Case Study: Successes and Challenges of Validation Content Reuse

Mike Chin
Intel Corporation, Folsom, CA
michael.a.chin@intel.com

Jonathan Edwards
Intel Corporation, Folsom, CA
jonathan.edwards@intel.com

Hooi Jing Tan
Intel Technology Sdn. Bhd., Penang, Malaysia
hooi.jing.tan@intel.com

Josh Pfrimmer
Intel Corporation, Folsom, CA
josh.c.pfrimmer@intel.com

***Abstract* - At Intel, the IP validation team creates internal test content used to validate the functionality of various IPs in the SoC. This content is flexibly designed to support different pre-silicon and post-silicon environments, and has been used to validate successive generations of IP revisions. This content is also scalable and has been reused in SoC validation where these IPs are ultimately integrated. This paper discusses the challenges to apply reuse from IP to SoC, and the key design and strategic decisions that we made to overcome them. We discuss our results, including headcount savings, RTL bugs found, and operational cost savings.**

Key Words: Verification, Validation, Portable Stimulus, PSS, Reuse, IP, SoC, Manufacturing

I. Introduction

Moving with agility is a primary challenge in hardware today. Variations in an SoC design grow each day as novel uses are innovated, new customers are found, and generational improvements are made to industry standards. As much as the challenge exists on the design side of the equation, validation faces this same problem in ways that grow exponentially in complexity. As our portfolio expands to include new IPs, new features on evolving IPs, and emergent features which cross IP boundaries, the responsibility of validation teams to verify functional correctness for all permutations and combinations of how different agents on a platform interact is an NP complete problem. Add in the complexity of IP versus SoC validation, pre-silicon versus post-silicon validation, and differences in execution platform configuration (real devices vs. virtual devices, platform capabilities). What results is a complex problem with multiple moving parts. Designing software to validate for this ecosystem is an equally difficult problem to solve.

How do we find efficiency in such an ecosystem? What approaches can we take to simplify the problem space into smaller, tractable problems that we have hope of finding solutions for? What challenges have we faced and continue to face in this domain? We have taken a multi-year journey down the path to create validation content to validate IPs in pre-silicon validation, extend this to SoC validation, and ultimately reuse this into manufacturing. In this paper, we'll discuss how we have tackled the growing challenges through reuse, our approach to solving a scalability problem, and the results of this effort.

II. Glossary of Terms

Term	Definition
API	Application Programming Interface
bare metal	minimal validation environment without an operating system
FPGA	Field Programmable Gate Array
HAL	Hardware Abstraction Layer
JIT	Just-in-Time
IO	Input-output
IP	Intellectual Property; refers to discrete functional blocks implementing a specific protocol or function within an SoC

ISA	Instruction Set Architecture
OEM	Original Equipment Manufacturer
post-silicon	phase of project occurring on physical prototype units
pre-silicon	phase of project development prior to the availability of silicon prototype units
PSS	Portable (Test and) Stimulus Standard
RTL	Register Transfer Language; object code for specifying and designing circuits and systems
sideband	communication path other than the primary objective bus
SoC	System on Chip
stepping	manufactured version of an SoC
transactor	software functional model of a bus or endpoint
TTR	Test Time Reduction

III. Background

IPs serve as foundational building blocks for the SoC design. At Intel, we deal with many different domains of IPs – high speed IOs, low speed IOs, security, context/audio, storage, etc. Thorough validation of these IPs is crucial to ensure the proper function of the IP in varying and unanticipated ways. The goal for our validation is to validate to specification – any behavior that is compliant to the specification must be supported. This includes different permutations and combinations of settings for an operation. This also includes different behaviors of the IP, both sequential and in parallel. The end goal is to validate for completeness of spec, and also to subject the IP to highly stressful workloads to ensure proper functionality. All of this allows us to thoroughly test the microarchitectural design, internal state, and any shared resources.

IP validation is executed on two platforms – IP emulation, and FPGA. FPGA platforms allow IP validation to test the IP with real devices and testcards. On emulation platforms, endpoint devices differ – in some cases virtual versions of testcards may be used for validation. In most cases, transactors are used to emulate the traffic between the controller and IP. Each platform achieves its specific goals towards the validation of the IP, so portability across platforms is an important consideration for our content. As a result, these differences in endpoints between platforms create challenges to implement extensible validation content.

SoC and subsystems are treated as collections of IPs with additional integration logic. The focus of validation is to ensure correctness in the integration of IPs, cross-IP flows and interactions, subsystem/SoC flows, and digital-analog layer integration. There is less emphasis on the depth of the IP but a focus on the role of the IP within the broader SoC or subsystem operation. Executing system flows often place a requirement upon the IP, which IP content must scale to support. IP content is also needed to validate normal operation after system flows – running IP operations successfully ensures the IP readiness. Thus, IP operations are used as a means to validate integration into the SoC.

SoC validation is executed on emulation, and post-silicon platforms. Similar challenges to IP validation platforms exist here – emulation uses transactors and limited usage of virtual devices while post-silicon platforms use real devices and testcards. Additionally, change in platform support exist between IP and SoC platforms. Power management, sideband access, and fuse support are a few of the differences between IP and SoC platforms. SoC platforms are also more configurable, as product SKUs are needed to be configured and tested. All of these variations add additional challenges to our ability to scale content.

Parts screening in manufacturing also presents its own challenges. Limited to post-silicon platforms, the constraints of the manufacturing environment also place constraints on the type of post-silicon platforms and devices that are used. Additionally, not only are the types of devices different, but due to the need for thorough screening, more units are needed to thoroughly screen all the IO interfaces.

The challenge that we face in validation content development is multifaceted. How do we address the depth of validation needed for IP validation? How does this content scale to SoC validation, and ultimately to screen all the interfaces in manufacturing? How easily can the content design scale to integrate and support new SoC content?

When we examine the different challenges of executing across a wide diversity of platforms, how does content scale to detect and support different endpoint devices?

IV. Solution

Our software architecture centers around a simplified software stack, and proper adherence to roles and responsibilities in each layer of the stack. As shown in Figure 1, the software stack consists of 4 main blocks of code. Our primary motivation in the software stack architecture is to design with reuse in mind. This reuse not only includes reuse in validation, but also allows for variation in how the code can be used. It is conceivable that our code must be used in bare metal, kernel driver code for various operating systems like Windows and Linux, or even as user space code in an application.

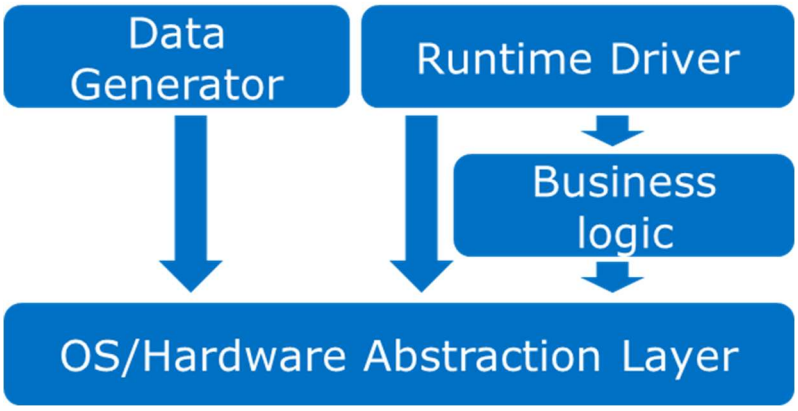

Figure 1. IP Validation software stack

The hardware abstraction serves as the main layer for code portability, providing APIs and services for operating system and hardware access. Access to operating system services like logging, thread control, and memory allocation are abstracted. Hardware access to devices over memory mapped IO also have API functions. All of our code is built on top of this layer.

```
//  Memory access functions
uint32_t mem_read32(uint64_t base, uint64_t offset));
void mem_write32(uint64_t base, uint64_t offset, uint32_t data);

//  Configuration access
uint32_t readCfgDictionaryAsInt(std::string dictionary, std::string key);
```

Figure 2. Sample Hardware/OS abstraction functions

The data generation layer serves as our code block that allows us to set up parameters in the IP, and any data buffers that are needed. This code does not touch hardware – no controller programming is implemented in this layer. Rather, any choices that need to be made for the IP are resolved in this layer. Users have the ability to specify different values for configuration knobs, which are resolved in this code to the operational parameters used elsewhere in the code. Data buffers are allocated and populated with pattern data for IP operations. Code in this phase consolidates all decision making so execution on the target can be as efficient as possible. As a result, this code can be shifted left to execute earlier the test build/execution flow – executed as part of a JIT, or off-target setup code.

```
DMAParams IPDMAGen(std::string dictionary) {
      DMAParams dp;
      dp.burstMode = readCfgDictionaryAsInt(dictionary, "burstMode");
      dp.size = readCfgDictionaryAsInt(dictionary, "size");
      return dp;
}
```

Figure 3. Sample data generation code

In the data generation, we utilize structs as our primary mechanism for data transfer. With the integration into a modeling frontend, we try to keep the interface functions as static as possible in order to minimize change in the code. By utilizing structs, we found that we can keep the interface static, and as new validation knobs need to be added, we can simply modify the struct and implementations that support the struct without having to modify the interface and modeling.

The device library layer is what we refer to as our business logic layer, where all the IP programming sequences reside. Code in this layer accesses the controller, and encompasses code to initialize and configure the IP, program operations, or even programmatic sequences for handling interrupts. Inputs are taken from the data generation layer, so we minimize the amount of non-IP code being run during the execution phase. The other restriction placed on code in this layer is that OS services are not used. This code is intended to be portable across multiple targets, so we keep it as agnostic to the OS as possible.

```
IPDMA(uint64_t baseAddress, DMAParams dp) {
        mem_write(baseAddress, 0x4, dp.burstMode);  // Burst mode
        mem_write(baseAddress, 0x8, dp.size);  //  Size of transaction
        mem_write(baseAddress, 0x0, 0x1);  // Go
}
```

Figure 4. Same device library

Drivers comprise the final layer in our software stack. Drivers serve as the mechanism to marshal the user intent during execution with what is allowed by the framework or operating system. Inputs from data generation are passed to driver instances, and any hardware accesses by the driver are accomplished through function calls into the device library.

Portability is our main concern with our software stack. The data generation and device library layers are designed for portability so they only need to be implemented once and can be ported to any validation framework without modification. As the foundation for data generation and device programming, the HAL layer must be implemented for any new framework to be used. We try to keep the subset of functions to fully implement the HAL as small as possible, to minimize work. Drivers, by design, are not portable across frameworks. Since they interface directly with their target framework, we assume reuse of this layer is not possible because different operating systems and validation frameworks have different implementations to support features like interrupts. However, based on work done to apply reuse, we estimate this layer to only comprise of roughly 10-15% of the overall code.

This software stack allows for the fundamentals needed to access hardware and program. One of the problems when attempting to develop validation content is the fundamental code support for hardware that all applications need to use. With this layer, we believe we have taken a significant step forward to reduce, and in many cases eliminate the hardware support, so we can focus on the validation applications that will help stress the hardware. Validation applications can reflect different validation methodologies and focus areas. Some applications may focus on the memory controller and caching, while others may focus on concurrency of IOs, or even power management.

One approach to an application layer for hardware validation is to utilize modeling. Our current modeling utilizes Cadence Perspec and the SLN syntax (a modeling syntax that pre-dates and heavily influenced Portable Stimulus) to form models of each IP, with an eventual full move to Portable Stimulus. The design of each IP model focuses on defining units of operation at the same level of granularity as the validation test plan. These operations are referred to as content capabilities and can be combined to create executable tests that are expressed at the operational IP level. The choice in the level of granularity is crucial in effective model design – too high level of a flow will result in similar operations overlapping. Too low level of modeling can devolve into specifying individual register programming. A key barometer in judging the correctness of our flows is to examine the modeling actions against their realized implementation – well-formed actions generally share little code with other actions in the model for programming the intent of the operation. These IP models contain all the code needed to execute the IP using our software stack as a realization layer to direct data generation and driver calls at the appropriate time.

Applied to subsystems and the SoC, our modeling treats each IP as an individual subcomponent of the SoC. This allows the IP model and their realization layer to exist without the need for any modification. Additional modeling is

needed to support cross-IP and system flows. These reside as part of the SoC model in order to maintain a clean separation of responsibilities between the integrated IP model and the SoC. Utilizing this approach, we are able to fully support the functionality of the SoC, down to the individual IPs and their operations, while only adding incremental improvements and avoiding rework.

Outside of enabling validation applications, we must solve the problem of supporting different platform capabilities. As described previously, platform differences between pre-silicon and post-silicon, and variations on the configuration of a platform create a wide diversity of challenges for validation content to execute on. Misconfiguring our content results in mis-execution and wasted cycles. We utilize a strategy to discover as much information as is needed to properly configure and constrain our IP and SoC models. Discovery looks at the capabilities of the IPs to determine the version, and hardware features. It also examines the attached devices to try to determine what type of device is attached. Along with information retrieved through BIOS settings, this gives a full picture of all the necessary inputs for our content to run on a platform. All of this information is then fed into our modeling, which allows us to enable and disable features based on the capabilities on the platform, and subsequently generate content that can run on the platform. This process of platform discovery is critical in our ability to scale across IP, SoC and manufacturing. The discovery and configuration process allows us to detect devices and properly support them in their usage model.

V. Results

Effectiveness of our validation content is primarily measured in terms of RTL bugs found. Figure 5 illustrates the number of RTL bugs found across all IP families covered by our content. The number of bugs is significant, as each RTL bug is found in pre-silicon, which prevents bugs from escaping into real silicon. IP design that is reused across multiple programs also benefit. Bugs found and fixed in pre-silicon result in cleaner silicon, which helps to reduce additional steppings at the cost of millions per stepping.

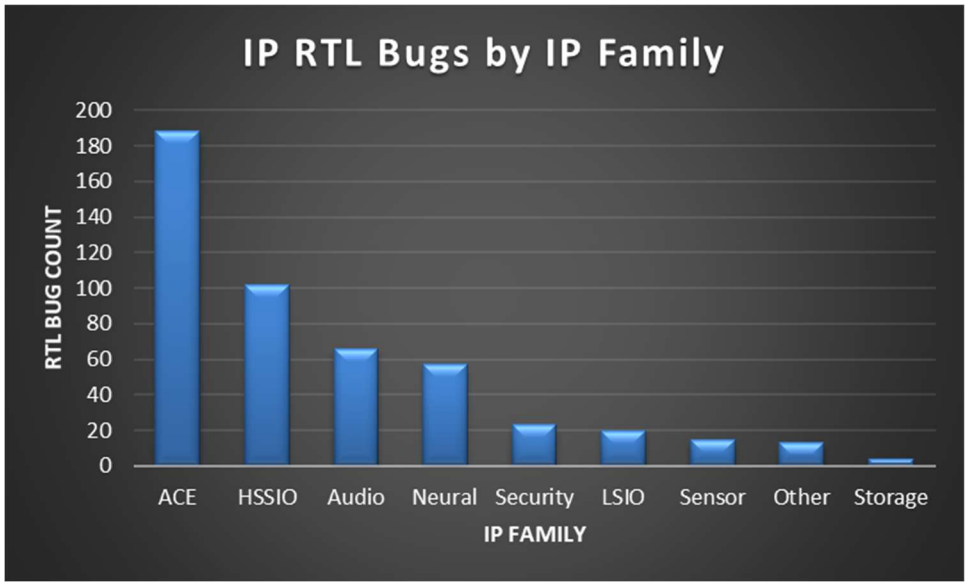

Figure 5. Reusable IP content has been successfully applied across a breadth of IP domains to find RTL bugs

At the IP level, we have reused content project-over-project for multiple generations. As illustrated by Figure 6, reused IP content is effective in finding bugs project-over-project, with a growing number of bugs found as content and validation continues to mature.

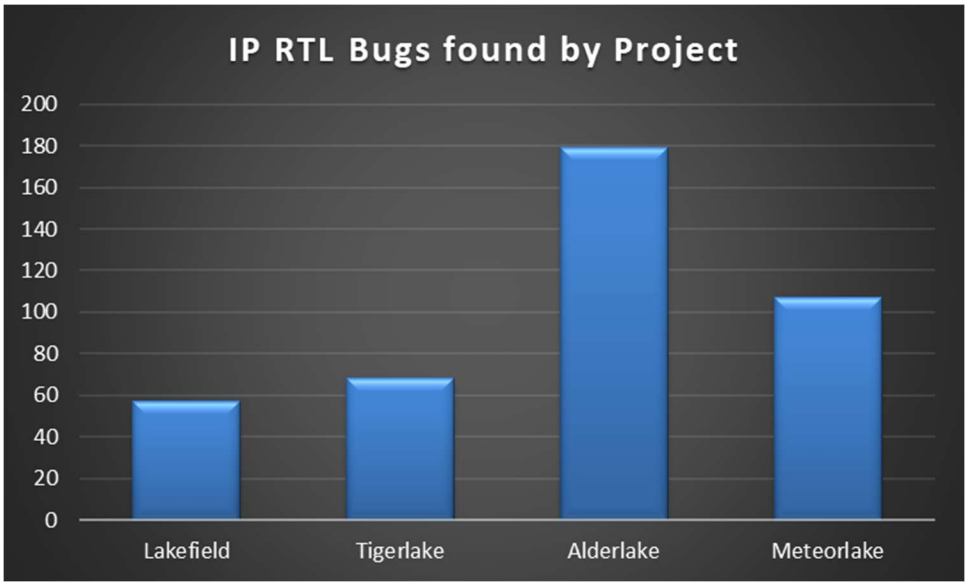

Figure 6. Reuse applied to multiple projects continues to find bugs and increase in effectiveness. (Meteorlake is currently in-flight)

SoC RTL bugs found is another key measure of the success of reuse. By reusing IP content and only investing in SoC-specific flows, SoC bugs found are deemed very high in ROI due to the effort expended. Through 4 successive projects, we have found a total of almost 200 SoC RTL bugs. Figure 7 shows the percentage of bugs found through content reuse and illustrates the increasing effectiveness of reuse as it trends to find a higher percentage of bugs (green) compared to all bugs found (blue).

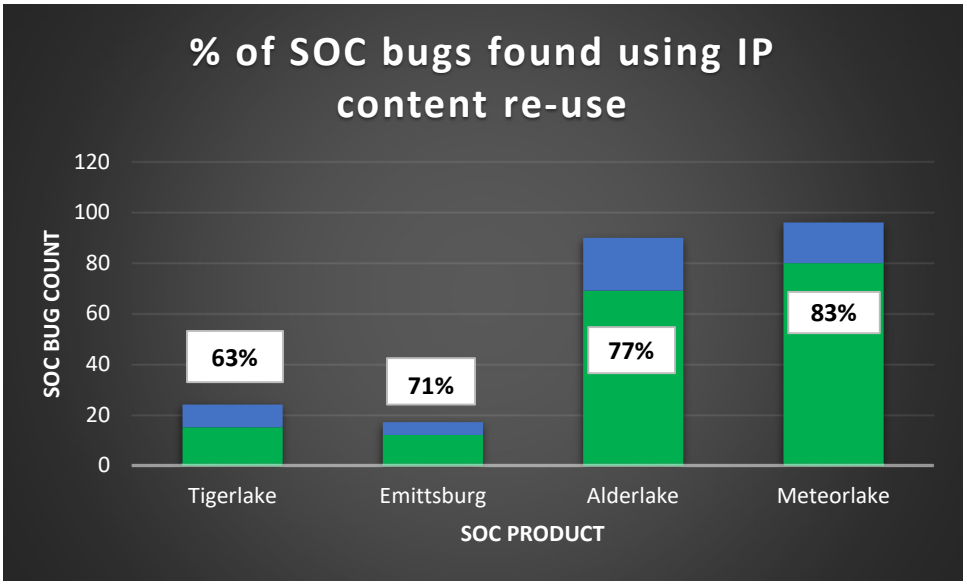

Figure 7. Content reuse applied to SoCs shows high effectiveness at finding bugs at integration.

Our reuse engagement with manufacturing is a key driver towards operational cost savings. Content is reused used in parts screening at the full system level and is the last step between finding manufacturing defects and releasing to OEM partners. Operational cost savings through content reuse is derived from effectiveness of screening content, test time reduction (TTR) and resulting throughput of parts testing. Sufficient throughput based on test time reduction lowers the need for more screening systems to offset test time.

TABLE 1
MANUFACTURING COST SAVINGS FROM CONTENT REUSE

Manufacturing Areas	Calculation	Savings
System-to-Tester		$10M
TTR	$0.02 * 240M	$4.8M
Screening Unit Elimination	$450K * 11	$4.95M
Total savings (to date)		**$19.75M**

Reuse efforts naturally reduce the total amount of development needed to achieve the same amount of content. Our development team of roughly 15 is able to leverage our reuse architecture to scale across a broad range of validation scope. Areas where content development would need to be duplicated are naturally areas where development costs can be reduced. Table 2 shows our estimated headcount reduction over the lifespan of our effort (4+ project generations) by embracing content reuse. This headcount can be used to provide additional benefits elsewhere in the validation ecosystem.

TABLE 2
DEVELOPMENT HEADCOUNT SAVINGS FROM APPLYING REUSE

Downstream Re-use Customers	HC Savings
South complex pre-si	4
North complex pre-si	4
Cross-die pre-si	4
Post-si Client/Devices	14
Manufacturing	6
Total	**32**

Reuse has also resulted in increased stability of content. Figure 8 illustrates the number of issues related to content filed project-over-project. Referred to as noise, we strive to reduce noise for every successive project. As illustrated, we find that new content adds noise, primarily due to enabling, while the bulk of the content base remains stable.

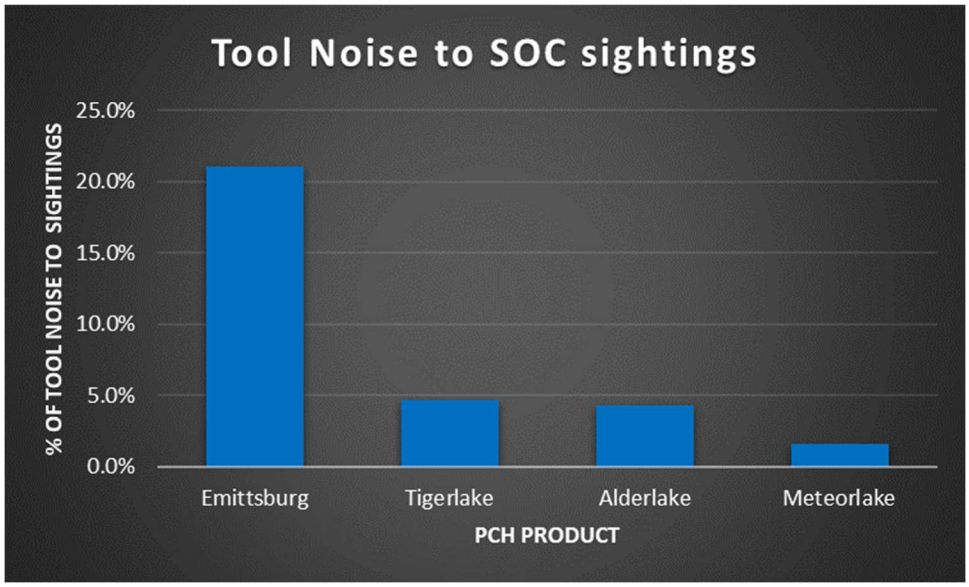

Figure 8. Content stability drives efficient validation execution velocity. Decreasing trend in content noise yields higher quality content.

VI. Future work/Challenges

We face a multi-faceted roadmap forward to address the growth and application of content reuse. Technical improvements are at the forefront of continuing and proliferating our success. In this area, we are focusing on 3 key areas to expand the effectiveness of reuse:

- Portable Stimulus Standard – Moving our modeling to fully embrace the Portable Stimulus industry standard is one of our key efforts. Embracing this standard allows validation to talk at the level of the test plan and create content that reflects this. As an abstracted frontend interface, this gives us capabilities to easily adapt our content when the underlying IP and SoC changes. Preliminary scoping shows applicability of the standard to ~70% of our modeling problems. Effort and collaboration with the industry at-large is needed to address identified modeling problems.
    - A related challenge to modular Portable Stimulus modeling is proper encapsulation and support for cross-IP flows. As advancements in hardware design continue, the creation of subsystems of multiple, inter-connected IPs is the new norm. We must be able to scale IP models to support subsystem modeling and interconnected operations, and their inclusion in multiple subsystems concurrently, while also supporting standalone IP operations.
- 3rd party interoperability – As the scope and variety of projects grows into the future, we see potential for 3rd party IPs to be integrated with our products. With IP RTL comes the need for validation collateral that is necessary to validate the integration of IP with SoC. Requiring vendors to support the breadth of validation platforms and operating systems/frameworks being used is an unreasonable requirement to a vendor. Rather, we look to explore standardized abstraction layers that driver collateral can be developed on.
- One of the challenges as Intel expands its foundry portfolio is its ability to scale the validation environment from one instruction set architecture to another. In this effort towards more ubiquitous computing, it is not guaranteed that all architectures follow x86. This provides a substantial challenge to overcome, which otherwise will limit the expansiveness of our reuse. We continue to investigate solutions that allow us to be portable across multiple ISAs without customizing our content.

Portable stimulus serves to standardize the upper layers of validation test intent across the industry. As we look at our end-to-end challenges in the content space, we feel that end-to-end standardization also provides tremendous potential for efficiency and interoperability. A standard that materializes as a common verification framework or operating system could lead to a hardware abstraction layer for device programming, and abstract platform/ISA. In a world of increasing SoC complexity and 3rd party RTL integration, we feel like PSS is the tip of the iceberg – driving additional standardization further down the realization layers will help increase velocity and quality in the industry.

The desire to embrace our content reuse also drives more diverse usage models. Headless execution platforms (platforms and environments without a main processor) provide one such usage model, both in pre-silicon validation as well as in areas such as manufacturing screening. Co-validation of the production software stack with synthetic content presents another opportunity. Mixing production and synthetic allows validation to test cutting edge features in an environment where no device support exists. This capability also allows pathways into more efficient debug and issue reproduction. We continue to explore reuse down these paths and others to drive validation and debug efficiency, feature coverage, and cost savings.

VII. Conclusion

Through multiple generations of major Intel client projects, we have successfully used and reused validation content from IP validation across generations and into SoC. Validation content was designed for portability, and exercised across FPGA, emulation and post-silicon platforms - each platform being used with different configurations and capabilities. Starting from our reuse engagement, validation teams found in excess of 600 RTL bugs across IP and SoC. Proliferating reuse out of validation, specifically into defect screening, we are able to provide additional benefits in the area of reduced operational costs and higher product quality.

We continue to drive enhancements and engagements to expand the footprint of reusable content and improve the effectiveness and efficiency when embracing our work. Future work towards fully embracing PSS, varying depths of coverage are targeted to continue to drive towards higher levels of quality. Additional engagements that leverage our content base continue to challenge and expand our definition and applicability of reuse. All of these efforts will continue to feed a stream of innovative ideas and solutions to benefit Intel and the broader industry.

ACKNOWLEDGMENT

We would like to thank the Intel VICE software team and management for their efforts in reusable IP content. We would like to thank the CAVE software team and management for their support and effort in adapting synthetic validation content into their methodology. We appreciate our continued partnership with our validation and manufacturing teams as they continue to embrace reusable content into their methodologies. A special thanks to Ernie Jennison for his sponsorship of this initiative, mentorship and influence.