Validation methodology for Nest Memory Management Unit

Nandhini Rajaiah
IBM
Bangalore,India
nrajaiah@in.ibm.com

Jayakumar N Sankarannair
IBM
Bangalore,India
sjayakum@in.ibm.com

Larry S Leitner
IBM
Austin,United States
lleitner@us.ibm.com

***Abstract*— The growing demand for performance makes the processor logic design more complex, thereby making post-silicon validation a critical and complex step in processor development life cycle. There are complex units with newer timing and control logic paths which are almost impossible to exercise in regular verification environments. One such unit to cater to newer workloads in recent superscalar processors is the Nest Memory Management Unit (NMMU), a memory management unit for all I/O devices. This paper presents some of the major challenges in validating Nest MMU. A post-silicon validation framework is proposed to mitigate these challenges. An asynchronous non-blocking accelerator job submission model is used in this approach to increase the translation traffic from the agent to NMMU. Core MMU translation is used as the reference model to validate nest MMU. The processor core storage exception handlers are leveraged to minimize the validation tool software development effort and to increase the efficiency of validation as well. This method makes use of an optimized threshold-based checker to detect potential NMMU hardware issues. The proposed methodology has been experimentally evaluated in Power9 NMMU to demonstrate the effectiveness of the method in providing considerable stress to the unit.**

Keywords—Post-silicon validation*, *address translation mechanisms, microprocessor, accelerator, design verification

I. INTRODUCTION

Hardware accelerators allow the machine to offload work from the host CPU to the Accelerator. The Accelerator then completes the computation and returns the results back to the host CPU which relieves the host of precious computation cycles. Accelerator chips are becoming an important part of most processor designs where performance is considered essential. Due to the increase in the volume of data and inputs from the system, performance improvement has become critical. This renewed perception of performance has made accelerators an integral part of the system [1].

A unified virtual address space between the host CPU cores and accelerators can largely improve the latency and bandwidth of FPGAs and ASICs [2]. It also allows accelerators to behave as if they are integrated into custom microprocessors, which in-turn necessitates hardware support for address translation. Supporting address translation for customized accelerators is becoming a complex task. In recent processors with state-of-the-art I/O subsystem technology, this has been achieved using Nest Memory Management Unit.

The Nest Memory Management Unit (NMMU), as shown in Fig. 1, is a complex integrated circuitry that resides within each processor chip and provides address translation support for multiple accelerator agents, including the on-chip nest accelerator (NX), off-chip Nvidia Processing Unit (NPU) and Coherent Accelerator Processor Proxy (CAPP0/1) units. Nest MMU primarily communicates with external units through the system bus (i.e., Fabric). The NMMU also interacts with memory to perform table-walks and to update the translation tables, as needed. In addition, cache management instructions (Translation cache invalidates) are sourced by the core/NCU (Non-Cacheable Unit) of a given processor in the system and are snooped and managed by the NMMU on behalf of the attached accelerator units. The primary goal of NMMU is to provide effective address (EA) to physical address (PA) translation for the various Accelerator Agents within the processor's storage subsystem without going through the main processor core. This improves the response time of accelerator agents working in virtual address space. In addition, the NMMU protects the pages that are being translated by ensuring that only tasks with the proper authorization can access them [4].

This paper is organized as follows: Section II describes the related works in this area and discusses the major challenges involved in validating NMMU. Section III describes our methodology used to validate NMMU. The experimentation setup and results are summarized in Section IV. Finally, the last section concludes the paper with a summary of the work as well as future directions.

II. BACKGROUND

The Memory management unit (MMU) of a processor translates the effective address (EA)/virtual address (VA) to physical address (PA). The MMU is one of the complex units of a modern microprocessor and probably most ambiguous and difficult unit to validate due to various caching arrays such as Translation Look-aside Buffers (TLB) and Page Walk Caches (PWC) [4]. The result of translation, the physical address is not directly observable to a program, hence failure is detected late in the test and make it hard to debug the failure.

Recent studies have presented a complete view of bug models for the address translation mechanism (ATM) and methods to detect ATM bugs using self-checking mechanisms [3]. This work presents a comprehensive experimental study on a state-of-the-art microarchitecture to assess and identify the bugs in address translation caching arrays and explains why these bugs persist across generations. The methods used in this approach primarily targets the verification and validation of address translation subsystem of the processor core. Our paper addresses the challenges involved in validating address translation sub-system of accelerators and proposes a solution for overcoming the challenges associated with the validation of nest address translation unit.

A. Challenges

The Nest MMU is a complex unit which requires rigorous validation effort to make sure that the design is bug-free. Apart from design complexity and short schedule, NMMU validation poses many other challenges. One such major challenge involved in NMMU validation, unlike core MMU validation, is the need to have special accelerator agents to generate high-rate traffic conditions sufficient enough to test the unit. The second major challenge is that, unlike in core address translation, where there are multiple processors creating simultaneous traffic to stress the unit, in accelerator environment, there are a few agents, through which translation requests can be triggered to the NMMU. Unlike in core address translation validation, where a single byte-level operation such as load/store can trigger a translation request to MMU, in nest, predominantly accelerator operates on blocks of data, although, explicit work-loads can be created with byte granularity for validation. And moreover, the role of the core in job submission is to post the control block into the accelerator queue. Then the accelerator will pick up the job when it is free. There may be a delay between the time when a job request is submitted to the accelerator and the actual completion of the job. So we need a mechanism to submit jobs asynchronously to the accelerator, in order to increase the potential throughput of checkout, i.e., translation requests to NMMU. Translation faults from agents are presented to the processor as external interrupts by a virtualized external interrupt hardware unit, and moreover, applications running on a processor can be interrupted asynchronously with address translation faults from the accelerators. In this case, it is the responsibility of the interrupt service routine running on the processor to communicate the fault information to the corresponding application running anywhere in the system through the standard inter-process communication (IPC) mechanism. The interrupt hardware unit was configured in such a way that the interrupts were fairly distributed among different applications running on the system. This approach enhanced the efficiency of the tool by not swarming a specific processor with interrupts.

III. METHODOLOGY

A new method has been demonstrated to overcome the major challenges associated with validation of NMMU. This method uses Nest Accelerator Unit (NX) as the agent to induce translation traffic to NMMU. Nest Accelerator Unit comprises one cryptographic and two memory compression/decompression engines (coprocessors). The Nest MMU and core MMU share the same translation table that maps effective addresses to physical memory. The main objective of sharing the translation table between CPU and accelerator is to use core MMU translation as the reference to validate nest MMU and also to reuse existing core interrupt service routines to set up nest translations. An asynchronous job submission model has been used for submitting jobs to the accelerator, where each processor builds its own job-table with ‘n’ number of jobs and submits each of the jobs to the targeted coprocessor and continues with the next job without waiting for the previously submitted job to complete. The intention of using this model is to create a swarm of translations requests without waiting for each request to complete.

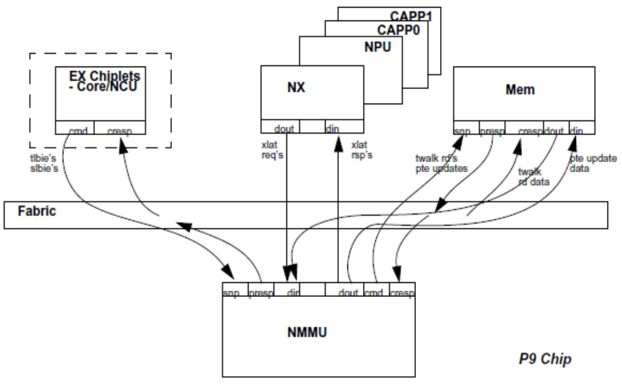

Fig.1. Nest Memory Management Unit

The test environment includes multiple processors configured to submit jobs to the accelerator in a bare-metal configuration i.e., validation program is executed directly on a system without any operating system. As part of the processing of a job, the accelerator may generate a translation request to NMMU. If a translation does not exist for the address being processed NMMU performs a look-up of its cache to see if a translation entry exists for the requested EA. If so, it returns the physical address. If not, it performs a table-walk to obtain the targeted PA. If the table-walk also fails, an interrupt is generated to notify the processor with the fault information, including the fault status, faulted address. The translation faults are handled, and translations are installed as described in Section III B. After handling the faults, the processor now resubmits the job to the accelerator and starts polling the coprocessor status block. The expectation is that NMMU should not generate the same fault again since the translations are now available. The main function of the Virtual Accelerator Switchboard (VAS) unit is to allow user-level software code running on a processor core to directly access the Nest Accelerator engines. Fig. 2 outlines the validation test setup and control flow between the units involved in the nest address translation process.

A. Testcase Generation

The job to be performed by an accelerator is defined by an agent-specific control block in memory. The control block contains all the control information and pointers needed to allow the accelerator to access the input parameters, input data and to know where to store the output data and finishing status. It also has a pointer to address translation context. Translation context has control fields including privilege levels, translation mode, partition-id and process-id, to control the address translation. The control block used by NX is called the Coprocessor Request Block (CRB). The

CRB contains all the information necessary for NX to perform the coprocessor functions.

The test case generator builds the coprocessor request block and the translation context in the memory. It selects a random coprocessor operation. Each processor pre-selects a random translation context with pseudo-random values for privilege levels, modes and the process-id, to increase the test coverage.

Test case generator generates random addresses for the agent's source and target area. It then builds coprocessor request block and other control structures with fields specific to the randomly selected coprocessor operation. The context information for each job is stored in a data structure in main memory. Thereafter, the job is dispatched to the accelerator. Each of the processors can continue with other tests. The status of submitted jobs is determined from the coprocessor status block or from the processor job queue. The coprocessor status block in memory is updated by the accelerator upon job completion. the per-processor queue is updated by the interrupt handling process, with fault information, upon fault interrupt.

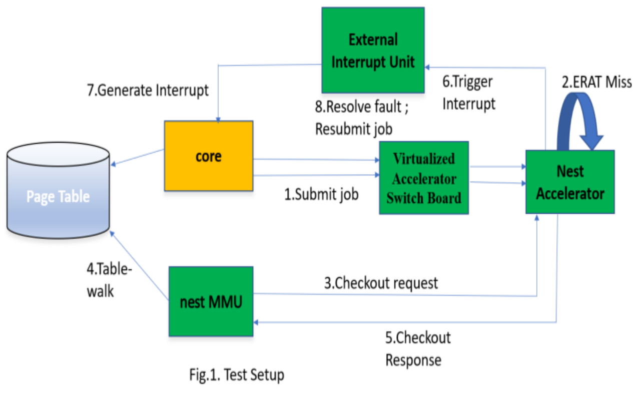

Fig.2. Validation Test Setup

One limitation with the proposed methodology is that, over the course of test execution, translations will be installed for most of the address pages and the number of faults generated by the unit will be drastically reduced. To alleviate this problem, two strategies were adopted. One, a special irritator mode was used, where a randomly selected processor invalidates translation addresses used by other processors in the system, by changing their corresponding translation entry valid bits to invalid, which in turn, triggers storage interrupts. Another, after a predefined number of test-cases, the accelerator job table is rebuilt with new parameters for translation context and agent source/target addresses for each job. This helps to maintain the fault requests from the agent to an optimum level, sufficient enough to stress the unit.

Nest accelerator has a provision to specify source and target memory locations as a list of Data Descriptor Entries (DDE). Our method leverages this hardware facility to increase the number of storage interrupts generated by the agent. Each DDE of a job points to a different effective address page. This provokes the agent to generate a translation fault for each DDE processed by it.

B. *Translation Generation*

The accelerator can interrupt any processor on the system when it fails to find the translation for an address. The translation entry to be installed for the address depends on the translation context of the processor which submitted the faulted job. The submitted processor number is determined from the fault data structure in memory which is updated by the accelerator on a page fault. The interrupted processor sends the faulted address and other fault information to the submitted processor through an inter-process communication mechanism for further processing. The submitted processor, on receipt of fault information from the interrupted processor, attempts to access the data from the faulted address. This generates a storage interrupt to the core and the core storage interrupt handler sets up the required translation entry for the address. The core and the nest environments use the same translation table and hence the new translation entry installed by the core is visible to the nest. The validated core MMU and core storage interrupt handlers are used in the validation of nest MMU. This approach eliminated the need for developing explicit interrupt service routines for NMMU fault handling and in-turn reduced the software development effort for NMMU validation.

C. *Checker Logic*

An end-of-test checking method is used to check the correctness of target data. The expectation here is that, with a correctly functioning memory management logic, the agent should find the correct physical address for a given virtual source or target address and should read the right source location and write the output data to the correct location. Each job has two operations and the output of the first operation is used as an input to the second operation. The second operation is the reverse of the first one. At the end of each job, data-checker will compare the expected data (original source) with the actual data (output of the second operation). Any incorrect translation is manifested as a data mismatch and the mismatch is detected by the data checker logic.

To detect the case where NMMU is not able to resolve the page fault, a checker code is used which increments a counter value upon fault from NMMU. When the checker has reached a maximum threshold value, it stops the test and reports fail. The challenge with this approach is to identify the right threshold value to halt the test execution. If the threshold value is too high, there is a possibility that we may miss a potential bug. For example, a specific processor invalidates a mapping in the MMU cache, as part of setting up its entry, but the invalidation signal is missed. In this case, the test continues to use the old translation, which in turn generates repeated faults. As part of processing these faults, the subsequent invalidation may eventually go through and the bug goes undetected. In this case, a high threshold value is undesirable. On the other hand, if the threshold value is too low, there is a possibility of false alarm, indicating a test failure. For example, consider that a specific processor has installed a translation entry in the translation table. Now if some other processor replaces this entry, then the first processor will fail although it had installed the entry previously. Here, it would be appropriate to give another chance for the processor to re-install the entry, instead of abruptly stopping.

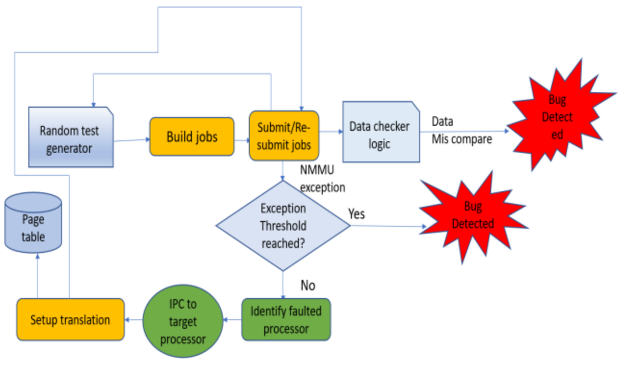

Fig.3. Proposed validation methodology flow

An optimal threshold value is determined by empirical means, and this value varies depending on the work-load and core configuration on which the test is running. Fig. 3 outlines the proposed validation methodology flow.

IV. EXPERIMENTATION AND RESULTS

The proposed methodology was applied to validate Nest Memory Management Unit (NMMU) of Power9 processor. The test was run on both pre-silicon verification and post-silicon validation environment.

Our tool was implemented using a mix of assembly-level and C programs. The kernel has procedures required to enable the NX accelerator engines and setup configuration registers. The test generator builds 16 NX jobs, each comprising of two Coprocessor Request Blocks. VAS has memory-mapped areas called send and receive window contexts to establish a communication channel between user process and the accelerator. As part of the system initialization, the kernel configures VAS receive-window contexts for each of the accelerator types that will be accessed directly via user-level processes. The VAS receive-window context points to the requested accelerator’s FIFO (First In First Out) data structure in system memory. When the processor wants to access an NX accelerator, it sets up a VAS send-window context. The number of requests that can be simultaneously submitted is controlled through a hardware credit-based system. The test now configures a CRB and uses the copy-paste facility [4] to copy the contents of the request block to the accelerator’s receive FIFO. The NX-unit accelerator receives notification of the request and pulls it from the FIFO to be processed. When the operation is complete, the processor is notified via an interrupt, or it detects completion via polling, as configured by the processor.

A. *System-level simulator*

Developing validation software for complex hardware units in the absence of the target hardware is often error-prone. Many times, the test will fail due to software infrastructure issues such as improper memory configurations, illegal memory writes or software race conditions. However, delaying software debug until the hardware is available results in finding software defects too late, thus increasing the time to market for the product. To alleviate these problems, a system-level behavioral simulator is used to detect software defects earlier in the development cycle, effectively in parallel with hardware development [6].

The system-level simulator is a high-performance, functional behavioral model of architecture mirroring the hardware functionality that is visible to software. The lower level represents the hardware and operating system choices that can be used to execute the simulation environment. Various user-level programs can be loaded on top of it. This provided a virtual environment to validate design assumptions, verify the developed code and ensure the reliability of validation software.

B. *Verification test bench*

The verification environment also forms an important component of chip bring-up when the hardware arrives. Test cases are easily moved back and forth between hardware and simulation environments to perform root-cause analysis of any unexpected hardware behavior. This allows us to find workarounds and can also be used to find an occasional subtle software bug.

During pre-silicon verification, the test was run on an internal ASIC-based simulation acceleration platform called AWAN, also known as, Exercisers on Accelerators (EoA), [5] to provide two benefits. First, to provide additional functional coverage to pre-silicon testing. Secondly, it helps us to use the pre-silicon coverage data to further enhance the test-cases. In addition, tool development and testing are done in the simulation environment before the actual hardware is running in the laboratory.

AWAN uses a massive network of Boolean function processors each loaded with multiple logic instructions. Typically, each run through the sequence of all instructions in all logic processors in parallel constituted one machine cycle, this implementing the cycle-based simulation paradigm.

Formal verification re-uses RTL (Register Transfer Level) models abstracting blocks (i.e., units) with behaviorals [5]. Models exceeding 31 million gates have been simulated in AWAN. These are essentially multi-unit models with heavy black-boxing. Simulation speed depends on the configuration, model size, model complexity, and the amount of host interaction. The chip level model used in our experiment had POWER9 chip with four cores together with the L2 and L3 cache complex, the on-chip fabric, memory controllers populated with behavioral DIMMs and the nest complex constituting of VAS (Virtual Accelerator Switchboard), NX, NMMU and XIVE (External Interrupt Virtualization Engine) units.

Internally, BugSpray, [7] an extension of VHDL is used for functional coverage and assertion instrumentation. This tool is used to efficiently annotate the RTL with assertion and coverage events. BugSpray enables verification objects to be portable across verification disciplines and across hierarchies and allows for their reuse with design. The coverage events provided by the design team helps in assessing the efficiency of the validation tool used to stress the device under test. Test coverage statistics were collected using this coverage checker tool and analyzed to detect low coverage areas. The test was enhanced to hit all the coverage events to exercise the corner cases. Fig.4 shows the pre-silicon verification setup.

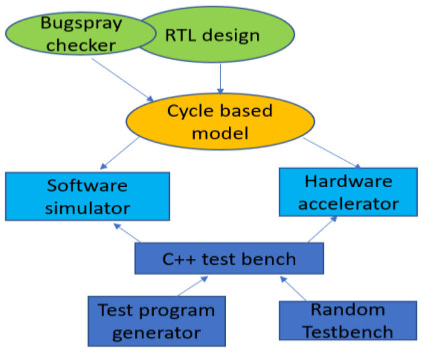

Fig 4. Pre-silicon verification setup

C. *Post-silicon validation*

The test was run on post-silicon environment, to achieve the following objectives. First, to validate the correct behavior of NMMU in various translation modes. Second, to ensure that the unit is stressed well by collecting coverage statistics and analyzing them. In addition, the proposed method runs the way application is supposed to use accelerator which verifies the interface between NMMU and the associated hardware units such as NX, processor core, External Interrupt unit. The post-silicon validation procedure followed is described in Fig. 5.

System reset and initialization are done through boot procedures.
Kernel and testcase generator are loaded into system memory.
All cores are triggered to start execution.
Scheduling of test-case is done on different cores by the kernel.
Scratch register is written which is polled for pass or fail criteria.

Fig 5. Post Silicon Validation procedure

Experiments were carried out on a two-chip processor with 12 cores per chip in SMT4 mode (4 threads per core) using NX agent. Nest Accelerator consists of a cryptographic engine which performs Advanced encryption standard (AES) and Secure Hash Algorithm (SHA) operations, 842 Compression/decompression engine and GZIP compression/decompression engine, on each chip. All the units are in the nest clock domain and run at a frequency of 2MHz.

The test used two methods to collect details about the traffic stimulus to the unit. The tool has a mechanism to record various events that happened during test case execution. We used this data to collect information such as the number of times a certain event is hit by the test during the test execution, that gave us a measure of how well we were exercising all the possible design paths of the unit. The second method used the Hardware Performance Monitoring Unit to record various events in the system.

IBM processors have a special hardware facility, the Performance Monitor Unit (PMU) [4], to collect the events related to the operations in the processor. Each processor core has six 32-bit PMC (Performance Monitor Counter) registers, PMC1 through PMC6. These registers are programmable, so you can specify what events to collect by setting the Monitor Mode Control Register (MMCR). PMC registers count the events when a processor executes instructions. The post-silicon validation flow is shown in Fig. 6.

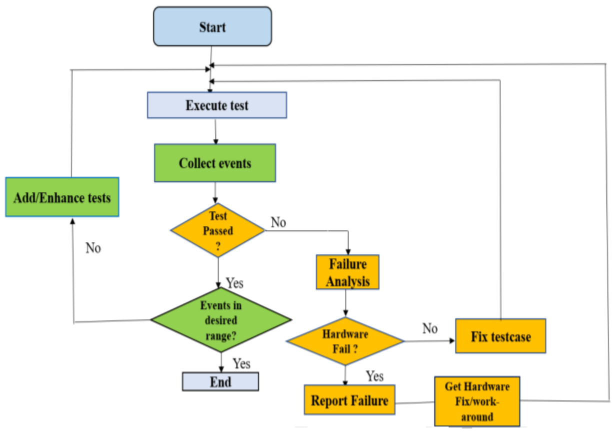

Fig 6. Post Silicon Validation Flow

D. *Results*

The tool generated event data and PMU data have been used to provide feedback on the effectiveness of our tests, to tell about the amount of stress created to the unit. This helped us to enhance our test-cases to improve the coverage. For instance, during the initial post-silicon test runs, the total number of translation requests and the number of checkout requests generated by the agent were low compared to the maximum theoretical limit. The event data analysis showed that the agents were reusing the translations. The Translation Table irritator was enhanced, and it increased the translation request and checkout request counts to the desired level.

To put this in perspective, Table I and II show samples of event data collected primarily with Validation Test Suite 1 (VTS1) and Validation Test Suite 2 (VTS2) on a post-silicon validation environment. VTS1 is the bare metal test designed using this methodology to generate more page/segment faults. VTS2 is the test created by randomizing the authorization privileges of each page, to generate more protection faults.

Table I
Statistics collected by coverage tool with VTS1

Coverage Parameters	**Values**
Total translations	5440
Clean checkouts	3452
Total number of cycles	469341426
Cycles per translation	86275.99743
Translations per second	23181.418

Table II
Statistics collected by coverage tool with VTS2

Coverage Parameters	**Values**
Total translations	4481
Clean checkouts	3812
Total number of cycles	237279016
Cycles per translation	52952.24
Translations per second	37769.88

Fig. 7 shows the average number of checkout requests generated by the agent measured by running validation suite 1 and 2 on the processor.

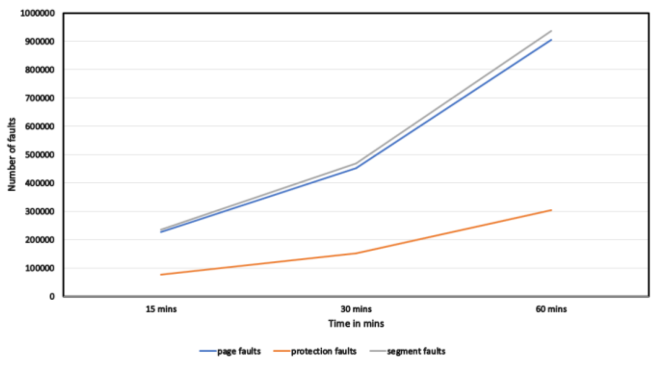

Fig.7. Number of checkout requests against time

The proposed approach proved invaluable for the validation program in the following 3 areas:

1. Logic verification and validation: Some of the functional bugs were uncovered, fixed, and re-checked at the early verification stage. Tests were executed, accumulating tens of billions of simulation cycles and ultimately ensuring a high-quality tape out of the ASIC, reducing overall program risk. After hardware arrived, the same test was leveraged without any modification, to validate the unit.

2. Coverage and throughput enhancement: At pre-silicon level, coverage metrics were used to quantify which design functions have been reached by simulation. Test coverage statistics were collected in the post-silicon phase and tests were tuned to improve the coverage.

3. Foster rapid software development: The kernel software for core interrupt handlers were reused in this approach. This avoided duplicate development work for interrupt handlers. We achieved the goal of reducing the software development time, thereby shrinking the overall validation tool development time.

V. CONCLUSION

Validation is one of the most complex and critical tasks in the current processor design process. Recent trends in computer systems have evolved into many accelerators and units to support new accelerator functionalities. Especially for units which are external to the core, innovative validation techniques are needed to achieve throughput and efficiency. In summary, this paper discusses the major challenges in validating accelerator address translation sub-system. A validation methodology has been presented for nest MMU, that uses an asynchronous accelerator job submission model with an optimized threshold checker. The framework is designed to use core MMU and core storage interrupt handlers for installing translation for the nest environment. This reduced the validation software development effort by a considerable amount. The proposed methodology has been successfully applied in stressing the nest MMU unit of the processor and the results are presented for the same. The test could reach up to 80% of the maximum throughput supported by NMMU.

The NMMU interface supports up to eight outstanding translation requests from each of its agents. With all the four agents, there can be up to 48 outstanding requests to NMMU. To keep all the NMMU channels busy, it is good to run multi-agent test-cases, where all the agents together can swarm NMMU with translation traffic. In the future, the innovative methods described in this paper can be extended to other agents/units to generate multi-agent traffic to NMMU.

ACKNOWLEDGMENT

The authors of this paper would like to thank Manoj Dusanapudi, IBM India Pvt. Ltd., for his valuable feedback and support in completing this paper.

REFERENCES

[1] Semiconductor Engineering - The Secret Life of Accelerators,2017. https://semiengineering.com/the-secret-life-of-accelerators/

[2] Accelerate the Future of Computing with Power Acceleration,2018,https://www.linkedin.com/pulse/accelerate-future-computing-power-acceleration-manoj-dusanapudi/?published=t

[3] George Papadimitriou; Athanasios Chatzidimitriou; Dimitris Gizopoulos; Ronny Morad,"An Agile Post-Silicon Validation Methodology for the Address Translation Mechanisms of Modern Microprocessors", *IEEE Transactions on Device and Materials Reliability*,2016.

[4] Power9 Processor's User Manual, 2018, https://openpowerfoundation.org/?resource_lib=power9-processor-users-manual

[5] J. Darringer et al. ,"EDA in IBM: past, present, andfuture", IEEE Transactions on Computer-Aided Designof Integrated Circuits and Systems, 19(12):1476–1497,December 2000.

[6] K.-D. Schubert ; S. S. Abrar ; D. Averill ; E. Bauman ; A. C. Brown ; R. Cash ; D. Chatterjee ; J. Gullickson ; M. Nelson ; K. A. Pasnik ; K. Sugavanam ,"Addressing Verification challenges of heterogeneous systems based on IBM POWER9",*IBM Journal of Research and Development*,2018

[7] Viresh Paruthi, *"Large-scale application of formal verification:from fiction to fact",* Formal Methods in Computer Aided Design,2010.