build a complete system simulation that covers far more than just the computer hardware under test.

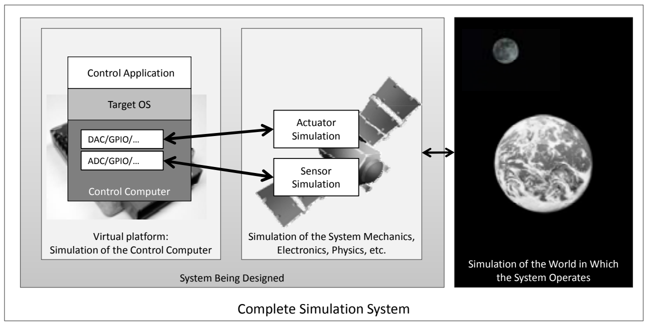

Control Application
Target OS
DAC/GPIO/…
ADC/GPIO/…
Control Computer
Actuator Simulation
Sensor Simulation
Virtual platform: Simulation of the Control Computer
Simulation of the System Mechanics, Electronics, Physics, etc.
System Being Designed
Simulation of the World in Which the System Operates
Complete Simulation System

Figure 4: Simulation Integration

A complete simulation system might not be needed for unit tests, but even for such tests it is recommended to use the real IO paths as used in the integrated stack, rather than trying to push data into the target software directly. The fewer variants of software that you need to build and maintain, the better. Instead, the simulation of the world might be replaced by inputs from a file.

We have also seen cases where the simulation of the rest of the world was actually run on a virtual platform. In such a setup, you have multiple virtual platforms inside the same simulation session, with one running the real-world code under test, and the other running a simulation of the environment. Such a setup directly mirrors a real-world setup that involves multiple computer systems, allowing reuse of test cases.

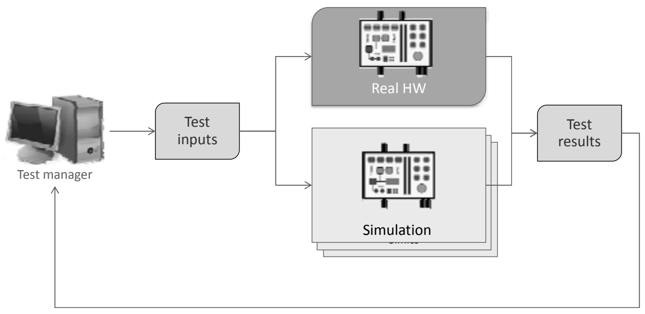


Figure 5: Simulation and Real Hardware Tests

It should be noted that we are building a system for executing tests and collecting test outputs – not for generating the test cases themselves. As shown in Figure 5, simulation is used alongside hardware, running the same tests as are run on the hardware – but running them in a more convenient way and in greater volume. This reuses existing assets in terms of test designs and test scripts, increasing flexibility and ensuring consistency and test validity. Since the simulator is just a software program (or several, in the case that multiple separate simulation environments are used together) with no hardware dependencies, many instances can be run in parallel using a batch processing, cluster, grid, or cloud systems. This can be used to speed up testing considerably. By using simulation, more tests can be run, making it possible to find more errors in testing and have fewer escape to the released software. With more test bandwidth available, more aspects of the product can be tested, increasing overall quality.

In addition to running the tests that are used with the hardware, it is highly recommended to extend the testing with tests specific to simulation, as discussed in more detail below.

The use of virtual platforms to support continuous integration considerably broadens their applicability in terms of the product life cycle. If we look at the PLC stages shown in Figure 6, continuous integration is applicable from platform development all the way to deployment and maintenance. This contrasts with the more common way to view virtual platforms as a way to shift the development cycle to the left by enabling earlier software development.

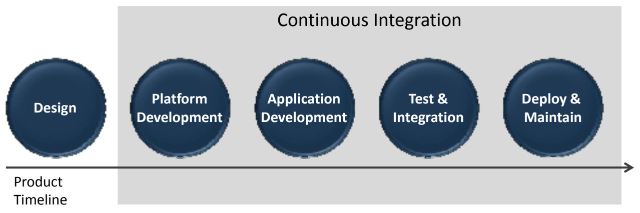


Figure 6: Product Life Cycle

When using virtual platforms and simulation for integration testing, the virtual platform setup is useful even after the final integrated system has shipped. As new software is developed for existing hardware, it has to be integrated and tested, and virtual platforms in a CI system has a key role to play there. Software is continuously evolving, and CI needs to be done continuously.

Of course, as discussed above, once the system gets to deployment, testing will have been done on hardware as well. But the virtual platform is still there, helping to the end and beyond as the system is maintained, extended and upgraded.

5. CHALLENGES ADDRESSED BY INTEGRATED SIMULATION

By using a virtual platform and integrated simulators to do CI, we solve a number of problems that typically affect hardware-based setups.

Hardware Availability Bottlenecks

When working with physical labs, hardware availability is almost always an issue. The number of physical systems available is limited, and time on them rationed in some way. With a simulated setup, hardware availability is not an issue, since the simulator can make any computer simulate any embedded board. The simulator augments the availability of physical boards, removing the constraints that hardware availability puts on both developer spontaneous testing and CI testing.

Using virtual platforms and simulators, each user can have a system of any particular kind to run on whenever they need it. It is also possible to temporarily increase the testing pool by borrowing computer resources from other groups within the same company, or even renting time on a cloud computing service.

Hardware Asset Inflexibility

With a simulation, the same physical hardware box – a generic PC or server - can be used to run software for a wide variety of embedded systems. This provides much more flexibility than hardware labs, since one hardware system cannot be repurposed to test software build for another system.

Hardware Control

Compared to hardware, managing a simulated test system is much easier. As the simulation is just software, it will not run out of control, hang, or become unresponsive due to a bad hardware configuration or total target software failure.

It is also easier to manage multiple software programs than multiple hardware units. Where a physical test system will need to coordinate multiple pieces of hardware and software as shown in Figure 2, a simulation-based setup as shown in Figure 3 has the much simpler task of coordinating a few software programs.

Test Run Latency

When hardware is the bottleneck for testing, it is common to see test campaigns becoming longer and longer as more tests are added when more software is integrated. The time from the point that a job is submitted for execution to the point that it is completed gets longer and longer, as it has to wait for a hardware unit to become available. With a simulation-based setup, much more (virtual) hardware is available for the quick tests, and this leads to shorter test latency.

Test latency is also reduced by parallel testing, making it possible to run through a particular set of tests in shorter time than on hardware. We have seen users previously limited by hardware greatly increase their test coverage and frequency thanks to parallel testing – if you can run your test suites daily rather than weekly, errors will get found earlier, regressions will be caught quicker, and fewer errors will make it out in the field, reducing development costs and increasing product quality.

Test Design Reflecting the Hardware

When limited by hardware availability, real-world tests are often designed to fit into available testing resources rather than to optimally detect problems. This is a practical necessity, as some testing is still infinitely better than no testing. However, with virtually unlimited hardware availability, this is no longer as much of an issue. Tests do not have to be scaled down or modified to match available hardware; instead, the virtual hardware can be set up to match the tests that need to be performed.

For example, we have seen real systems with no connectors available for testing – no serial, JTAG, or utility Ethernet at all. In such a situation development board have to be used for automatic basic testing, and only rare final system tests are painstakingly carried out on the real hardware. With a simulation of the board, a virtual back door can be used to inject software and test the real software.

Limited Configuration Space

In a hardware lab, there is normally a single or a few different hardware configurations available for testing. These might not represent all the different configurations actually found in the real world, but rather a compromise between expense and breadth of testing. With a simulation, there is no need to limit the configurations available, since there is an infinite pool of boards available. It is very easy to create and save and reuse configurations, since they are all just software configurations and setup scripts.

One particular example where configuration richness is important is testing software that runs on multiple different platforms. Ideally, we want to test that software on all target platforms for every change – which multiplies the number of hardware test rigs needed. In simulation, it is very easy to continuously and quickly test software changes across all platforms, even at the unit-test and subsystem-test level.

Hardware Compromises

Another aspect of hardware availability that affects test design is the common use of development boards rather than production boards. With a simulation, a model of the real board can be employed, removing the need for a variant build for the development board. This makes the testing have higher fidelity, and saves the cost to maintain an extra build variant to support testing. It means the software is the real thing, which smoothens the path towards continuous delivery.

Partial System Integration Issues

It is often desirable to integrate some subsystem or other part of a larger system for deep testing, without having to integrate it with the whole system. Indeed, when working with hardware in development, the rest of the system might well not exist at all. Such partial integration is much easier to do using a virtual platform and simulation, since it is simple to capture the interaction between the subsystem under test and the rest of the system, and to inject information that convinces the subsystem under test that the rest of the world is there.

Such stubbing is often more difficult to perform in hardware, as the interfaces to be intercepted can be difficult to get at. In some cases, it is easy, as for a CAN bus, but for a wireless communications system, stubbing things is much harder. Stubbing a rack back plane is also hard in hardware. When it gets down to components on a board, hardware stubbing is pretty much impossible today.

Very Large Setups

There are cases where hardware is just impossible to manage when scaling towards the theoretical limits of a system. For example, in Internet-of-Things (IoT) sensor systems as well as in servers [11], you often need to have hundreds or even thousands of nodes in a single system to test the software and system behavior. In a simulated setting, it is possible to automatically create very large setups without having to spend the incredible amount of time it would take to set up, maintain, and reconfigure such a system in hardware form. Even when hardware is very cheap, configuring and deploying hundreds of separate hardware units is expensive.

Another example would be testing software for hardware that is in development or in prototype state - such hardware is usually very limited in quantity and getting tens or hundreds of nodes for testing networked systems and distributed systems just isn't possible.

Testing Fault Situations

The code that handles faults and erroneous conditions in a system can be very difficult to test on hardware [4], and yet it is critical to ensuring system reliability and resiliency. Hardware test rigs tend to be expensive, and testing is often destructive, which limits fault injection testing on hardware to fairly rare cases.

In contrast, in a simulator, injecting faults is very easy since any part of the state can be accessed and changed. Thus, systematic, automatic, and reproducible testing of hardware fault handlers and system error recovery mechanisms can be made part of the CI testing. This will ensure that fault handling remains functional over time, and increase system quality. Often, the fault and error handling code in a system is the least tested and a constant source of issues [7]. Using simulation and injected faults, such code can be tested to a much higher extent than is possible using hardware.

Testing System Changes

In a simulator, it possible to change the system as it runs. Not just injecting events (like in fault situation testing), but also adding new hardware, removing hardware, or reconfiguring the connection between hardware units. In hardware, doing this either requires manual intervention or some form of robot physically altering the system. It is not impossible, but certainly very difficult. In a simulator that supports dynamic reconfiguration, this is very easy to do and automate.

A concrete example is inserting and removing boards from a rack. With a simulator, it is possible to automatically test that a system automatically configures and boots a newly inserted board. Another example is pulling a board and checking that the system detects that the board is removed and rebalances the software load to the new system configuration. In the context of CI, this makes it possible to test that the platform and middleware performs as designed, when integrated with the hardware and each other.

Introducing Environmental Changes

It is also very valuable to introduce varying environmental conditions as part of continuous integration and testing. In the end, an embedded system is integrated into the world, and that integration needs to be tested. We are not talking about "faults", really, but rather behavior that is expected from an uncooperative physical world. Testing how a system responds to various environmental conditions is a very valuable use case for simulation – and one where simulation is being used extensively for physical systems already.

For example, for a wireless network system, we want to test the integrated software behavior in the presence of weak signals and asymmetric reachability. Such testing is very easy to perform using a model of the network, but very difficult to perform in the real world.

Feedback to Developers

Using virtual platform checkpoints [6], it is possible to save intermediate points in the test setup flow, such as the point where a system has been booted and after software has been loaded. Figure 7 shows a typical workflow where the system to use for tests is first booted, the booted state is saved, and used as the starting point for loading software. Since checkpoints should be handled as read-only, it is possible to base many runs off of the same checkpoint. Once software is loaded onto the system, another checkpoint is saved, and this checkpoint is used as the starting point for a series of tests.

On a hardware system, each test would have to start by booting the system or cleaning it in some way to remove the effects of previous tests. In a simulator, each run can start from a known good state, with no pollution from other tests. Checkpoints can save a lot of time in running tests by removing this overhead.

We have also seen customers use checkpoints to manage the setups used for testing in a more proactive way. For example, by creating a checkpoint of each nightly build of the basic software platform, all developers and all tests run on a particular day will start from a known and well-defined state, rather than relying on whatever software happens to be loaded on the target systems from the previous day.

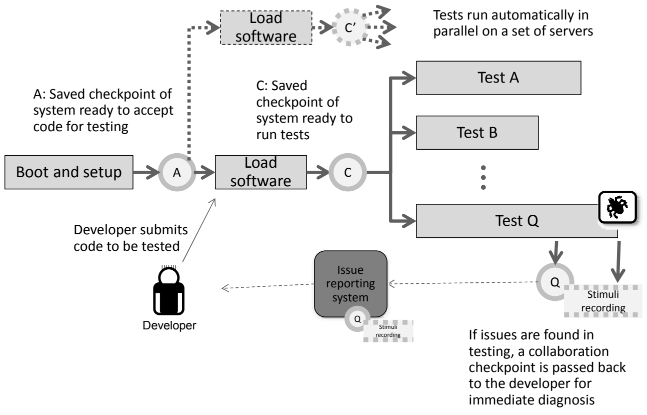

Load software
C'
Tests run automatically in parallel on a set of servers
A: Saved checkpoint of system ready to accept code for testing
C: Saved checkpoint of system ready to run tests
Test A
Test B
Boot and setup
A
Load software
C
Test Q
Developer submits code to be tested
Issue reporting system
Q
Stimuli recording
Developer
If issues are found in testing, a collaboration checkpoint is passed back to the developer for immediate diagnosis

Figure 7: Workflow Optimizations with Checkpoints

Figure 7 also shows how checkpoints are used to manage issue reports from testing. In addition to the classical information in an issue report (text describing what happened, collections of logs and serial port output, version and configuration data, etc.), checkpoints (containing a recording of all asynchronous inputs) can be used to provide the developer with the precise hardware and software state at the time that the issue hit. This removes the guesswork in understanding what the test did and how the software failed, and is a tremendous boost for debugging efficiency.

The checkpointing methodology works with external simulators or data generators, by simply recording the interaction between Simics and the external simulator. When reproducing the issue, the data exchange is simply replayed, without the need for the external simulator or data source. Such record-replay debugging is a very powerful paradigm for dealing with issues that appear in complex real-time and distributed systems with many things happening at once. A recorded debug session can also be used to drive a reverse execution system and reverse debugger, such as Simics [1]. Once a recorded session has been replayed, it is easy to reverse within it, allowing reverse debugging to be used with an integrated simulation.

6. REAL-WORLD USAGE

This paper is based on many real-world customer applications of simulation technology, and we integrated lessons learnt in the above text.

In general, the first benefit that is realized by using virtual platforms and simulation for testing is that hardware availability constraints are removed. This makes it possible to shorten the lead time for tests and for all developers to have access to test platforms [10]. The second step is to automate testing using the virtual platforms as the execution engine – and the logical conclusion to the automation process is the implementation of a full CI system using both virtual platforms and hardware.

Once automatic testing is in place, test parallelization is the next step. In this step, the latency for running batteries of tests is reduced, sometimes radically. In particular, by making test batteries complete faster, they can be brought from weekly to daily, and from daily to hourly, shortening feedback loops and increasing product quality.

Final testing before release is always performed using hardware, and so are certification tests. A virtual platform no matter how good does not contain all the details needed to reflect real-world hardware stability.

Virtual platform use in testing does not end when a product is shipped. The software load in the product will be maintained and updated over time, and the new software will need to be tested on the "old" hardware. Indeed, as the hardware gets older, the virtual platform might be the only choice for volume testing, as lab hardware availability shrinks due to breakage or culling of lab systems.

The platforms targeted for testing using virtual platforms have ranged from single boards to a few boards to racks of many tens of boards and hundreds of processors.

7. SUMMARY

Continuous integration (CI) is an important part of modern software engineering practice. By using CI, companies achieve higher quality and enable continuous deployment, among other benefits. However, implementing CI for embedded systems can be a real challenge due to the dependency on particular processors, particular hardware, and particular environments. Using simulation for the environment and virtual platforms for the computer component, it is possible to enable CI for systems that seem "impossible" to automate in the physical world.

Virtual platforms can also bring other benefits, such as better feedback loops to developers for issues discovered in testing, and expansion of testing to handle faults and difficult-to-setup or difficult-to-afford configurations.

8. REFERENCES

[1] Aarno, D, and Engblom, J. *Software and System Development using Virtual Platforms - Full-System Simulation with Wind River Simics*, Morgan Kaufmann Publishers, 2014.

[2] Aynsley, J. *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative (OSCI), 2009.

[3] Bland, M. "Finding More than One Worm in the Apple", *Communications of the ACM*, Volume 57, Number 7, July 2014.

[4] Chessin, S. "Injecting Errors for Fun and Profit", *Communications of the ACM*, Volume 53, Issue 9, September 2010.

[5] Eldh, S. *On Test Design*, Doctoral Dissertation #105, Mälardalens Högskola, Västerås, October 2011.

[6] Engblom, J. "Transporting Bugs with Checkpoints," *Proceedings of the System, Software, SoC and Silicon Debug Conference (S4D 2010)*, Southampton, UK, September 15-16, 2010.

[7] Ganssle, J. *The Firmware Handbook,* Newnes, 2004.

[8] Girard, A. *Agile and DevOps for Embedded Systems – Engineering Trends Analysis*, VDC Research, 2014.

[9] Koerner, S., Kohler, A., Babinsky, J., Pape, H., Eickhoff, F., Kriese, S., Elfering, H. "IBM System Z10 Firmware Simulation", *IBM Journal of Research and Development*, Volume 52, Number 3, Paper 12, 2009.

[10] Magnusson. P. "The Virtual Test Lab", *IEEE Computer*, May 2005.

[11] Rechistov, G. "Simics on the shared computing clusters: the practical experience of integration and scalability", *Intel Technology Journal*, Volume 17, Issue 2, 2013.