Post Silicon Validation of the MMU

Tom Kolan[1], Hillel Mendelson[1], Vitali Sokhin[1], Shai Doron[1], Hernan Theiler[1], Shay Aviv[1]
Hagai Hadad[1], Natalia Freidman[1], Elena Tsanko[2], John Ludden[2], Bryant Cockcroft[2]
[1]IBM Research. Email: {tomk, hillelm, vitali, dshai, Hernan.Theiler, Shay.Aviv, hagaih, natalief}@il.ibm.com
[2]IBM Systems & Technology Group. Email: {etsanko, ludden, cockcrof}@us.ibm.com

I. Abstract

Post silicon validation is a unique challenge in the design verification process. On one hand, it utilizes real silicon and is therefore able to cover a larger state-space. On the other, it suffers from debugging challenges due to a lack of observability into the design. These challenges dictate distinctive design choices, such as the simplicity of validation tools and a built-for-debugging software design methodology.

The Memory Management Unit (MMU) is central to any design that uses virtual-memory, and creates complex verification challenges, especially in many-core designs.

We propose a novel method for post silicon validation of the MMU that brings together previously undescribed techniques, based on several papers and patents. This method was implemented in Threadmill, a bare metal exerciser and was used in the verification of high-end industry-level *POWER* and *ARM* SoCs. It succeeded in increasing RTL coverage, hitting several hidden bugs, and saving hundreds of work-hours in the validation process.

II. Introduction

Modern day Systems on Chip (SoC) are becoming more and more complex, implementing multiple levels of functionality. To verify that the designs work as intended, the SoCs go through a rigorous functional verification process, from pre-silicon verification in the design stage to post-silicon validation after fabrication. This verification is crucial. Fixing a bug after an SoC has been shipped to customers can be expensive and could require another tape-out or even a recall. Furthermore, it harms customer trust and may cause future revenue to decline. Post-silicon functional validation is the last step in the verification effort of a processors design, before it goes into mass production. Improvements in pre-silicon design verification methodology cannot yet bridge the design-verification gap, and the role of post-silicon validation continues to be significant. Moreover, according to one study [1], 75 percent of processors today go through at least 2 tape-outs, and 25 percent go through 3 or more tape-outs. The reasons include a tight design schedule that limits core verification time and the extreme scarcity of some bugs (e.g., "SAO/TLB-invalidation" [2]). The post-silicon phase complements that of pre-silicon, with the added ability to leverage the fabricated hardware to find these difficult bugs.

Bare-metal exercisers are an essential post-silicon tool, used by many companies (see [3]). An exerciser is a piece of software that is loaded to the Design Under Test's (DUT) memory, and runs an infinite loop of pseudo-random test generation, execution, and checking. It generates tests on-target to avoid the I/O bottlenecks associated with on/off-loading. Because exercisers run on the silicon without an OS, hypervisor, or any other supporting software, they must be self-contained and implement memory allocation, address translation, exception handlers, and print methods.

A key component of many modern CPUs is the Memory Management Unit (MMU), which performs virtual to physical address translation. Every memory access, including those to I/O and other devices, is to a virtual address (VA) that the MMU translates into an actual physical address (PA). Address translation is the basis for virtualization, which lies at the heart of all modern OS, data centers, and cloud platforms. Thus, a bug in this mechanism can render the processor unusable. Hardware implementations of this mechanism are complex and consist of several pipelined layers of caches such as Translation Look-aside Buffers (TLBs) and ERAT, in-memory translation logic, and table-walk logic.

Pre-silicon verification of address translation cannot cover the full state space of the MMU logic. While it covers table-walks and specific sequences for replacing translation entries well, it can only scale up to a handful of cores. Thus, it misses corner cases that depend on rare timing combinations between the MMU and other micro-architectural state machines, such as the Load Store Unit (LSU). One of the goals of post-silicon validation is to cover these corner cases. The challenge is to produce rich, high-quality translation stimuli by generating a large variety of translation paths. These paths should span all possible translation levels, with a focus on resource sharing, e.g., the same real address used by two or more translation paths, the same virtual address used in different contexts, or different memory allocations translated using the same page. Another challenge is to create real-life OS scenarios, such as context switches and page migration, in a randomized manner that will stress the DUT into interesting corner cases.

Existing pre-silicon solutions for address translation such as those by Adir et al. [4] [5] are too slow; they will result in poor utilization of the silicon and low overall coverage. Existing post-silicon solutions are too simplistic; they mostly use a constant offset to map each VA to a PA, and the resulting translation entries are not shared between different paths. They usually lack the abstraction level and built-in mechanisms to create stimuli which emulates interesting real-life scenarios.

In this paper, we present a novel solution for post-silicon MMU validation, which is particularly well-suited for exercisers. We implemented it on Threadmill, a state-of-the-art exerciser [6], for both *POWER* and *ARM* architectures. The solution leverages the speed of the platform to reach corner cases of the design and found several hard-to-hit bugs.

The rest of the paper is organized as follows. Section III describes our solution and Section IV describes translation related test scenarios, made possible by it. Section V provides results for the RTL coverage of *IBM POWER10* MMU, and compares our solution with other exercisers. Additionally, the section describes hard-to-hit bugs found by our system. We

978-3-9819263-5-4/DATE21/©2021 EDAA

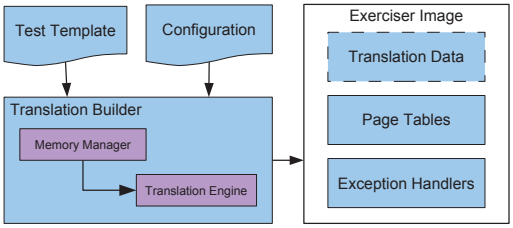

Fig. 1. Translation validation system

present our conclusions in Section VI.

III. System Design

The MMU implements a contract between software and hardware that indicates how VAs are mapped to PAs. Typically, the OS is responsible for creating the address translation mapping, for initializing and maintaining it in memory, and handling exceptions in case of translation faults. The hardware MMU allows the software to access physical memory on the device by carrying out the given translation, and by handling functional attributes of the memory (e.g., protection, caching, and shared memory). Thorough verification of such translation mechanisms requires sophisticated test-generation methods, including the use of constrained-random stimuli generators [7].

We chose to create the translation mapping off-target, at the time the test image is built. The tables are packed and embedded into the image, which is loaded onto the target machine. Whenever a memory address is accessed for the first time, these tables are used by our exception handlers to install the translation path in memory. When the CPU returns to the test code, it can then continue its execution. We considered several trade-offs when choosing this method: build time vs. exception handling time, size of the test image, and verification quality. We opted for off-DUT translation path generation, because it allows us to utilize constraint satisfaction solvers, and offers the most flexibility and randomness. Moreover, from a software engineering perspective, it allowed us to build a standalone package that was reused in several other projects.

As shown in Figure 1, our system comprises the following components: (a) off-DUT Translation Builder consisting of a *Memory Manager*, which generates random memory allocations, and a *Translation Engine*, which creates a variety of address translation paths for each allocation; (b) *Translation Data*, which may be part of the image loaded into the DUT and used to create address translations in the Page Table area. To reduce the number of translation interrupts, it is possible to have Page Tables created off-DUT and pre-loaded as part of an image. In this case, Translation Data is not added to the image; (c) *Exception Handlers* for installing address translations in memory and handling translation issues, either unintended or as part of a user directed scenario.

A. Memory Manager

In the post-silicon test environment, physical memory is limited in size. Hence, the allocation of intervals in memory becomes a challenge when the test case requires many intervals of many different attributes. For example, in the ARM architecture, there may be hundreds of different combinations of memory attributes. To achieve good RTL coverage, it is also desirable to: allocate the intervals across the entire physical memory, create false-sharing cases, and generate translations for these intervals with a variety of page sizes (e.g., not only small pages of 4KB). Moreover, certain intervals would typically be translated using only large pages in order to increase the utilization of specific micro-architectural resources.

Translating intervals with large pages imposes significant constraints on their allocation, since memory attributes need to be consistent across all overlapping pages, while virtual pages of different sizes can overlap with each other in the physical memory domain. Our test generator provides a solution for allocating the intervals in memory such that the physical memory can be translated with large pages while maintaining the consistency of its attributes.

Our Memory Manager allocates memory according to the specifications of the design under test (DUT), the physical memory configuration of the post-silicon platform, and the user requirements extracted from the test template. The Memory Manager is composed of two stages:

Coloring Stage:

In this preliminary coloring stage, we divide the memory into different segments according to the given ISA, the DUT memory configuration, and the user requirements written in the test template, such as desired page sizes and attributes. Examples for attributes are protection bits, memory cacheability type, and valid bit setup. We: (a) Identify all the intervals that need to be allocated and group them together according to their attributes (aka color-group), summing up the space required by each group. (b) Segment the available memory into different colors, such that fragmentation would be minimal yet still yield an allocation that is interesting for verification purposes.

Algorithm 1 Memory intervals allocation

Data: C is the set of all colors; $I = \{I_i^c\}$ is the set of all intervals where $c \in C$; K is a constant parameter.

Coloring:
for c in C **do**
Allocate K segments $S_1^c, S_2^c, ..., S_K^c$ in memory for this color, such that $\Sigma_k Size(S_k^c) \geqslant \Sigma_i Size(I_i^c)$.
end for

Allocation:
for I_i^c in I **do**
1. Allocate I_i^c in one of the segments S_k^c, s.t.:
- The interval's color matches the color of the segment
- The interval is fully contained within the segment boundaries
- The interval creates (randomly biased) false sharing

2. If such an allocation is not possible, go back to coloring stage, allocate larger segments, and start the Allocation stage again.
end for

Allocation Stage:

The objective of this stage is to optimize the allocation of physical memory, such that intervals are placed in memory according to the coloring done in the above stage. We allocate the intervals in physical memory according to their attributes (i.e., color) and the preliminary coloring segmentation. Each interval is allocated into one of the color segments of its color-

group. Additionally, intervals are allocated in such a way that they share the same cache-line and page with as many other intervals as possible; or, alternatively, across cache-line and page boundaries, such that at execution time they trigger several memory accesses and translation exceptions.

The final step before invoking the Translation Engine, is to determine the set of page sizes used to translate each interval. The maximum page size is the maximal page size that overlaps with the interval but will not intersect with adjacent segments of a different color. For more details see [8].

B. Translation Engine

The common address translation process passes through a sequence of steps and can therefore be naturally described as a directed acyclic graph (DAG). Using a constraint satisfaction problem (CSP) solving technique, we developed a framework we call *graph-based constraint satisfaction problem (GCSP)* solving. These problems consist of a DAG, combined with a CSP, where each variable and constraint of the CSP is linked to a particular node or edge of the graph. A solution to the problem is an end-to-end path in the graph, such that all constraints defined along this path must be satisfied. We base our algorithm for solving GCSPs on conditional CSPs [7].

For each physical interval, we generate one or more virtual pages that are mapped to it, referred to as "virtual aliases". Large physical intervals may require more than one virtual page to cover them. On the other hand, a single virtual page can cover more than one physical interval, if they are small and dense enough. We generate translations for every possible translation context, for example one- and two-stage translations, secure and non-secure memory. For each translation context, we generate virtual page(s) for each interval, while randomizing translation parameters as much as possible (e.g., virtual address, page sizes, granule size, and memory attributes). On target, the exerciser executes the test-case code while randomly and frequently switching between various translation contexts and various translation aliases. We further stress the MMU as follows: 1) Reuse of the same virtual addresses across different translation contexts, by the same or by multiple threads. These VAs are randomly mapped to the same or to different physical addresses. 2) Sharing of translation entries across different pages in different translation paths, tables, and levels.

C. Data Structures and Their Usage

The output of the Translation Engine is a data structure that holds the actual translation tables, including additional meta data used by the on-DUT exerciser.

For each context there is a unique set of translation tables, packed into the image as `C-struct` arrays. The structures are organized as a Database (DB) of table entries, each including {VA, PA, page-size, and indexes of raw entries along the translation path of that page}. We require this DB to be efficiently searchable according to different attributes, including VA and PA. To accomplish this we keep several sorted pointer lists, which allow for a binary search according to each attribute. For test generation, whenever a memory access is initialized, a PA to VA search is needed. For exception handling during test execution, a VA to PA search is executed allowing for on-demand initialization of the tables in physical memory and entry modification. This modification is done by injecting errors into existing installed entries in memory. The subsequent faults and exceptions that arise are fixed by restoring the original legal values from the DB. These include page invalidation anywhere along the translation path, migration, and the changing of attributes such as memory type, security privilege, access permission (read/write/execute), and shareability.

D. Exception Handlers

Since an exerciser is a bare-metal program that runs without an OS, it requires exception handlers to be embedded inside the tool. Unlike OS handlers, which are usually focused on efficiency and security, our handlers are geared towards validation and the constant continuation of the user scenario. Each handler begins by printing debug information to the trace log. This includes the address of the instruction that triggered the exception (`DAR` in POWER / `FAR` in ARM), the exception reason (`DSISR` / `ESR`), and the cycle count SPR (`TB` / `CNTCV`). It then updates a counter per interrupt type, so these can be compared over multiple runs of the same test-case (multi-pass consistency checking). For example, the `interrupt priorities mechanism` can have a bug where a privilege instruction interrupt and a debug interrupt occur on the same instruction, in an incorrect order. If the debug interrupt handler is invoked first, it will skip to the next instruction and the privilege interrupt will be missed. We assert several invariants that should hold:

1) The machine state register is updated accordingly when entering an exception. For example, in POWER, verify that a Hypervisor Data Storage Interrupt (HDSI) sets the privilege level to Hypervisor.
2) The address of the instruction that triggered the exception should be a valid VA in our translation DB, and point to a PA within the test-case boundaries. The kernel code should not trigger exceptions, nor should the `DAR` point to address `0` (a common HW bug).
3) The VA that triggered the exception is the expected one. For this purpose we keep a partial reference model that enables us to expect a certain range of exceptions and addresses per executed PC [9].
4) SPRs that shouldn't be updated by an exception should remain unchanged. To ensure this, we reset some SPRs to `0` at the end of each exception handler. For example in ARM, if an exception is triggered in `EL2`, only `ELR_EL2` should be modified, and we can assert that `ELR_EL0, ELR_EL1, ELR_EL3` remain `0`.

Finally, we try to handle every exception in a way that will allow the CPU to return to the same instruction, and successfully execute it without re-triggering the same exception. This is useful for several reasons. It preserves the test-case intent, allowing non-perfect test generation and initialization, knowing that the exception handler would fix any faulty conditions that may prevent the user scenario from executing as planned. We create multiple exceptions on the same instruction (see [10]). This is important to validate many possible combinations of multiple pending exceptions. Therefore, we fix only a random subset of the entries that need to be updated. For example, in a page fault exception, we turn the Valid bit to 1; in a protection exception, we update the protection bits to allow the access. After that, the exception handler will conclude and the CPU can go back to executing the same instruction, triggering more exceptions. This allows us to create more variability and hit additional coverage events (see Section V).

IV. Test Scenarios

Some common real-life scenarios are important to verify, but difficult to implement in a bare-metal setting. To properly cover these use-cases, an exerciser should have built-in support for generating them. This would allow the user to add these scenarios in almost every test-template, rather than hand-coding them, thus increasing the cross-coverage and hitting bugs that involve interaction between the MMU and other design units (mainly the LSU, L2 which handles coherency, and the ISU). The following scenarios all use the "irritation template" [11]. The irritated VAs can belong to a data region or to the actual test-case instructions being executed (irritating the I-Caches). Importantly, any existing test scenario is suitable for irritation so that we can achieve full cross-coverage.

A. TLB-invalidate Irritation

TLB-invalidate instructions (e.g., `tlbie` in POWER, `tlbi` in ARM) are some of the most difficult for the processor to execute. They are typically broadcast to the entire core or even to the entire system, atomically invalidating the TLBs and possibly other tables. A typical data-side TLB-invalidate scenario includes several "victim" threads executing load/stores from VA_1; these may be any load/store instructions, of any access length and any offset. An irritator thread performs local and system-wide tlbi instructions on the same VA_1, or on other VAs to add more noise. Without a supporting design mechanism to throttle back the tlbis, this scenario could cause a HW hang. This type of irritation cannot flag bugs where the TLB-invalidate did not occur, but it can find bugs in the way that the TLB-invalidate instruction spreads throughout the chip.

B. Page Migration

Page migration is when translation tables are changed during run-time *within the same context*, such that a VA would point to a new PA, i.e., from $VA_x \rightarrow PA_1$ to $VA_x \rightarrow PA_2$. This would be similar to an OS moving a page on the physical disk, while the page is being used by a program. This requires a specific software sequence described by Algorithm 2. Verification of this scenario is aimed at finding TLB-invalidate bugs that cause the invalidation to not occur (or to not occur on time), thereby generating invalid or duplicate translation cache entries.

Our method follows Ibraheem and Kolan [12]. During build-time, we create groups of pages ("modification sets") of arbitrary size. The special property of each group is that all of the translations start from the same VA, but at some level of the tables, the paths split. This is where modification is done. Any change to this level's entries will cause the mapping to change to a different PA. Therefore, the modification handler can be invoked over and over again, and randomly change the VA to PA mapping to a different PA in the modification set. Importantly, we want one thread to modify another thread's translation; this is what happens in the OS scenario and self-modification is not difficult for the HW to implement.

The test scenario itself can modify translations for either the data or the instruction sides: (1) For data, we allocate a true/false shared array. Then we load or store to it from each thread, while one of the threads acts as the "OS" and performs the page migration. During this time, data access by other threads is blocked, because the translation entry is invalid, and causes an exception. (2) For instructions, we branch to the migrated reservation, execute some random instructions, and branch back. During this time, another thread modifies the VA to PA mapping, which is used by the tested thread as the mapping of the branch target page. Because the contents of the PAs are also swapped, execution can continue normally after the new translation is installed.

Algorithm 2 Page Migration

Thread 0:
for times **do**
1. Lock VA_1 page entry, so other threads cannot access it while being modified
2. Invalidate translation entry
3. Invalidate TLB
4. Modify translation
5. Swap data, so the victim scenario can continue from where it left off
6. Validate translation entry
7. Release lock

end for

Threads $1..N$:
loop
Perform algorithm under test involving loads/stores to VA_1
end loop

On top of the general scenario, we randomly generate all forms of TLB-invalidate instructions that can possibly invalidate a given migrated entry. The purpose is to create more stress and confusion across the system. Even though two threads ("OS"+"Application") are sufficient to expose some bugs, we found it beneficial for many threads to share the same context. In this way, they all experience the page migration at once. Some additional ingredients to detecting the triggered bugs are:

1) To preserve multi-pass consistency and enable *true sharing*, we swap the memory content between the "old" and the "new" physical intervals. This enables any scenario to continue undisturbed after a migration is done.
2) To ease debugging, we employ false sharing. Each HW thread stores to a constant offset in a common interval, and the stored value is the thread's logical index. This way, we can easily see which thread "missed" its store, when a bug occurs.
3) To catch bugs closer to where they appeared, we use self-checking. This involves loading the value from the modification interval after the modification and asserting that it contains the expected value.

C. Context Change

Context change is one of the most heavily used sequences in operating systems and hypervisors. It is used to switch between programs that run on a certain HW thread. Effective and quick context changes are key to gaining high performance in modern day workloads. In POWER, the context change itself is done by changing the `LPIDR` and/or `PIDR` values; and in ARM, by changing the `TTBRx` value. Projected on the MMU, a context change is an alteration of the translation context of a running HW thread, thus changing the VA to PA mapping of the thread (using tlb-invalidate, translation entries updates, and more).

The main differences from the above page migration scenario are: *(1)* Although both involve changing from $VA_x \rightarrow PA_1$ to $VA_x \rightarrow PA_2$, here we do so by changing the entire translation

context rather than modifying the existing one. *(2)* This is a single threaded scenario, since the thread switches its own context rather than waiting for some other "OS" thread.

Papadimitriou et al. [13] use one VA pointing to two different PAs in two different contexts. After a context change, any data previously written by the thread to this VA should no longer be accessible. We expand on this idea, creating a full VA to PA bipartite graph following Kolan and Mendelson [14]. During image build-time, we create "context change groups" of `N>=2` physical pages that are mapped to the same `N` virtual pages in *every* context. When switching between two contexts, a specific VA is randomly mapped to a different PA. This allows us to verify that the MMU has updated its tables correctly and will use the data from the new PA after the context switch. This full graph enables using any of the VAs by any of the threads, creating multiple false sharing scenarios in the physical domain.

Similar to the page migration scenario, the context change scenario can be done for data and/or instructions. For data, we load/store from the same VA before and after the context change. For instructions, see Algorithm 3.

Algorithm 3 Instruction-side context change

1. Branch to the PA interval, which is part of the "context change group"
2. Execute some random instructions
3. Change the context, swapping the data into the new PA, so the CPU can continue reading the instruction stream; this is done using a system-call if running in user-level
4. Execute some random instructions
5. Branch back to the test

D. Light Context Change

In this method, which follows [15], we artificially create new "light" contexts that have the exact same translation mapping as other existing contexts. This is done instead of creating more "full" contexts, which require costly building of the entire translation structure and inflating the image. The purpose of this method is to create stress by filling μ-arch caches like the TLB. These caches store data tagged by the context, so they can be "fooled" by creating almost-duplicate entries at a very low cost to the exerciser. This allows a faster uninterrupted test execution that can hit some unique ERAT bugs (see Section V-B3).

In POWER, this is done by copying entries in the partition table and/or the process table into different entries; for example, entry 0 in the partition table is copied into entry 64. This means we can randomly change the `LPIDR` value from 0 to 64 (and back), anywhere in the test-case, without executing the full software sequence for context change. Specifically, no barrier instructions are required. The translations remain the same, so no additional computation is needed but the TLB will be filled by these new translation "aliases". In ARM, the implementation is done by copying the entire first level of translations to a different location, and randomly updating `TTBRx` to point to this new location.

When interleaved with "real" context changes as described above, this method enables us to hit more delicate corner-case bugs. This occurs because when the CPU carries out the "real" context change, it needs to invalidate many more entries of the TLB, because the it is full, thus the chances of erroneous behavior increase. For example, in POWER, two TLB entries that differ only in their `LPIDR` value, may be added to the same set of the TLB. A good stress scenario would over fill a TLB set to cause many evictions of TLB entries, while issuing TLB-invalidate instructions on the same entries.

E. Translation Attribute Changes

We implemented a set of functions that can be invoked and used to randomly change many attributes in translation entries, such as valid bit, protection bits, and reference/change bits. Because the translation entries are in memory, these functions can be applied by any thread to any other victim thread in any context. When the victim thread performs a data access, the CPU encounters the corrupted entry during the table walk. It then notices the wrong attribute and takes an exception. Since our kernel controls the translation database (see III-C), it has all the information required to fix the entry back to the value that will allow forward progress. We randomly invoke a combination of these functions in test-cases to create a "storm" of interrupts, which stresses many aspects of the design.

A significant part of this scenario focuses the above technique around Segment/Page/Cache-line crosses. This way, each data access can incur several different exceptions on the different pages it accesses. Additionally, we mix in other kinds of exceptions such as alignment and debug exceptions, to further stress the CPU and verify that they are handled in the correct order as defined by the ISA.

Moreover, we deliberately create a "race" between the threads that are invalidating the translations, and the ones trying to fix them so they could access the data, while making sure the latter are slightly faster to prevent live-lock.

The bugs are detected by HW hangs (i.e., instructions execution can't make progress) or by assertions in the exception handlers (see Section III-D). In general, bugs in this area can be very severe, causing problem for the OS/hypervisor that are very difficult to work around.

V. Results

When we evaluated the method described in this paper, the results were very positive. They contain qualitative (complexity of triggered bugs), as well as quantitative (coverage) improvements. The method has led to shorter time-to-market of the *POWER* processors, with fewer field escapes that could have significantly affected the users.

A. Coverage Improvement

We measured the RTL coverage improvement obtained by implementing our method in Threadmill, against three methods: (1) ("SOA") The existing state-of-the-art method for validating the MMU, used by another exerciser of *POWER* designs. (2) ("Simple") Against a simple straight-forward MMU validation method, implemented by a third exerciser of *POWER* designs. (3) Core pre-silicon coverage.

Coverage of the final synthesized design was collected on an emulation platform. Each coverage event belongs to exactly one design "entity". We compared the coverage percentage of the translation-related entities. For the sake of brevity, we show only the entities in which there was a difference in coverage between the four methods.

Table I depicts the RTL coverage of the MMU entities, in the POWER10 core. The results show that in all but two entities, Threadmill's coverage is at least as good as the other two exercisers. Overall, Threadmill outperforms the simple

TABLE I
RTL COVERAGE (%), POWER10 MMU ENTITIES

Entity	Core	SOA ex.	Simple ex.	Threadmill
Exceptions	58.90	50.20	30.10	51.70
TLB invalidate	80.60	69.90	10.70	79.50
Context table	96.10	96.10	23.30	96.10
Invalidation #1	100.00	90.50	76.20	100.00
Invalidation #2	63.50	58.10	31.40	61.10
Pipeline	86.30	84.00	64.20	85.90
Thread reconfig	80.20	79.00	79.00	80.20
SPRs	62.90	61.80	53.90	61.80
TLB	86.60	76.50	53.60	80.90
Store miss queue #1	96.30	96.30	96.30	94.50
Store miss queue #2	61.60	50.00	26.90	56.60
Store miss queue #3	88.60	85.90	80.40	77.70
Total	83.25	76.31	51.90	80.02

exerciser by 28%, and the state-of-the-art exerciser by 4%. Moreover, the gap between Threadmill's coverage and the core pre-silicon coverage is 3%, which is acceptable given that pre-silicon stimuli is much more directed and controllable.

B. Bugs Found

We briefly describe several unique bugs found in *POWER9* and *POWER10*, including the bug scenario, root cause, and detection mechanism. These bugs were critical and in one case even required two additional tape-outs to fully mitigate.

1) Incorrect register update in interrupt: The bug was detected by asserting that some special-purpose register contains zero value in the beginning of a translation exception handler, since this register should not be updated by the exception. The *POWER9* LSU contains multiple slices so that different parts of the same ld/st instruction may be processed by different slices. In such a design, different parts of an instruction may cause different exceptions, but the LSU has to process the exception with the higher priority. In our case, one slice reported an alignment interrupt, which has a higher priority, while another reported a hypervisor page fault. Because of the bug, the hypervisor page fault was handled first, and the hypervisor level address registers were updated instead of the user level ones. To trigger such a bug, a test has to be designed to cause different exceptions by different parts of a store instruction, for example, one slice hits TLB with exception and another slice misses TLB, see IV-E for more details.

2) TLBI is not executed properly: The bug was detected by multi-pass consistency checking at the end of test, while a test performed page migration as described in IV-B. In this bug, a *writer* thread stores data using $VA \rightarrow PA_1$ translation. Another thread, the *page mover* thread, first invalidates the translation used by the writer thread. It then invalidates the TLB entry, moves the data, and finally validates the translation entry. The writer thread hits a page fault during the next access, updates the TLB and uses the new translation. But as it turned out, even though the data was moved after the *TLBI* had completed, the last store was not drained before the TLB snoop was completed. This caused a mis-compare at the end of test.

3) ERAT multi-hit: The bug was detected by an embedded HW assertion that there can be no more than one entry with the same *EA* in the ERAT table, for a given context. The bug was triggered by a context-switch test described in IV-D, and required nested user-level translations. The light context-switch happened while an instruction fetch missed the ERAT and was sent to the MMU to provide a translation and update the ERAT. The context-switch caused the fetch request to become invalid, but the MMU still updated the ERAT. Then, the LSU sent a new translation request for the same EA and context tag, but with a different index into the ERAT table. The request was served and another ERAT entry was created with the same EA, which triggered the assertion.

VI. CONCLUSION AND FUTURE WORK

We presented an end-to-end method for post-silicon validation of the MMU. Our method is based on existing bare-metal exercisers, and extends them to support a thorough validation of the MMU and its components. This method has shown good results on both *POWER* and *ARM* designs for several of the most complicated industry-level processors to date. Specifically, we were able to cover more RTL events and uniquely detect several novel critical bugs. The method continues to save a significant amount of labor in future designs, and helps secure revenue by reducing the number of tape-outs and field escapes.

There is still room for more work to be done. Here is a partial list of the future items on our plate:

1) Cover more built-in scenarios, especially sequences that are frequently performed by the OS and hypervisors.
2) Improve existing scenarios to cover more cases.
3) Achieve better coverage of page-cross, segment-cross, physical memory boundary accesses, etc.
4) More tightly control VA bits to create more VA reuse between different contexts, e.g., reuse specific bits in the VA for different hash functions in the design.
5) Add debugging mechanisms to shorten the time-to-debug of a post-silicon fail in the MMU.

MMU specifications are ever evolving, and new ISA features are constantly being added to them, for security reasons and for performance improvements. These features create new challenges for the MMU validation, and we believe that the method described in this paper is a good infrastructure to support most of these future ISA changes.

REFERENCES

[1] H. Foster. Functional verification study - 2018. [Online]. Available: https://verificationacademy.com/seminars/2018-functional-verification-study
[2] T. Kolan *et al.*, "Post-silicon validation of the ibm power9 processor," in *DATE*, 2020, pp. 999–1002.
[3] T. Kolan, H. Mendelson, A. Nahir, and V. Sokhin, *Post-Silicon Validation of the IBM POWER8 Processor*. Springer International Publishing, 2019.
[4] A. Adir *et al.*, "Deeptrans - a model-based approach to functional verification of address translation mechanisms," in *MTV*, 2003, p. 3.
[5] A. Adir, L. Fournier, Y. Katz, and A. Koyfman, "Deeptrans - extending the model-based approach to functional verification of address translation mechanisms," in *HLDVT*, 2006, pp. 102–110.
[6] A. Adir *et al.*, "Threadmill: a post-silicon exerciser for multi-threaded processors," in *DAC*, 2011, pp. 860–865.
[7] M. Aharoni *et al.*, "Using graph-based csp to solve the address translation problem," in *CP*. Springer International Publishing, 2016.
[8] S. Doron *et al.*, "Attribute driven memory allocation," Patent, 2017.
[9] T. Kolan *et al.*, "Utilization of partial results for post-silicon validation," Patent, 2020.
[10] T. Kolan *et al.*, "Recoverable exceptions for post-silicon validation," Patent, 2020.
[11] A. Adir *et al.*, "Advances in simultaneous multithreading testcase generation methods," in *HVC*, 2010, pp. 146–150.
[12] W. Ibraheem *et al.*, "Circuit modification," Patent, 2017.
[13] G. Papadimitriou *et al.*, "Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation," in *ICCD*, 2016, pp. 544–551.
[14] T. Kolan *et al.*, "Efficient translation context change testing for memory management unit verification," Patent, 2020.
[15] T. Kolan, H. Mendelson, and V. Sokhin, "Efficient translation table replication for memory management verification," Patent, 2020.