Post-Silicon Validation of the IBM POWER9 Processor

Tom Kolan[1], Hillel Mendelson[1], Vitali Sokhin[1],
Kevin Reick[2], Elena Tsanko[2], Greg Wetli[2]
[1]IBM Research. Email: {tomk, hillelm, vitali}@il.ibm.com
[2]IBM Systems & Technology Group. Email: {reick, etsanko, wetli}@us.ibm.com

***Abstract*—Due to the complexity of designs, post-silicon validation remains a major challenge with few systematic solutions. We provide an overview of the state-of-the-art post silicon validation process used by IBM to verify its latest IBM POWER9 processor. During the POWER9 post-silicon validation, we detected and handled 30% more logic bugs in 80% of the time, as compared to the previous IBM POWER8 bring-up. This improvement is the result of lessons learned from previous designs, leading to numerous innovations. We provide bug analysis data and compare it to POWER8 results. We demonstrate our methodology by describing several bugs from fail detection to root cause.**

I. Introduction

High-end processors, such as the IBM POWER9 processor described in this paper, are generally scheduled for more than one tape-out. A common assumption is that not all bugs can be found in the pre-silicon stage. For this reason, a post-silicon phase is planned, targeted at finding the remaining timing-sensitive bugs by leveraging fabricated hardware.

Post-silicon validation differs from pre-silicon verification in many aspects. With post-silicon validation being conducted on real hardware, the system's scale and speed are far greater than what is available in pre-silicon. That said, the silicon platform offers little to no ability to observe or control the micro-architectural state of the design under test (DUT). These differences dictate the need for different approaches for stimuli generation, checking, and debugging. In this paper we describe the state-of-the-art post silicon validation process, used by IBM to verify its latest POWER9 processor. This process was the result of lessons learned from previous generations [1], as well as numerous innovations.

II. POWER9

POWER9 is designed for high-end enterprise-class servers[2]. It is one of the most complex processors ever created. The chip is fabricated using the GF 14nm fin field-effect technology and contains 8 billion transistors. It uses 17 layers of metal and exploits embedded dynamic random-access memory (eDRAM) for its L3 cache.

Figure 1 depicts a high-level block diagram of the IBM POWER9 processor chip. Each chip can have up to 24 cores. Each core is capable of 4-way simultaneous-multi-threading (SMT) operation, and can issue up to 9 instructions per cycle. The processor also comes in a 12-core variant, which supports up to 8-way SMT. POWER9 introduces a modular execution slice micro-architecture, with the inner core optimized as a two-dimensional matrix. Each core is built from multiple slices that are aggregated together. Each slice implements major pipeline functions such as dispatch, issue, execute, and cache access. A single SMT8 core or a pair of SMT4 cores are supported by a dedicated 8-way 512KB Level 2 (L2) cache and a 20-way 10MB eDRAM Level 3 (L3) cache.

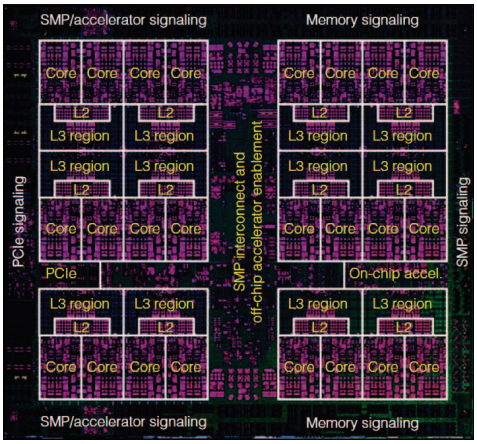

Fig. 1: IBM POWER9 chip diagram

III. Exercisers

Bare-metal exercisers were the primary method for test-case generation in the POWER9 post-silicon functional validation. An exerciser is a software application that is loaded directly to the DUT. The main input to an exerciser is a test-template that specifies the desired scenarios from a verification plan. The test-template language consists of basic instruction statements, sequencing-control statements, standard programming constructs, and constraint statements. Users combine these statements to compose complex test-templates that capture the essence of the targeted scenarios. Exercisers continuously generate random test-cases based on the template, execute them, and verify their results. No operating system (OS) is needed, as the exerciser is self-contained and includes basic OS services, such as exception handling and memory translation mapping. Once loaded onto the platform, it runs with no additional interaction with the environment.

An important aspect of exercisers, is the balance between those parts of test generation that can be done off-DUT, and those that need to be done on the DUT. Off-DUT computation enables using large computing resources and multiple libraries, but there is a hard limit on the amount of pre-computed data that can be packed into an image.

An example for such balancing between off and on-DUT, is the validation of POWER9's new Radix translation mode. To validate this mode, the OS translation table structure must be imitated. This type of problem is extremely difficult to implement in pure *c* code, while fairly easy to express as a Constraint Satisfaction Problem (CSP) (see [3]). This led to a decision to generate all required translation paths off-DUT

978-3-9819263-4-7/DATE20/©2020 EDAA

using a standard CSP solver and pack them into the exerciser image. This approach proved highly effective, when we needed to incorporate rare or hard to reach cases such as page migration (remapping an existing virtual address (VA), to a different real address (RA)), cross-thread Translation Look-aside Buffer (TLB) invalidation, translation table sharing between threads, nested translation, aliasing (several VAs translate to the same RA), VA sharing between threads, and variable length page walks. The cost was an increased image size, which required additional work to scale to large targets.

From a software perspective, bare-metal exercisers should be relatively simple, to enable easy pinpointing of hardware errors. Some important design principles are as follows: 1) Using assertions in interrupt handlers, to verify the correct state of special purpose registers, correct interrupt reason, and the correct handling of address translation and security faults. 2) Including self-checks as part of the generated test code. 3) Applying multi-pass comparison to interrupts, to verify that no interrupts were missed or incorrectly triggered by the interrupt controller. 4) Each instruction written in the test template should succeed, i.e., it should correctly complete on the DUT, unless explicitly stated otherwise. This allows the test engineer to freely express the test intent in the template, while the tool verifies the correct generation and execution of the test. If an instruction triggers an exception (e.g., page access violation), the tool corrects it and re-executes the instruction, much like an OS would.

IV. Pre-lab phase

Lab preparation starts after the end of the previous processor's lab and spans the entire time-frame until the first silicon samples arrive from the fab. Our goals for exerciser lab readiness are the following:

1) Run all of our templates, without fails, on the Instruction Set Simulator (ISS) and acceleration. This indicates that our tool and templates are free of SW bugs. 2) Run all of the possible templates in a silicon Cycle-Reproducible Environment (CRE) [1]. A special hardware mode in which executing the same image twice would produce the exact same micro-architectural state. 3) Make sure that exerciser images are ready, in terms of time-to-build and image size. This is to avoid long load times in the lab environment. 4) Enable fast-forward capability, which allows us to recreate a fail by setting the exerciser tool's state so it can generate the failing test-case directly, rather than reaching it after a long run.

A. Software Development

A failure encountered during an exerciser run, could be caused by either an environment setting, a hardware bug, or a software issue (in the exerciser or the template). Software issues during the silicon lab are difficult for HW designers to detect. We cannot use regular off-the-shelf SW testing tools, since an exerciser is not compiled as an executable to run under an OS. Therefore, most of our continuous integration process is done on an ISS. We use Jenkins to run all possible configurations on the entire lab test suite, thus detecting both tool and template errors. Additional hard-to-hit SW timing issues, which are not exposed by the ISS, can be detected on the accelerator. Finally, some SW issues are so rarely triggered that they can only be found in the silicon environment. Since we do not have the current developed project's silicon to test on yet, we use (N-1) regression, running most of our templates on POWER8 machines. This requires maintaining code compatibility with previous designs.

Another important aspect of software development is the performance of the tools. For exercisers, utilization is measured by the number of test-cases generated per second. We use Jenkins reports on ISS runs and real cycle counts from EoA to measure and improve our performance. During the course of the POWER9 project, we were able to improve the utilization of the Threadmill exerciser [4] by more than 25% just by profiling and refactoring of the code.

B. Acceleration

The acceleration environment is the first environment in which exercisers run that contains the actual design. IBM has an in-house AWAN simulation acceleration platform [5], which is characterized by high speed and observability. EoA proved useful in finding hardware bugs, that escaped the HW simulator. The *shift-left* methodology dictated running EoA earlier in the project. In POWER9, roughly 6% of all functional bugs were found by running exercisers on accelerators, compared to 1% in POWER8.

Another important aspect of acceleration is its ability to generate RTL coverage from synthesized coverage monitors. We specifically target areas that are less covered by the simulation tools [6], as well as closing coverage holes in the exercisers themselves. See additional details in IV-D. Acceleration budget per exerciser, measured by cycles, is allocated differently at each point in the project. We use the simpler exercisers more at the beginning of the project, as they are easier to run and debug. We monitor the coverage, and run-time of each tool using Verification Cockpit (see [7]).

Each image generated by an exerciser is designed to run on the ISS, acceleration, and silicon with only very minor changes. This allows a failed image from the lab to be used as-is in the accelerator, which is more observable. Exercisers can detect their environment and automatically limit features that are only supported on silicon (e.g., very long loops, deep power states).

When the bug rate drops, we employ several more techniques. First, we use hardware irritators [1] to help trigger some new bugs, for example by artificially reducing cache sizes and queue depths. Second, we select our best templates based on both the RTL coverage of a template, and on its bug finding history. We run these templates for 10 to 100 times longer than usual on the accelerator. This step was a lesson learned from POWER8, where several early lab bugs could have been hit in EoA if the templates had run for longer periods. In some cases, finding such a bug would postpone an upcoming tape-out, and could even prevent an additional one.

C. Triggering Bugs, Stimuli Generation

In POWER9, exercisers used the same techniques described in [1]: thread irritation [8], macros, and hardware irritators. We describe here some additional concepts that were employed and proved useful.

First and foremost, we use a Design For Debug (DFD) methodology in our test templates. When writing a new scenario, we consider the possible bugs it can hit, the way they will be detected, and our debugging abilities on silicon. If a potential bug is assumed to be too difficult to localize, we find ways to change the scenario. A good example for that would be a load/store exclusive scenario. Here, all N threads

increment a shared counter in a loop, X times each. Hence, the final value of the counter should be $N * X$. The most likely bug would be two threads entering the critical section together, causing the final counter value to be $N*X-1$. For a large N it is almost impossible to locate when exactly the fault occurred, even in acceleration. We came up with two possible mitigations. One adds a self-check right after the counter is incremented by each thread. However, this was too intrusive and could conceal some of the bugs. We further improved the template by having each thread increment the counter by its own ID, or increment the counter by an ever increasing value ($1, 2, 3, ...$, instead of always 1). In this way, assuming we have a single missing incrementation, we can know exactly where it happened in the test-case: the missing value from the counter is the iteration number that is missing.

Another important concept was to test real-world OS-like scenarios. Debugging bare-metal exercisers that are built for this purpose is much easier than debugging the real OS. Examples include OS-like context switching, exception handling, emulation of page swapping by translation invalidation, and entering into power-save modes and wake up.

Finally, we allow as much randomness as possible in our scenarios, without impairing the user's intent. Exercisers have the advantage of running large numbers of cycles. We exploit this advantage by randomizing each aspect of a scenario. For example, a load/store scenario will probably use all types of loads/stores, with all possible offsets and alignments, to different or same cache lines and cache congruence classes, with a different order between test-cases. We try to cover many features using the same scenario. In POWER9, many bugs were found in this off-target manner. E.g. a bad exception handling bug caused by an exceptionally wide access load that spans several cache lines, was triggered by a transactional memory test that changed the commit order of those loads.

D. Coverage Closure

The first step in achieving coverage closure uses the highly instrumented EoA environment. We always prefer to close coverage holes in the most general possible way, adding built-in support in the tool itself, or modifying existing test templates to cover new features. A good example for this from POWER9 was adding load/store cross-page access. It proved optimal to have the tool allocate memory adjacent to page crosses. This enabled page crossing in all test templates, rather than only when explicitly directed by the user. Another example involved one exerciser missing a nested branch speculation bug. Rather than writing a test exactly for this purpose, we modified the exerciser so it would randomly generate deeper speculative paths whenever a branch was invoked. An example for sub-optimal coverage closure was validating POWER9's sliced design. Slices enable one instruction to be split between two, or three HW execution pipelines, for parallel memory access. This introduced many verification challenges, such as triggering several exceptions by the same instruction, from different memory slices. Some of the bugs involved incorrect prioritizing of exceptions, or incorrect values loaded into special registers. Exercisers added validation for this feature only in specific templates, which proved inefficient in finding some of the above errors. A better approach would have been to add an abstraction in the tool itself, so it would generate each memory access with the aim of stressing the slice mechanism, thus testing this feature in all legacy templates.

In POWER8, the second step in achieving coverage closure involved choosing the correct test suite for the post silicon lab, out of the numerous test templates and parameters that exist. Many recent works address this effort, and propose various ML and coverage-based pruning techniques. Towards POWER9, this effort was superseded with generalizing and drastically reducing the number of templates, with the goal of running the entire test suite in the lab. Additionally we used the *Kitchen Sink* approach. We added seemingly unrelated stimulation to directed templates, to create a cross-product of micro-architecture events. In POWER9, several highly complicated multi-unit bugs, were found using this method. One example involved incorrect system resource restoration after exiting system power-state, and transactional memory. Previously, both of these were only tested independently. Finally, we used Template Aware Coverage (TAC) (see [9]): All exercisers and simulations ran under a joint TAC policy that recommends the next test to run by looking at past coverage statistics. This increases the coverage of lightly-hit events at the expense of well-covered areas.

V. Lab

The lab triage process consists of reviewing fail logs and clustering them by similar fail signatures. Contrary to pre-silicon, in post-silicon it is vital to classify a fail in the correct cluster, because a bug may only be hit one or two times during the entire bring-up lab. In POWER9, we employed a unified semi-automatic triage for all exercisers. This mitigated a major work item from POWER8, where lab and exerciser teams wasted precious time debugging different instances of the same fail signature. The scripts inspect every fail log in a given time window, and search for complex user defined regular expressions, that were added for each new fail signature.

The open bug list is prioritized daily. We mostly preferred to switch to an easier fail to debug, especially if it was in the CRE. We consider root cause analysis complete only when a bug is reproduced in all three environments: simulation, EoA, and lab. This meant the coverage hole was sealed and enabled a thorough verification of the fix before the next tape-out.

We used the lab to study our best templates and find out what makes them better at finding bugs. These templates are not necessarily the ones that cover the most RTL events, but may contain a "secret sauce" that causes them to hit bugs that should be identified, e.g a certain mix of instructions. Finally, when the lab bug rate drops we further stretch the environment to expose new bugs, by randomizing hardware irritators and lab environment parameters, as well as tool and template parameters.

VI. Debugging

The best-practices used in the IBM post-silicon lab during the validation of the POWER9 processor were similar to those described in [1]. The silicon CRE was the preferred environment for debugging. In POWER9, it included two SMT4 cores, L2 cache, and a 10MB region of the L3 cache (annotated in Figure 1). Having two separate cores in the CRE allowed us to efficiently debug issues that required inter-core interaction. We also use the CRE to eliminate the possibility of an electrical bug by running exactly the same single-core image on several cores. If only one of the cores fails, we suspect an electrical bug. Next, we describe two bugs from the POWER9 bring-up, their impact, detection method, debugging process, and the reasons for their escape to the lab.

The first bug caused a non-cacheable load to be executed twice. The fail occurred at the system test stage running the AIX OS and was observed as a software hang (live-lock) due to a lost interrupt. The OS code reads a register that returned an erroneous status of pending interrupt. Reading this register has an additional side-effect of cleaning it. Thus, if there are two read requests instead of one, the first would clean the register then the second would return the wrong data. The first step in debugging was to reproduce the fail using a bare-metal exerciser test. We did so by taking the instruction stream of the fail and incorporating it, in a generalized manner in an exerciser template. A HW irritator was used, to decrease the size of a specific micro-architectural queue, so it would fill faster. The next step was to analyze fail traces from similar previous bugs, to find the differences between them and and the new bug. All traces shared three conditions: two full micro-architectural queues, and a non-cacheable load that hits the same TLB entry of another load, executed a short time before it. Only one of these conditions was unique to the new bug. The EoA team used this information to modify their directed tests to contain the new ingredient. Additionally, an RTL checker was added to the design, to indicate the filled queues. This escape occurred because there was no detection mechanism for this kind of error in the exercisers. Moreover, it required a specific micro-architectural state (two concurrently full queues), which was not covered by an RTL event (thus, missed by the coverage review). Finally, the bug could have been exposed earlier if the correct HW irritator had been used more extensively. The "Kitchen Sink" approach introduced in POWER9, that added multiple random non-cacheable accesses into many templates, was a required condition to hit this bug.

The second bug was causing a core hang while running an exerciser in the CRE. The bug was caused by constantly flushing a load instruction due to irritation by another core, preventing forward progress. The load instruction was reading a Strong-Access-Order (SAO) memory address, while other threads irritated this cache-line with TLB-invalidate instructions. Note that it was not a test targeted for testing the SAO feature. Nevertheless, randomizing this attribute of the translation entries helped expose this bug. Debugging in this case was fairly easy, since it occurred in the CRE, where relevant waveforms from right before the hang can be easily collected. The reason for this pre-silicon escape is that in EoA, we could not employ TLB-invalidation irritation at full speed, due to the inherent characteristics of this environment. Therefore, this bug could only be exposed in a full chip system. Again, the "Kitchen Sink" approach was used to introduce random Strongly-Accessed-Ordered memory into all templates. Additionally, we used TAC to improve this template's coverage.

VII. Results

Several factors cause logic bugs to be identified only at later stages of the bring-up lab. These include triage errors, limitations on the manual debug time of the lab team, overcoming setup and manufacturing problems, changes in the lab configuration, changes applied to the hardware or software to work around bugs, extreme rareness of certain bugs, and additional tape-outs of the chip that could expose additional bugs that were hiding "behind" known ones. Therefore, the bring-up lab is a continuous effort. The POWER9 bring-up was considered to be highly productive. The team was able to handle 30% more logic bugs in 80% of the time, compared to POWER8, largely through increased usage of the CRE. The increase in the number of bugs was due to an extensive redesign of the core and late availability of new features.

	POWER8	POWER9
% bugs found in EoA	1%	6%
% bugs found in PostSi	1%	4%
Days to root cause 90% of bugs	31	17
ML-based coverage	no	yes
OS-like scenarios	no	yes

TABLE I: Validation of POWER8 vs POWER9

Overall, about 4% of all POWER9 bugs were found in post-silicon validation, compared to 1% in POWER8. The following four units had the most post-silicon logic bugs: 30.9% of bugs were found in the load-store unit, 10.1% in the bus bridge unit, 8.5% in the sequencing unit and 7.8% in the pervasive logic. Table I shows the improvement in the post-silicon validation of POWER9 vs POWER8 processor.

VIII. Conclusion and future work

There is a growing need for fast and comprehensive acceleration and post-silicon validation to ensure a processor's quality prior to its shipment. We presented the approach taken by the IBM team during bring-up of the POWER9 processors. Meticulous preparation has made the bring-up a success, despite a comprehensive re-design of the entire micro-architecture and a radically new memory translation mode.

We believe the following topics in post-silicon validation merit further research. The largest area for improvement is the post-silicon debugging process. Lab debug and root-causing is chiefly a manual effort, conducted by experts. Automating some of the debug process could reduce time-to-root-cause, and increase the lab team's bandwidth. This could include automatically finding a *Minimal Working Example* of the failing test, in terms of both time-to-hit and test-case length.

Another challenge is fully stressing large systems in a bare-metal setting. Concurrently fully utilizing several resources such as buses, arithmetic units, and memory controllers is usually only achieved in a much later stage, typically when the OS and advanced performance tools are available. Shifting some of these techniques to the exerciser stage is an open and important research goal.

References

[1] T. Kolan, H. Mendelson, A. Nahir, and V. Sokhin, *Post-Silicon Validation of the IBM POWER8 Processor*. Springer International Publishing, 2019.

[2] S. K. Sadasivam *et al.*, "IBM power9 processor architecture," *IEEE Micro*, vol. 37, no. 2, pp. 40–51, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.40

[3] M. Aharoni *et al.*, "Using graph-based CSP to solve the address translation problem," in *CP*, 2016, pp. 843–858. [Online]. Available: https://doi.org/10.1007/978-3-319-44953-1_53

[4] A. Adir *et al.*, "Threadmill: a post-silicon exerciser for multi-threaded processors," in *DAC*, 2011, pp. 860–865.

[5] J. A. Darringer *et al.*, "EDA in IBM: past, present, and future," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1476–1497, 2000.

[6] A. Adir *et al.*, "Reaching coverage closure in post-silicon validation," in *HVC*, 2010.

[7] M. Arar *et al.*, "The verification cockpit - creating the dream playground for data analytics over the verification process," in *HVC*, 2015, pp. 51–66. [Online]. Available: https://doi.org/10.1007/978-3-319-26287-1_4

[8] A. Adir *et al.*, "Advances in simultaneous multithreading testcase generation methods," in *HVC*, 2010, pp. 146–150.

[9] R. Gal *et al.*, "Template aware coverage: Taking coverage analysis to the next level," in *DAC*, 2017, pp. 36:1–36:6. [Online]. Available: https://doi.org/10.1145/3061639.3062324