Unveiling Difficult Bugs in Address Translation Caching Arrays for Effective Post-Silicon Validation

George Papadimitriou[1] Dimitris Gizopoulos[1] Athanasios Chatzidimitriou[1]
Tom Kolan[2] Anatoly Koyfman[2] Ronny Morad[2] Vitali Sokhin[2]

[1] University of Athens, Computer Architecture Lab, Greece
{georgepap | dgizop | achatz}@di.uoa.gr

[2] IBM Research - Haifa, Israel
{tomk | anatoly | morad | vitali}@il.ibm.com

***Abstract*—Post-silicon validation is one of the most important parts of the microprocessor prototype chip lifecycle. It is the last chance for debug engineers to detect defects and bugs that escaped pre-silicon verification, before the chip is released to the market. Effective solutions are required to harness the peak performance of the hardware prototype and evaluate whether the microprocessor chip is fully compliant with the instruction set and other specifications. We perform a comprehensive experimental study on a state-of-the-art microarchitecture to assess and identify the most difficult bugs in address translation caching arrays (multi-level TLBs and MMU Caches), and explain why these bugs persist across generations. We also categorize them into distinct bug scenarios. We then propose a novel methodology for generating random self-checking stimuli programs, which expose and detect such bug scenarios. Our experimental results show that the proposed method can detect difficult bugs that are likely to be missed by traditional post-silicon validation techniques. [1]**

Keywords—post-silicon validation; silicon debug; design verification; difficult bugs; address translation; virtualization

I. INTRODUCTION

As the complexity of microprocessor designs continues to grow, we are seeing an increasing gap between their design and verification. Consequently, effective post-silicon validation techniques become even more necessary to detect design flaws that remain after pre-silicon verification and manufacturing defects in prototype chips. There are two distinct phases in microprocessor verification: pre-silicon and post-silicon. Pre-silicon verification mainly employs several simulation-based tools, but it suffers from the low throughput of simulators that run the detailed design. On the other hand, post-silicon validation on actual chip prototypes offers very high execution throughput, so design verification teams attempt to execute as many test programs as possible to obtain the largest possible validation coverage before massive chip production. For this purpose, post-silicon validation focuses on extremely large numbers of parameterized random test-programs. Unlike pre-silicon verification, post-silicon validation lacks observability and ease of control in the architectural and microarchitectural structures of the silicon prototype [1].

The address translation mechanism (ATM) of modern microprocessors is one of several complex mechanisms realized in state-of-the-art microarchitectures today. It is also probably one of the most difficult subsystems to validate for correctness ([2] [3] [4] [5]), due to the intricate functionality provided by its caching arrays. Address translation caching arrays (ATCA) such as TLBs and MMU Caches [6], lower the address translation latency by significantly reducing multiple memory accesses that aim to translate virtual addresses into physical ones. Several publicly available official errata reports from major microprocessor vendors present severe escaped bugs in ATCAs that persist across generations. These bugs affect both the performance and functionality of the microprocessor [7] [8] [9] [10] [11] [12] [13] [14] [15] [16]. Such bugs are difficult to detect using traditional validation tests. This occurs because they barely influence correct functionality, making their influence hard to observe.

Therefore, it is unlikely to take advantage of both prototype's increased performance and the verification of all potential bugs. This is especially true when there are rare bugs that manifest themselves only under certain operating conditions and their manifestation does not result in an incorrect architectural level output**.** Traditional post-silicon validation methods mainly rely either on comparing results to simulation-based reference/golden models, or on using multi-pass consistency end-of-test results [17]. However, both of these methods have a common drawback: they are incapable of detecting the bugs that do not affect the correct functionality of the validation program. This occurs because the checking phase is performed by comparing only the silicon's final outcome with a pre-generated reference model. As a result, when a bug gets masked by other valid microprocessor operations, the final output of the design under verification (DUV) will completely match the reference (bug-free) model. In short, the bug will not be detected at all.

A. Contributions of this Work

Our primary contribution is a novel post-silicon self-checking validation method that unveils and detects rare bug scenarios in ATCAs. ATCAs are among the most important structures for microprocessor functionality and performance. Escaped bugs in these arrays can lead to unpredictable system behaviors in the field.

Using a comprehensive experimental study, we present and analyze rare bug scenarios and why they are difficult to detect. Our goal is to unveil and detect difficult bugs in ATCAs. Even if bugs manifest themselves by executing traditional validation tests, detecting them is unlikely due to the high possibility of masking during the execution of the validation test. For that

[1] This work is supported by the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement 611404 and by IBM Research through an IBM Faculty Award.

reason, we propose a novel method that detects such difficult bugs in the silicon prototype.

Our validation method is self-checking and it does not require any hardware instrumentation. The method is easily applied to any CPU and ISA. Because it is software-based only, it avoids significant security vulnerabilities associated with embedded debugging mechanisms [18]. We demonstrate that, in contrast to traditional validation tests, our proposed method effectively detects all of the injected bugs we created (by using common use-cases of a real hardware prototype), according to bug scenarios shown in Table II. For our experimentation, we enhanced the Gem5 simulator with modern MMU Caches, as described in [19], to have a more detailed representation of a commercial microprocessor chip.

II. Impact of Bugs in Address Translation Caching Arrays

A. Motivation

ATCAs are the predominant source for severe escaped bugs in silicon prototypes. Escaped bugs or faults in other performance structures such as branch prediction units (BPU) or prefetchers, lead only to performance degradation and not to incorrect functionality [20]. However, a bug in an ATCA may affect both the performance and the correct functionality of the microprocessor. Massive published errata from microprocessor vendors demonstrate that severe escaped bugs in ATCAs can persist across generations (shown in Table I) [7] [8] [9] [10] [11] [12] [13] [14] [15] [16]. Clearly, traditional post-silicon validation methods are not adequate to detect specific kinds of bugs.

TABLE I. PUBLISHED ATCA-RELATED BUGS FROM OFFICIAL ERRATA SHEETS.

Bug Description	Vendor / Year	Phenomenon
Stale data in processor translation cache may result in hang [10].	Intel 2008	Ignored Invalidation
Possible use of stale translations even after software has performed a TLB flush [7] [8] [9].	AMD 2008 - 2011	Ignored Invalidation
The processor may use stale linear addresses translations [11] [12] [13] [14].	Intel 2015, 2016	Ignored Invalidation
The processor may store an incorrect value into bits of 11:0 of linear address field [13] [14].	Intel 2015, 2016	False Hit or False Miss
Some instructions or task switches may not completely invalidate the processor translation cache [10].	Intel 2008	Ignored Invalidation
Stale data may be loaded into the processor's TLB and used for memory operations [10].	Intel 2008	False Mapping

Table I lists a subset of the most significant hardware bugs that escaped to the market according to official errata documents published from major microprocessor vendors, with a short description of the bug. The last column classifies each bug into a bug scenario described in Subsection B. The last decade has seen specific types of bugs that affect microprocessor functionality, which persist across newer microprocessor generations. Our experimental study demonstrates and explains why these bugs are difficult to detect.

B. Experimental Study

It is essential to study the behavior of the microprocessor during the execution of actual workloads to gain a deeper understanding of rare and difficult bugs in ATCAs, and the conditions that prevent these bugs (shown in Table I) from being discovered during post-silicon validation. For this experimental study, we employed the benchmarks of MiBench suite [21] and the Gem5 simulator [22] in x86-64 mode, which was enhanced with MMU caches [6] [19]. MiBench programs have been used in reliability studies and have significant similarities to SPEC CPU2006 benchmarks in terms of instruction mix and throughput [23]. The Gem5 simulator was configured for our experiments to have 64 Instruction TLB (ITLB) entries, 64 Data TLB (DTLB) entries, 2 Page Map Level 4 (PML4) cache entries, 8 Page Directory Pointer (PDP) cache entries and 32 Page Directory (PD) cache entries (for data and instructions). These sizes are based on an Intel Core i7 according to [19]. Our experiments are based on bug injection in random clock cycles and in random entry fields, to all available entries of each ATCA structure.

In all ATCAs, there are three different fields for each entry: the virtual address field, which is practically the tag; the data field, with physical address and attributes; and the valid bit, which is cleared upon a requested invalidation, usually when a context switch or a page table modification occurs. For each benchmark, we injected 680 different bugs in the virtual address field and 680 different bugs in data field. For the valid bit, we injected 65 bugs on average; this depended on the number of invalidations requested by each execution.

For the valid bit case, we ran experiments with one benchmark assigned to one core, to study the invalidations due to page table modifications for a single process. Furthermore, we ran two benchmarks also assigned to one core, to force multiple context switches from different processes during the whole execution. Each process is given a fixed time quantum from the Linux kernel scheduler, so each benchmark runs concurrently with another benchmark. The scheduling algorithm depends on the kernel. For our experiments we used the Linux kernel 2.6.22 with O(1) scheduler. For all (single and pairs of) benchmarks, we counted the total amount of invalidations (N) that occur during the execution and then executed each benchmark N "buggy" times. Each individual "buggy" execution affects one particular invalidation for one of the ATCAs. The same procedure was repeated for each different ATCA, one at a time.

To trigger as many potential scenarios as possible in ATCAs, we expected each execution for all entry fields to contain as much diversity as possible. For example, we chose the pairs of benchmarks to have similar execution times. This helped ensure that the requested invalidations occurred from context switches between user processes. Moreover, the benchmarks contained both integer and floating point operations, and so on. For each experiment, we recorded either the bug in any field affects the correct program's outcome (*affected*) or not (*masked*). We concluded with the following categorization of bug scenarios shown in Table II.

TABLE II. BUG SCENARIOS THAT CAUSE DIVERGENCE FROM CORRECT BEHAVIOR AND SAMPLE ACTIVATION CRITERION THAT CAN UNVEIL THE BUG.

Bug Scenario	Phenomenon	Activation Criterion
A requested translation by the processor matches a wrong cached entry of an ATCA	False Hit	The VA field gets corrupted before its access
A requested translation by the processor matches a correct cached entry of an ATCA but its data field is incorrect	False Mapping	The data field gets corrupted before its access
A requested translation by the processor matches a stale cached entry of an ATCA	Ignored Invalidation	A requested invalidation fails to invalidate that entry
A requested translation by the processor does not match a correct cached entry of an ATCA	False Miss	The VA field gets corrupted before its access
A requested translation by the processor does not match a correct cached entry of an ATCA	Unintended Invalidation	An invalidation occurs to that entry without being requested

Each bug scenario describes *a divergence from the expected behavior*. The diverging behavior is attributed to a functional bug or an electrical defect and can appear in any ATCA structure. The last column of Table II describes an *example* of an activation criterion for each bug scenario.

C. Findings

Although bugs in ATCAs may exist in silicon and can be unveiled during the execution of validation tests, overall it is difficult to detect specific kinds of bugs. This occurs because they may not affect the correct functionality of the executed programs, and in most cases they do not provide divergences in the states of the silicon. The post-silicon validation process mainly resorts either to multi-pass consistency end-of-test checking methods [3], or to golden responses generated by architectural simulators. Both methods provide a known-correct reference model to check the silicon's outcome for divergences at the end of the test. If a bug takes place in an ATCA during the execution of the validation test and gets masked by a valid entry before its access (masked), the silicon output does not differ from the "bug-free" reference model. As a result, the comparison will succeed, and the bug will remain in the silicon prototype.

Fig. 1 presents the results of our experiments. The percentage shown above each stacked bar refers to the probability that a bug can *affect* the output of the program. The results are categorized into the distinct bug scenarios shown in Table II. These led us to the following observations: *false mappings* have a high probability of influencing the correct program output, thus, they are likely to be detected by traditional validation tests. On the other hand, the most difficult bugs in ATCAs are the *ignored invalidations* and *false hits*. This is due to their low probability of affecting the correct program outcome, as well as the *unintended invalidations* and *false misses*, which can only result in performance degradation and not incorrect output. The latter phenomena have zero probability of affecting the program outcome. For example, when an *unintended invalidation* occurs, the CPU will just miss the translation cache (the correct entry exists, but its valid

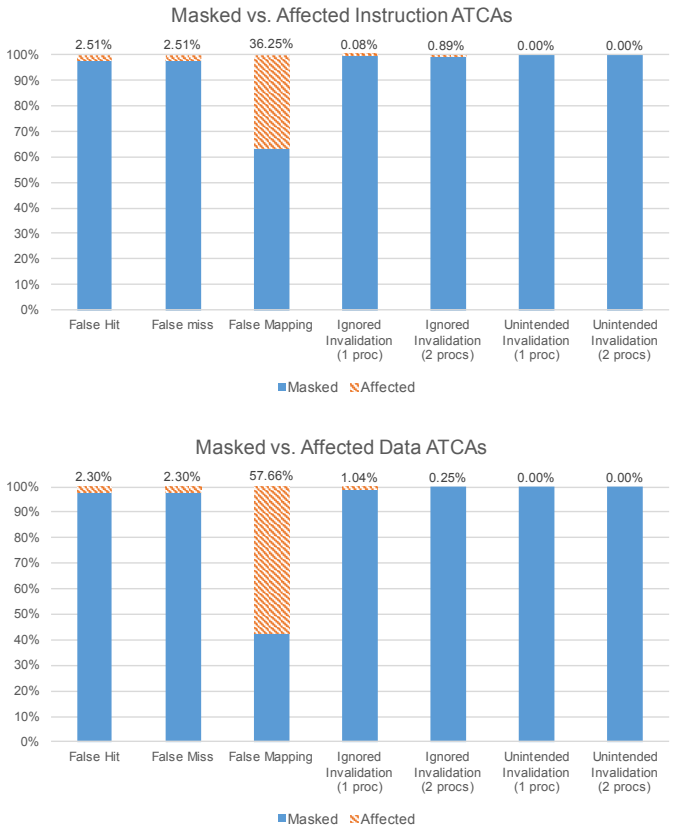


Fig. 1. Masked vs. Affected entries in ATCAs for instruction (upper graph) and data (lower graph) pages show the probability of a bug to affect program outcome.

bit is 0), and will walk the page table in memory to fetch the correct translation.

When a bug affects the tag (the VA field of an ATCA – *false hit* and *false miss*) or the valid bit (*ignored* and *unintended invalidation*) of an entry, it is unlikely to be detected by a traditional validation test. In most cases, the "buggy" entry gets replaced by other valid entries before their next access (masked). The results shown in Fig. 1 demonstrate the most difficult bugs in ATCAs and why major microprocessor vendors publish errata reports with these kinds of bugs.

III. GENERATING VALIDATION TESTS TO UNVEIL DIFFICULT BUGS IN ATCAS

A. The Main Concept

Modern architectures allow multiple processes to execute as if they own the entire virtual address space. More specifically, several processes could write to a memory location at the same virtual address without influencing each other's results. This is done by virtualizing the address space. Assume, for example, processes P1 and P2 both generate a store to memory location at each one's virtual address 0x0123XXX. The processor, aided by the operating system, performs additional adjustments to the address, and maps it to a physical address 0xAABBXXX for P1, and to 0x1122XXX for P2.

The most significant bits of the virtual address (VA) represent the page (called virtual page number - VPN). Those

of the physical address (PA) represent the corresponding frame in memory (called physical frame number – PFN). The last 12 bits of both the VA and the PA (in a 4KB page in this example; there is no limitation for other page sizes) represent the Offset into the page or the frame respectively, which does not participate in the address translation process. Therefore, *different processes can have the same VAs, but, their corresponding physical addresses (PA) may be different*. The exception is global pages, which are shared among several processes. Consider, for example, two running processes on a system, processes P1 and P2, which have the mappings shown in Table III. We assume for simplicity that the addresses are 16-bits long instead of 64-bits in modern microarchitectures. Both processes have the same virtual address space, but different mappings to the physical memory.

TABLE III. EXAMPLE PAGE TABLE ENTRIES (INSTRUCTION AND DATA PAGES/FRAMES) FOR PROCESSES P1 AND P2.

	P1		P2	
Page Type	**VPN**	**PFN**	**VPN**	**PFN**
Instruction Pages/Frames	0x0123	0xAABB	0x0123	0x1122
	0x1234	0xBBCC	0x1234	0x2233
	0x2345	0xCCDD	0x2345	0x3344
	0x3456	0xDDEE	0x3456	0x4455
Data Pages/Frames	0x4567	0xEEFF	0x4567	0x5566
	0x5678	0xFFAA	0x5678	0x6677
	0x6789	0xFFBB	0x6789	0x7788
	0x7890	0xFFCC	0x7890	0x8899

Each validation test consists of two types of processes executed in a row: the *Exerciser* process and the *Checker* process. Exercisers and Checkers are executed in the same core; Exercisers executed in different cores can have shared pages (we take advantage of shared pages; the description shown in Subsection B). However, for the validation of a multicore system, we execute the pair of these processes to all available cores. The Exerciser accesses N different virtual pages, which are cached in ATCAs, and stops execution. The Checker is then invoked to the same core, so a context switch occurs. Upon the context switch, all entries of ATCAs must get invalidated (except for shared entries), according to the instruction set architecture specification. The Checker accesses the same N virtual pages (the accessing order of VPNs does not matter). Given that the Checker accesses the same VAs as the Exerciser, after the context switch, the Checker misses all ATCAs (in a bug-free execution) and must access completely different physical addresses. For example, if an invalidation fails to a DTLB entry (*ignored invalidation*), the Checker will inevitably hit the DTLB, and so it accesses the data of the Exerciser process. If, on the other hand, an invalidation fails to the ITLB, the Checker will execute part of the Exerciser process code.

The same example applies to all other translation caching arrays and entry fields, but with different granularity. The N parameter is the value of maximum entries of the translation cache with the greatest size being validated. For example, if we validate the TLB, then N=64 (in our setup), and if we validate the PD cache, then N=32, and so on (Subsection B). This procedure continues as is but with a different set of virtual-to-physical address pairs for each core to achieve the desired validation coverage for all ATCAs in the microprocessor.

B. *Exercisers*

The key idea behind a comprehensive post-silicon validation is to create a complex combination of validation tests with multiple interleaved execution flows and address patterns that exercise the ATCAs, to expose potential corner case bugs. This is the fundamental role of the Exerciser process. Each of the ATCAs caches both instructions and data in different ways. For effective validation and exercise of all available structures and logic of ATCAs, we propose the following exercising methodology.

To exercise instruction entries of an ATCA (e.g., Instruction TLB), the validation test uses conditional or unconditional branches, jumps, and calls to N different instruction pages. The allocation of instruction entries for ATCAs is achieved by hopping across code page boundaries (Fig. 2). Given a wide range of different address patterns and repetitions that are produced, ATCAs can be successfully stressed [24]. The allocated entries for instruction pages are then used by subsequent loads and stores to N different data pages. These pages are randomly inserted after a jump to a new instruction page, to exercise the data entries of an ATCA (e.g., DTLB).

The Exerciser performs at least one Store operation for each data page; the value stored in physical memory is *its own virtual page number*. Further, the process ID (PID) (or any other number that uniquely characterizes the process) is also stored along with the VPN to the same word, if it is a *global page*. PID information is used to distinguish which process accesses the *shared* memory location. For example, if the VA=0x1234XXX is mapped to the PA=0xAABBXXX, and the current process ID is PID=001, the value stored (datum) in the physical memory is 0x012345**001**. As we described above, each VA contains the offset of the page to the least significant bits (e.g., 12 bits for 4KB pages); this offset does not take place in the ATCAs, so there is no need to validate it. As a result, the last 12 bits of the datum word that is stored in memory can be used to store the PID.

The Exerciser also records the number of translation cache hits and misses that occur during its execution by reading the performance monitoring counters (PMC) of the microarchitecture. When the Exerciser process begins, it declares two global variables to be shared between the two processes. These variables store the translation cache hits and misses before the Exerciser's execution. At the end of the exercising code, it again reads the PMCs to record; these are the total translation cache hits and misses of the Exerciser (N misses, 0 hits). The Exerciser is executed as is once more, so the number of misses in the second execution must be 0, and hits must be N in a bug-free execution.

The role of the Exerciser is twofold: (a) to exercise and stress the ATCAs at all levels (instructions and data) in order to expose as many potential corner cases as possible by accessing a wide range of addresses and address patterns, and (b) to stamp its own physical frames in memory (data and instructions). This is achieved by:

a. Using the same virtual page numbers for instruction and data pages as the Checker process.

b. Storing its virtual page number to the corresponding data frame in memory.

c. Storing a special number in memory that uniquely identifies itself in the last 12 bits of the datum.

d. Recording the total translation cache hits and misses that occur during its execution.

These conditions guarantee that if a bug in ATCAs of any level occurs, the Checker process either executes part of the Exerciser's code, or recognizes that it reads data from the Exerciser. Further, PMCs are used to recognize if a bug affects only performance, for example, *unintended invalidations* or *false misses* (during the second execution).

C. Checkers

The Checker process is used to test whether there is a divergence from the correct (bug-free) behavior, while the Exerciser exercises and stresses the ATCAs. It is essential for the Checker to be realized as a different process (a context switch occurs) in order to validate the valid bit, and as a procedure at the end of the exercising code (without context switch) to validate the tag and data fields. There are also alternative techniques that can cause a change to the state of the valid bit, although they are more complicated. The Checker performs load operations to the last N virtual pages (data and instruction pages), which have been accessed by the Exerciser, and then it checks if the ATCAs operate correctly. Specifically, the Exerciser previously performed N stores to the memory to N different data pages, and the Checker performs N loads from the same VAs. Because the virtual-to-physical address mappings are different between two processes, the fetched datum *must not be equal to the Checker's current VA*. If the fetched datum has the same VA and it is a global page, the Checker also performs a comparison with its PID to the last 12 bits. When the Checker validates the data pages, a mismatch indicates a bug.

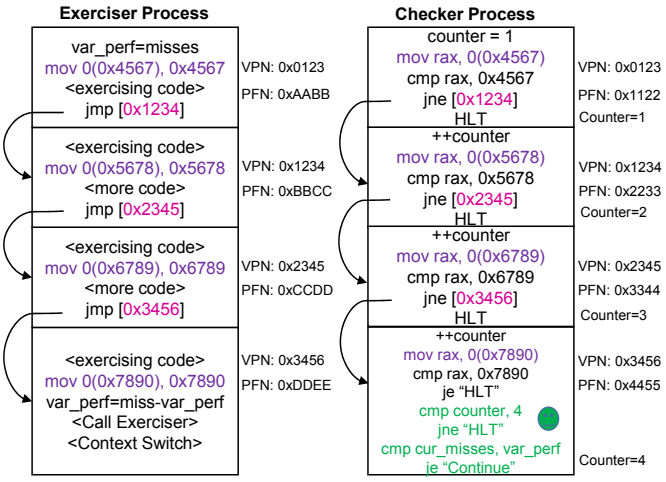
In the validation of instruction pages, consider, for example, that the Exerciser's code (instruction pages) is located in the physical frames (PFN) shown in Table III. Although the Exerciser and Checker processes have the same VAs, their mappings to the PA are different. When the Checker is invoked after the Exerciser's execution and a bug occurs in ATCAs, it is likely that the Checker will execute part of the code of the Exerciser, if, for example, there was a stale entry in the ITLB. To get knowledge of this, the Checker also has a counter increased by one for each jump to the next instruction page. At the end of the Checker's execution, this counter must have a value equal to N. If, for example, the counter equals to N-1, the Checker can recognize that one of its instruction pages has not been executed. Further, the Checker records the total amount of translation cache hits and misses from PMCs, as the Exerciser.

Apart from a context switch, which automatically invalidates the ATCAs (by writing the CR3 register in x86), there are also implicit instructions that invalidate the ATCAs. This is usually when a page table entry modification is done by the kernel. To avoid a context switch that will again invalidate the ATCAs, the Checker is not a process; it is a procedure at the end of the Exerciser and thereby validates the functionality of such instructions. Similarly, the Checker is used as a procedure to validate *false hits* and *false mappings*. When the Checker is a procedure, there is also a change to the condition of comparisons: the *jne* instructions in the code are changed to *je* instructions. Assume, for example, that the Exerciser stops its execution, and the code of the Checker (realized as a procedure now) is executed (without invalidation). Practically, the Checker will load the data from the same VAs to which the Exerciser previously stored its VPNs. If a fetched datum *is not equal* with its VA, a *false hit* or a *false mapping* occurred. The limitation here is that the verification engineers cannot distinguish whether the bug is due to a *false hit* or due to a *false mapping*.

D. Example of the Proposed Validation Flow

In this section, we present some examples of the proposed validation flow (a bug-free flow, an *ignored invalidation* in ITLB, and an *ignored invalidation* in DTLB), which consist of the two processes, the Exerciser and the Checker. Examples for all bug scenarios are omitted because of space constraints. Assume, for example, that we validate the data and instruction TLBs, and there is no bug in both of them, so the validation test should pass. Moreover, we assume for simplicity that the addresses are 16-bits long (plus a 12-bits offset), the size N of the TLBs is 4 entries, and the main memory has been initialized with zero values before the validation process begins. We also use the address mappings of Table III as reference.

As shown in Fig. , the Exerciser process is first invoked for execution. Each block represents an instruction page in memory, with its virtual-to-physical address mappings shown at the right. Each instruction page contains two main instructions: a store operation, and a jump to hop across instruction pages. Data and instruction caching arrays are exercised concurrently. Each store operation stores its own



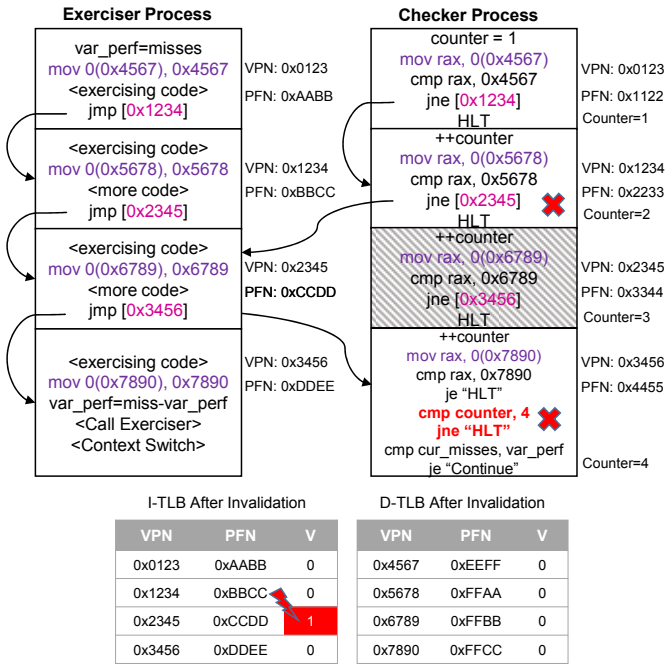
I-TLB After Invalidation

VPN	PFN	V
0x0123	0xAABB	0
0x1234	0xBBCC	0
0x2345	0xCCDD	0
0x3456	0xDDEE	0

D-TLB After Invalidation

VPN	PFN	V
0x4567	0xEEFF	0
0x5678	0xFFAA	1
0x6789	0xFFBB	0
0x7890	0xFFCC	0

Fig. 2. Validation test flow example – Pass; no bug detected.



VPN	PFN	V
0x0123	0xAABB	0
0x1234	0xBBCC	0
0x2345	0xCCDD	1
0x3456	0xDDEE	0

VPN	PFN	V
0x4567	0xEEFF	0
0x5678	0xFFAA	0
0x6789	0xFFBB	0
0x7890	0xFFCC	0

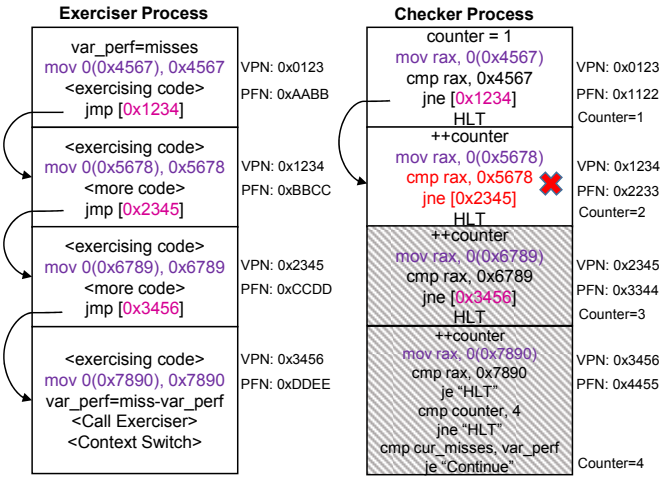
Fig. 3. Validation test flow - Ignored Invalidation in ITLB.

VPN. For example, the value 0x4567 is stored to VA=0x4567XXX. Then, there is a jump to the next instruction page, in which the same procedure takes place. The last instruction page records the total amount of hits and misses of the Exerciser code, and the Checker is invoked. Upon a context switch, the ATCAs must get invalidated.

After the context switch, as we can see in the TLB entries, all entries are invalidated correctly (all valid bits are zero). Although the Checker accesses the same virtual addresses as the Exerciser, it will miss the TLBs, so the Checker passes the test. The first Checker instruction page initializes a counter, performs a load from the VA=0x4567XXX and compares the fetched datum. Given that the DTLB is correctly invalidated, this virtual address will miss the TLB, so the fetched datum that returns will be different from the previous store of the Exerciser (practically it will be zero due to the initialization of physical memory with zero values performed before the validation process begins). The process continues with hops across the same virtual addresses of instruction pages as the Exerciser. On the last page, there are also two more comparisons: with the counter (to detect if a bug occurs in an instruction page) and with the total amount of cache hits and misses (to detect if there is a bug that affects only performance, such as *false miss* and *unintended invalidation*). Given that there is no error in the procedure, the validation test passes.

Consider, for example, that an *ignored invalidation* occurs in ITLB, after the execution of the Exerciser process, as shown in Fig. 3. When the Checker attempts to jump to the instruction page with VA=0x2345XXX, due to the *ignored invalidation* of that entry in ITLB, it will hit the ITLB; hence, it will execute the corresponding code of the Exerciser process and the counter (that exists in the hashed block) will not be increased. The hashed block in the Checker process is not executed, so the comparison of a counter at the end of the Checker does not match. Therefore, the validation test fails due to the *ignored invalidation* of the ITLB.

Assume now that an *ignored invalidation* occurs in DTLB, as shown in Fig. 4. As a result, when the Checker process attempts to load a datum from VA=0x5678XXX, it will go through the cached translation that exists in the DTLB (stale entry), so the datum from PA=0xFFAAXXX will be fetched. As we can see in the physical memory state, the Exerciser process has stored *its virtual addresses as data values into the memory*. In a bug-free execution, when the Checker attempts to read data from VA=0x5678XXX, it must read a zero value (which is the initial value). Instead, due to the *ignored invalidation* in DTLB, it incorrectly reads the value 0x5678. When the Checker process loads the datum stored in VA=0x5678XXX, it will hit the DTLB due to the *ignored invalidation*, and the fetched datum will be *equal to its virtual address*. This means the Checker process reads data from the Exerciser's address space, and the validation test fails. The hashed blocks in Fig. 4 are not executed at all.



VPN	PFN	V
0x0123	0xAABB	0
0x1234	0xBBCC	0
0x2345	0xCCDD	0
0x3456	0xDDEE	0

VPN	PFN	V
0x4567	0xEEFF	0
0x5678	0xFFAA	1
0x6789	0xFFBB	0
0x7890	0xFFCC	0

PFN	Datum
0xEEFF	0x4567
0xFFAA	0x5678
0xFFBB	0x6789
0xFFCC	0x7890
0x5566	0x0
0x6677	0x0
0x7788	0x0
0x8899	0x0

Fig. 4. Validation test flow - Ignored Invalidation in DTLB.

IV. Experiment Evaluation

A. Experiment Setup

To evaluate the effectiveness of the proposed post-silicon validation method, we employ random-generated validation programs, each targeting a different bug scenario shown in Table II. We implemented the test generation using a mix of assembly-level and optimized C programs. To compare the proposed method with a traditional post-silicon validation flow, we developed assembly-level constrained-random

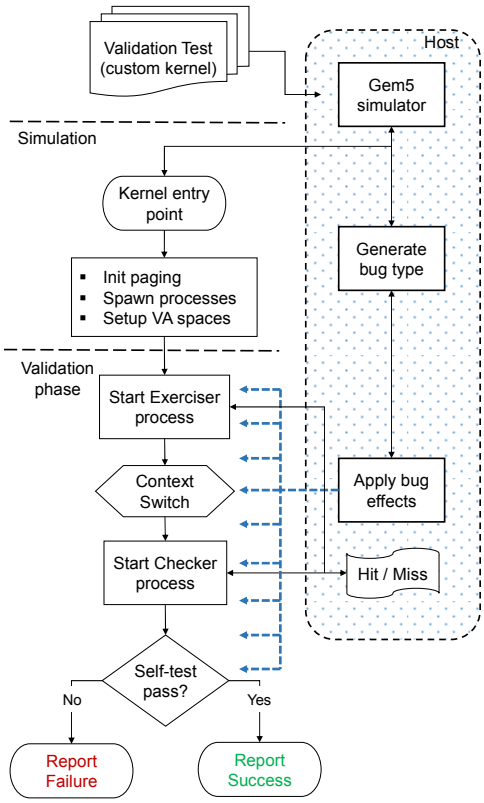


Fig. 6. Proposed post-silicon validation flow and experiment setup.

programs; these have an amount of executed instructions that is comparable to the validation tests of our method.

We configured the Gem5 simulator to have 64 ITLB entries, 64 DTLB entries, 2 PML4 cache entries, 8 PDP cache entries, and 32 PD cache entries (for data and instructions); however different sizes do not provide limitations for the proposed method. We modified the Gem5 simulator to support the injection of *random, non-deterministic* bugs in the address translation caching arrays of the x86-64 model, by covering as many erroneous conditions that may occur in prototype chips as possible. We also modeled known ATCA-related errata described in Table I. With these two approaches, our simulator resembled a realistic "buggy" microprocessor prototype chip. We also developed a *custom minimal kernel* to create a realistic bare-metal post-silicon validation "environment" and execute our validation programs.

We contribute to the most critical part of functional post-silicon validation, the one based on bare-metal-infrastructure. During functional post-silicon validation, only random-generated test programs and legacy tests are applied, not normal applications. Bare-metal modeling on Gem5 was a major part of our development work, enabling it to resemble the real hardware infrastructure. Our kernel initiates all the procedures required to set up the paging interface needed to operate in Long Mode (x86-64). It also offers a fundamental infrastructure for the virtual memory mappings of two different processes. *Such a bare-metal validation flow modeling with interprocess communication (IPC) on microarchitecture simulators to evaluate post-silicon validation methods has not been presented in the literature before.*

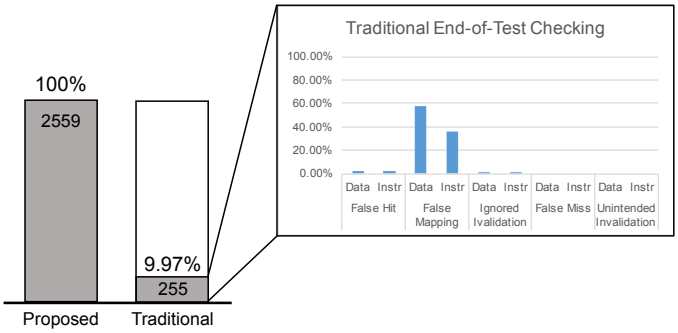


Fig. 6. Bug coverage for the proposed method vs. traditional end-of-test checking techniques.

Fig. 5 presents the proposed post-silicon validation flow as it is modeled in the Gem5 simulator. This procedure can also be integrated in a real prototype chip without modifications in the kernel or validation tests. The kernel is responsible for initiating and managing the processes and their address spaces needed for the validation tests. The host (simulator) generates bugs (by alternate random bits in each field) according to each phenomenon described in Table II, and applies the bug into a random entry of ATCAs. Furthermore, it records the total amount of translation cache hits and misses. In a real microprocessor chip this is done by reading the performance monitoring counters. At the end of execution, the validation test reports whether the test succeeded or failed.

B. Results

To evaluate the effectiveness of the proposed method, we performed massive injections of random bugs. Each validation test consisted of 200K committed instructions. Table IV presents the total number of bugs injected in ATCAs applied to each different entry, as well as the ATCA-related bugs presented in official errata sheets. These bugs were applied to all available entries of ATCAs for four different patterns of virtual address spaces.

TABLE IV. NUMBER OF DIFFERENT TYPE OF BUGS FOR EACH PHENOMENON.

Phenomenon	Number of Bugs	
	Random	Errata
False Hit / False Miss	1360	2
False Mapping	680	1
Ignored Invalidation	256	4
Unintended Invalidation	256	–
Total	**2552**	**7**

Fig. 6 summarizes the results of our bug injection experiments. Our proposed methodology detected all 2559 bugs injected into the Gem5 simulator (bug coverage 100%). Conversely, the traditional end-of-test checking techniques detected only 255 injected bugs (bug coverage 9.97%). This difference is due to the limitations of these techniques in the validation flow, given that they check if the output of the DUV is equivalent to a golden reference output. In most of the cases, although a bug manifests itself, it does not affect the output. For example, as we presented in Section II, *false mappings* are easier for a traditional validation test to detect, in contrast to other bug scenarios. Consequently, *false mappings* have significantly higher detection rates.

V. Related Work

Recent studies present different microprocessor post-silicon validation methods, which employ the inherent ISA diversity to generate self-checking post-silicon validation programs. The original instructions of a validation program are complemented with equivalent instructions [25], instruction sequences [26], reverse instructions, or diverse data operands [27]. Although these studies present comprehensive methods that validate the functionalities of the core, they cannot detect all bugs in the address translation subsystem (for example, bugs that only result in performance degradation).

Although the work presented by Papadimitriou et al. [28] proposes a method to detect bugs in ATMs, it has other limitations: (1) instructions are not validated at all, (2) store operations are not permitted because they jeopardize the validation process, (3) bugs that result only in performance degradation are not detected, and (4) bugs that can be masked by other valid entries are also undetectable. These are severe problems that can occur due to the intricate functionality of caching arrays, therefore requiring advanced validation tests. Given the importance of various self-checking methods and their limitations in detecting address translation bugs, our work in this paper contributes self-checking validation programs for the bugs that can occur in the ATCAs of microprocessors. The approach presented by Romanescu et al. [29] targets the runtime verification of the correctness of address translation for memory consistency, whereas our approach targets the post-silicon validation phase, which is performed before the chip is released to the market.

VI. Conclusion

We presented a post-silicon validation methodology for difficult bugs in the address translation caching arrays (ATCA) of modern microprocessors. We characterized the problem by experimentally demonstrating the most difficult bugs in ATCAs and showing why traditional validation techniques cannot detect them. We also presented a post-silicon validation program generation flow that unveils and detects difficult bug scenarios in ATCAs. Our proposed methodology is self-checking, so that it is fully de-coupled from time-consuming simulations or other end-of-test checking methods that produce golden models. Our experiment evaluation in an enhanced Gem5 x86-64 model with MMU caches demonstrates the effectiveness of the proposed methodology to detect severe kinds of bugs in contrast to traditional validation tests.

Acknowledgments

The authors would like to thank Chani Sacharen and Dalit Shmueli from IBM Research - Haifa for reviewing and proofreading the article.

References

[1] A.Nahir, M.Dusanapudi, S.Kapoor, K.Reick, W.Roesner, K.-D.Schubert, K.Sharp, G.Wetli, "Post-Silicon Validation of the IBM POWER8 Processor", DAC 2014.

[2] A.Adir, R.Emek, Y.Katz, A.Koyfman, "DeepTrans – A Model-Based Approach to Functional Verification of Address Translation Mechanisms", MTV 2003.

[3] Y.A.Katz, A.Koyfman, E.Tsanko, "Method of Verification of Address Translation Mechanisms", US Patent No. 8245164.

[4] A.Adir, L.Fournier, Y.Katz, A.Koyfman, "DeepTrans - Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms", HLDVT Workshop 2006.

[5] A.Koyfman, A.Adir, R.Emek, Y.Katz, M.Vinov, "Modeling Language and Method for Address Translation Design Mechanisms in Test Generation", US Patent No. 7370296.

[6] T.W.Barr, A.L.Cox, S.Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)", ISCA 2010.

[7] AMD Revision Guide for AMD Family 11h Processors, Rev. 3.00, 41788, July 2008.

[8] AMD Revision Guide for AMD Athlon™ 64 and AMD Opteron™ Processors, 25759, Rev. 3.79, July 2009.

[9] AMD Revision Guide for AMD Family 11h Processors, Rev. 3.10, 41788, December 2011.

[10] Intel® Pentium® 4 Processor Specification Update August 2008 Revision 071 Document Number: 249199-071.

[11] 5th Generation Intel® Core™ Processor Family, Intel® Core™ M Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family, Specification Update, October 2015, Revision 015, Reference Number 330836-015.

[12] Mobile 4th Generation Intel® Core™ Processor Family, Mobile Intel® Pentium® Processor Family, and Mobile Intel® Celeron® Processor Family, Specification Update, November 2015, Revision 028, Reference Number 328903-028.

[13] Intel® Xeon® Processor E3-1200 v3 Product Family Specification Update January 2016, Reference Number 328908-014US.

[14] Intel® Xeon® Processor E7-8800/4800 v3 Product Family, Specification Update, March 2016, Reference Number 332317-007US.

[15] Intel® Core™ Duo Processor and Intel® CoreTM Solo Processor on 65 nm Process Specification Update June 2009 Revision 020.

[16] Intel® Xeon® Processor 5100 Series Specification Update December 2012 Order Number: 313356-02.

[17] A.Adir, M.Golubev, S.Landa, A.Nahir, G.Shurek, V.Sokhin, A.Ziv, "Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors", DAC 2011.

[18] S.Ray, J.Yang, A.Basak, S.Bhunia, "Correctness and Security at Odds: Post-silicon Validation of Modern SoC Designs", DAC 2015.

[19] A.Bhattacharjee, "Large-Reach Memory Management Unit Caches", MICRO 2013.

[20] N.Foutris, D.Gizopoulos, J.Kalamatianos, V.Sridharan, "Assessing the impact of hard faults in performance components of modern microprocessors", ICCD 2013.

[21] M.R.Guthaus, J.S.Ringenberg, D.Ernst, T.M.Austin, T.Mudge, R.B.Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", IISWC 2001.

[22] N.Binkert, B.Beckmann, G.Black, S.K.Reinhardt, A.Saidi, A.Basu, J.Hestness, D.R.Hower, T.Krishna, S.Sardashti, R.Sen, K.Sewell, M.Shoaib, N.Vaish, M.D.Hill, and D.A.Wood, "The gem5 simulator", SIGARCH Comp. Arch. News, 2011.

[23] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators", IISWC 2015.

[24] H.Rotithor, "Post-Silicon Validation Methodology for Microprocessors", IEEE Design & Test of Computers 2002.

[25] N.Foutris, D.Gizopoulos, M.Psarakis, X.Vera and A.Gonzalez, "Accelerating Microprocessor Silicon Validation by Exposing ISA Diversity", MICRO 2011.

[26] T.Hong, Y.Li, SB.Park, D.Mui, D.Lin, Z.A.Kaleq, N.Hakim, H.Naeimi, D.S.Gardner and S.Mitra, "QED: Quick Error Detection Tests for Effective Post-silicon Validation", ITC 2010.

[27] I.Wagner, V.Bertacco, "Reversi: Post¬silicon Validation System for Modern Microprocessors", ICCD 2008.

[28] G.Papadimitriou, A.Chatzidimitriou, D.Gizopoulos, R.Morad, "ISA-Independent Post-Silicon Validation for the Address Translation Mechanisms of Modern Microprocessors", IOLTS 2016.

[29] B.F.Romanescu, A.R.Lebeck, D.J.Sorin, "Specifying and Dynamically Verifying Address Translation-Aware Memory Consistency", ASPLOS 2010.