Exhaustive validation of a cache memory controller using Formal Verification to meet performance and timing requirements

Himani Jawa, Nishant Raman, Sini Balakrishnan, Manas Karanjekar
Intel Corporation
himani.jawa@intel.com; nishant.raman@intel.com;
sini.balakrishnan@intel.com; manas.karanjekar@intel.com;

***Abstract*-The challenges related to verification of modern-day IPs and SOCs have significantly increased because of their ever-increasing complexity. The complexity has further led to a manifold increase in the performance issues. Each performance setback can lead to multiple revisions in a project. In a design such as that of a cache memory controller, timing of multiple requests such as read, write and refresh should follow certain rules and specifications to prevent data loss, data miss and any other data integrity issue. Violations of the rules might lead to bugs, leading to performance issues. Formal Verification is an exhaustive verification based on mathematical modelling which explores the whole input space state. This paper talks about a methodology for timing performance validation which uses different formal techniques such as abstractions, range covers and assertion modelling. This helped to uncover some complex hidden bugs for different combinations of cache memory requests. It also talks about some of the issues which might have resulted in huge performance loss if those were not caught.**

I. Introduction

Cache Memory is high-speed memory that sits between a processor (CPU/GPU) and a main memory. It stores the data and instructions which are frequently used by the processor. The transfer of data between the processor, main memory and cache memory is handled by a block called cache Controller. When a request is received, the Cache controller checks if the requested information is available in the cache memory. For a hit case, information is readily available for the processor to access. But for a miss case, where the information is missing in the cache, controller helps to fetch the information for the processor and update the cache line based on the request. Hence the architecture of a cache controller is always looking for the best possible configuration to improve the performance of cache and main memory access and thus improving the performance of overall system.

II. Performance Of Cache Architecture

A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer i.e., the main memory or lower-level cache. If the cache performance is low, it impacts the performance of the whole computer architecture. The performance of a cache memory controller can be measured multiple ways such as its Hit Ratio, average access time etc. Hit Ratio is the number of hits to the total number of accesses requested to the cache memory which is the overall number of hits and misses. Average access time is dependent on the hit time (the time required to access the cache multiplied by the hit rate), the miss rate and miss penalty which is the total time needed to fetch the data from the memory whenever there was a miss.

Hit Ratio = Number of Hits / (Number of Hits + Number of Misses)
*Average access time = Hit Time + Miss Rate * Miss Penalty*

Performance improvement aims for good hit ratio and low average access time. Both the performance measures are observed to be dependent on the miss rate or the number of misses. A miss happens when the requested data/address location is not present in the cache memory. This can happen because of multiple reasons, one of most important being the requests to the cache memory getting dropped or there were any data integrity issues in the cache memory leading to a high miss rate. Data scheduler (shown in Fig. 1) in the cache memory controller is responsible for scheduling the request coming from the processor. It makes sure that all the requests are scheduled at proper timing in order to make sure that no request is dropped, no two requests overlap each other and thereby making sure that the data integrity is maintained.

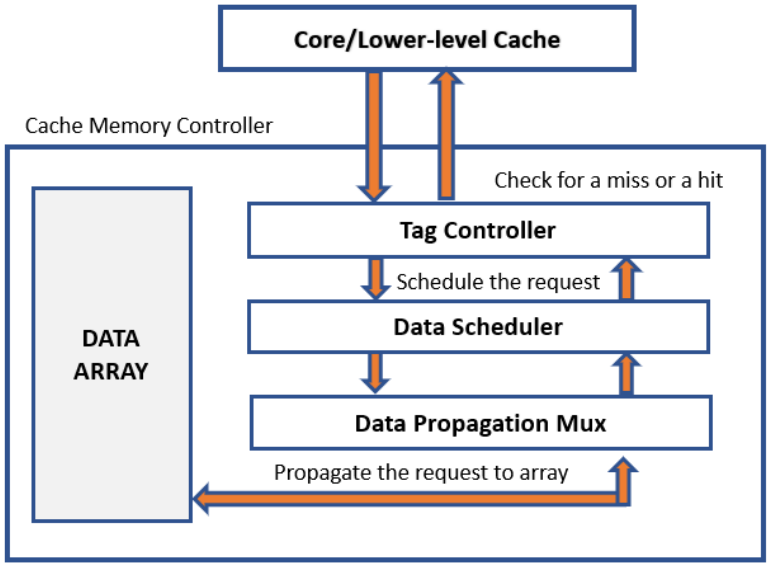

Figure 1. Cache Memory controller

III. Performance Challenges

This section talks about the different challenges which impact the miss rate and ultimately affect the performance, request scheduling being one of the major ones. Request scheduling is a critical feature which should follow different strict timing requirements so that there are no misses, and all the requests reach the cache memory controller at correct timing. The schedule of the requests should follow the below three requirements:

- Scheduling requests based on minimum timing constraints
- Avoiding request scheduling as per configurable single clock rules
- Request duplication based on Data Block (DB) timing specification for DB pair

A. Scheduling Requests based on Minimum timing constraints

Data scheduler in a cache memory controller schedules multiple requests such as read, write, refresh to the cache memory. A huge cache memory can be divided into slices, data-blocks and subarrays. Each slice is divided into data blocks, and each data-block is divided into subarrays. As the size of the cache increases, the number of slices and data-blocks also increase, leading to an increase in the timing complexity of the cache controller. For every location, slice, data-block, subarray, the timing for a request should follow a correct timing order so that all the reads and writes are happening correctly without any data corruption. With the increase in the size of the cache memory, the total number of combinations of constraints increase exponentially.

Under minimum timing constraints, any request followed by any other request for any location, a block, or a sub-block should only happen after a certain minimum no of cycles as shown in Fig. 2. There should be a minimum number of cycle gaps between the consecutive memory access to the same location to make sure there is no overlap between any 2 requests.

Case#	From cmd			To cmd			Min Clk		Data									
	Rd	Wr	Ref	Rd	Wr	Ref		Slice	Block	Subarray	clk0	clk1	clk2	clk3	clk4	clk5	clk6
1		1			1		2	diff	X	X	wr0		wr1				
2	1			1			2	diff	X	X	rd0		rd1				
3		1		1			1	diff	X	X	wr0	rd1					
4	1				1		1	diff	X	X	rd0	wr1					
5		1			1		2	same	diff	X	wr0		wr1	No Write	No Write	No Write	
6	1			1			4	same	diff	X	rd0		No Read/Write		rd2		
7		1		1			1	same	diff	X	wr0	rd1			No Write		
8	1				1		1	same	diff	X	rd0	wr1	No Read				
9			1			1	2	same	diff	X	rf0		rf1		No Refresh		
10		1			1		9	same	same	diff	wr0						
11	1				1		9	same	same	diff	rd0						
12		1			1		16	same	same	same	wr0						
13	1			1			16	same	same	same	rd0						

Figure 2. Minimum timing constraint spec for different combination of requests

The minimum clock cycles between two requests increases if the memory access happens on the same location or nearby location. As shown in Fig. 2, if one write is followed by another write, and the latter write wants to access a different slice, the minimum clock cycles is 2. However, if a write is followed by another write to the same location that is the same slice, data-block and subarray, the minimum clock cycles is 16.

As shown in Fig. 3, read to read can come in minimum 2 cycles for different slices, while read to write for same slice can come minimum after a cycle gap. Write and refresh requests for same slice, different data-block can come in the same cycle. And minimum gap between two refreshes can come after a minimum cycle gap of 2 for different slices. Different requests for different cache memory locations, slice, data-block, subarray, have different timing rules and all the possible combinations need to be validated which makes it more complex and difficult.

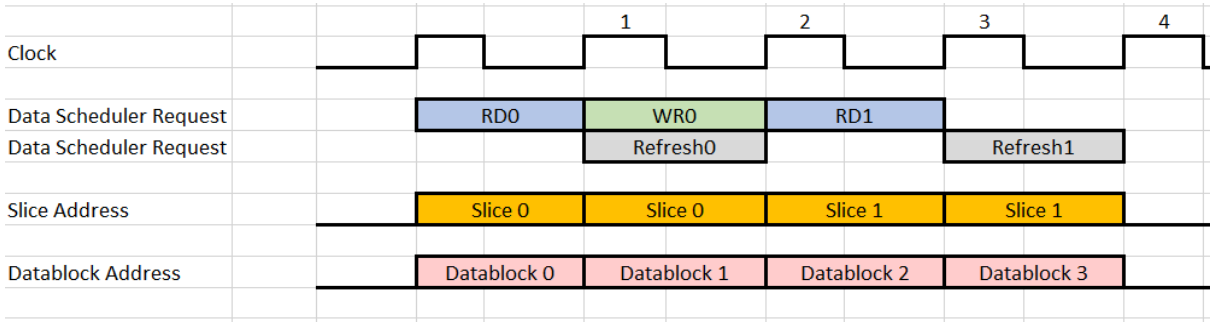

Figure 3. Request scheduling as per Min timing constraints

B. Avoiding request scheduling as per configurable single clock rules

Unlike Minimum Timing constraints, these rules make sure that the repeated memory access to a specific location is blocked from occurring at specific cycles after the initial access. A second request should not be scheduled at an exact number of cycle gap after the first request. As shown in Fig. 2, write is blocked and not scheduled on 3rd,4th,5th cycles after a write on the same slice and 2 cycles after a read. The read should not be scheduled 4 cycles after a write on the same slice and 2 cycles after a read.

Some of the single clock rules can be configured using configurable registers. Based on the value of the register, the request does not get scheduled at the specific cycles. For example, as shown in Fig. 4, w2w_same_slice register is used for the write to write for same slice. If w2w_same_slice's value is 'b11101, then write should not appear 1,3,4,5 cycles after a write on the same slice. Such single clock rule registers help block requests at different cycles for different configurations and find the best performance configuration of cache memory controller.

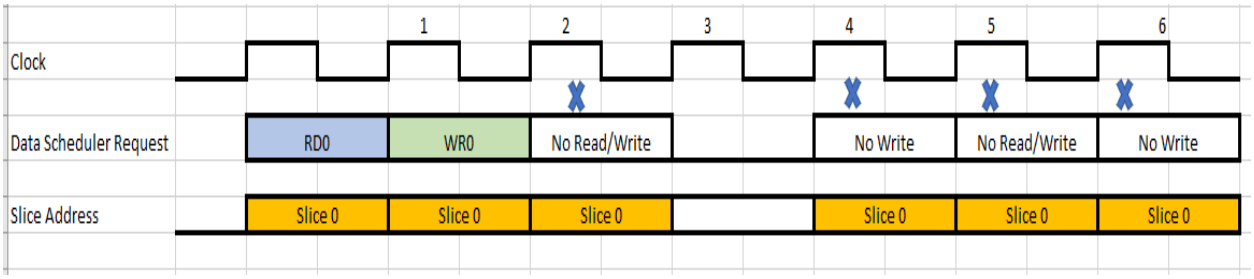

Figure 4. Write blocking as per single clock configurable register

C. Request duplication based on Data Block (DB) timing specification for Half DB pair

The requests from the data scheduler are forwarded to the data propagation mux which further sends it to the cache memory array. Data propagation mux duplicates each request coming from the data scheduler for each half DB.

Since the requests to the cache memory are duplicated, the timing complexity increases, so therefore there are specific DB timing requirements and more robust validation is required to make sure no request is overlapped or dropped. For read, the duplicated request for each half DB gets generated after a cycle gap. For write, the duplicated request for each half DB gets generated after 3 cycles gap, at the 4th cycle.

For example, as per Fig. 5, read after read for the same slice should be scheduled after a minimum 4 cycle gap, let's say it gets scheduled at the 2nd cycle. Then the second read request gets dropped because of the overlap between the duplicated read requests between the two reads.

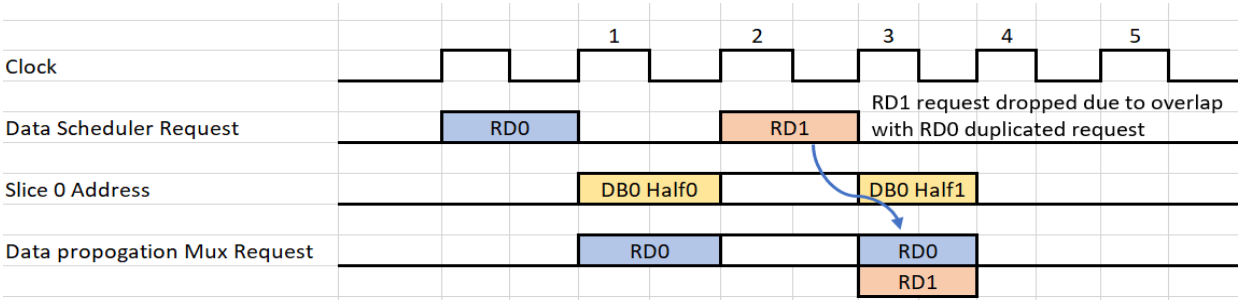

Figure 5. Request blocking as per single clock configurable register

With such huge timing complexity, different types of timing requirements and an exceptional huge number of timing rules to be verified, bug escape is very easy. Dynamic simulation on one hand can only verify certain cases since it is dependent on random test stimuli, Formal verification and advanced Formal Techniques pave the way for us to achieve no bug escape.

In the next section we talk about the different FV techniques that helped in exhaustively verifying the timing performance of a higher-level cache memory controller IP and how formal verification Methodology and techniques helped in not only validating the existing performance but also helped in optimizing the performance. This IP provides large capacity medium bandwidth cache to improve the performance and throughput in discrete graphics products.

IV. Formal Techniques To Resolve Performance Challenges

We propose a formal methodology to exhaustively validate the timing performance of the cache memory controller. The data scheduler is responsible for scheduling requests according to the different timing requirements as discussed in the previous section.

In stage 1, **Assertions** and checkers in FV for Min timing constraints, configurable single clock rules and the DB timing requirements have been modelled and developed. If any assertion failed, that means it is not following the timing requirement and is affecting the functionality of the cache memory controller, therefore it wase reported as a bug.

In Stage 2, if the minimum timing constraint assertions passed, we verify that scheduling operation happened on the minimum timing as specified in the spec using **Covers**. If the cover traced passed, then the scheduling operation was happening at the minimum timing possible, therefore leading to the best performance possible for that scheduling. If the cover trace fails, we move to stage 3.

In stage 3, we find the minimum possible cycle using **Ranged Covers** at which the minimum timing constraint is satisfied and the cover passes. Since the design complexity is huge, the covers convergence was difficult. This is when **Abstractions** were used and helped in the convergence of the covers and reduced the overall complexity. The cover traces are then analyzed, and feedback is given to improve the performance.

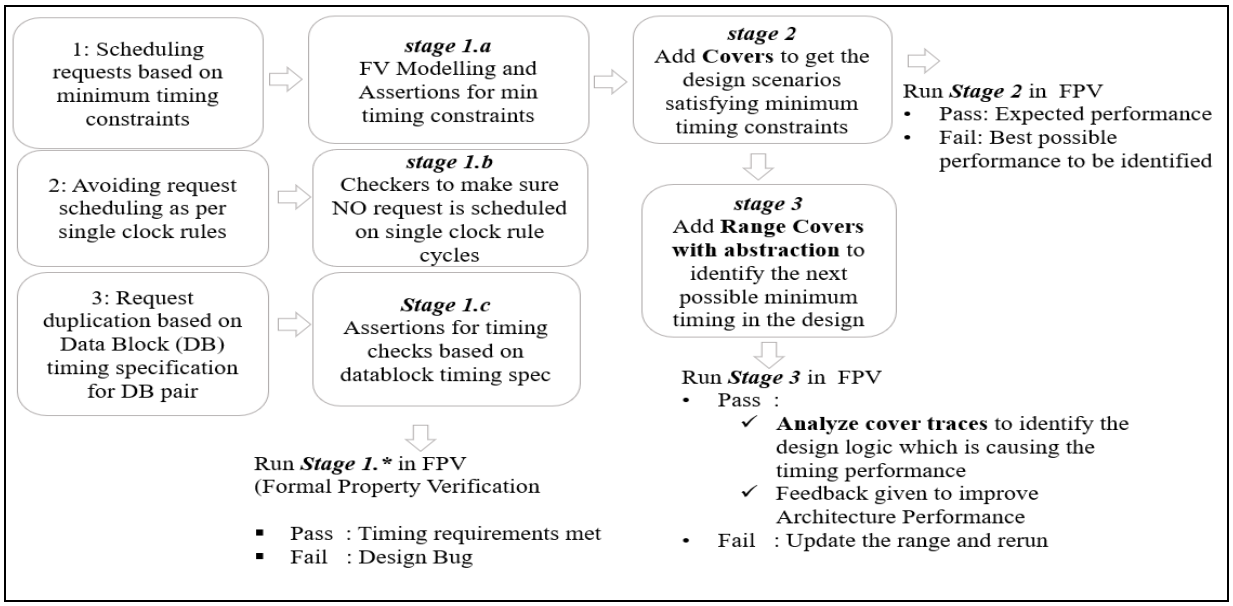

Figure 6: Overall Methodology to validate the performance of a cache memory controller architecture

Given the various timing and performance challenges, the complexity of the rules constraining the design is varied. Depending on the complexity of the rules, different Formal property implementations can be applied keeping in mind the optimization of the FV model. This section explores different Formal Verification techniques to verify the timing performance of a cache memory controller to minimize the miss rate.

A. Assertions

Various kinds of assertions have been used to exhaustively verify the minimum timing constraints, the single clock rules and checkers for the DB (data-block) timing spec as shown in Fig. 6. Assertions were modelled for each type of specification for all the different combinations of requests scheduled for each array and subarray, as shown in the sample specification in Fig. 2. The assertions can be classified into 4 categories based on the features they verify:

- Logical expression-based assertions such as the one in Fig. 7 are effective in verifying direct properties such as two requests not being scheduled in the same clock cycle. They are of the least complexity in terms of both code size and number of flops generated.

```
asrt_timing_same_array_no_rd_with_wr: assert property (
    @(posedge CLK) disable iff (RESET)
  !(write_same_array && read_same_array)
);
```

Figure 7: Logical expression-based assertions

- Assertions modelled with additional logic such as counters are more involved due to the supporting logic modelled along with it. They have been used to verify the minimum timing criteria of the various request combinations. The counters are triggered by an initial request that needs to be tracked. As the counter value increments, the assertion checks for new requests and if they are scheduled within the legal timing comparing with the counter and the specified minimum timing value. Combinations of such checkers are effective in verifying the more complicated single clocking requirements. The sample code in Fig. 8 and Fig. 9 show one such implementation to verify the case where a write request must not appear in clocks 1,3,4,5 after the initial write request to a particular data array. The first assertion ensures the absence of a repeated write request until

cycle 6, except for the 2nd clock cycle. If a write appears in the 2nd clock, the counter for the second assertion is triggered, where the latter ensures the absence of another write request in the consequent 5 cycles. Together, the pair of assertions cover the entire single clocking requirement.

```
asrt_timing_same_array_wr_after_wr_1: assert property (
   @(posedge CLK) disable iff (RESET)
   (counter_same_array_wr_1 > 'd0) &&
   (counter_same_array_wr_1 < 'd6) &&
   (counter_same_array_wr_1 != 'd2)
   |->
   !(write_same_array)
);
```

Figure 8. Checker1 for single clocking rule

```
asrt_timing_same_array_wr_after_wr_2: assert property (
   @(posedge CLK) disable iff (RESET)
   (counter_same_array_wr_2 > 'd0) &&
   (counter_same_array_wr_2 < 'd6)
   |->
   !(write_same_array)
);
```

Figure 9. Checker2 for same single clocking rule

- Single clocking requirements can also be implemented using configurable registers in the design which allows for variable single clock rules. The assertions to verify these compare the register value for the clock cycle in which they block a consecutive request with the arrival of a new request. A sample assertion in Fig. 10 verifies the case where a configurable register (W2W_CR_3) is blocking a write request 3 clocks after the first write.

```
asrt_timing_same_array_wr_after_wr_clk3: assert property (
    @(posedge CLK) disable iff (RESET)
    write_same_array_d3
    |->
    !(W2W_CR_3 && write_same_array)
);
```

Figure 10. Assertions using configurable register

B. Covers

Beyond precondition covers generated for assertions by the formal tool, targeted covers have been used to minimize the search space and view the failure scenarios faster. These also allow verification of certain properties such as performance requirements where the occurrence of an event is a sufficient indicator rather than the perpetual occurrence of an event or a sequence of events.

In the verification effort towards a cache memory controller, dedicated covers can be used to check for the minimum possible, legal repeated memory accesses. For example, if a minimum timing constraint between consecutive write requests is 10 clock cycles, the cover property checks for the earliest occurrence of a consecutive write request at the 10th clock cycle. Fig. 11 shows a code snippet of one such cover property in System Verilog HDL.

```
covr_timing_same_array_wr_after_wr_16_clk: cover property (
    @(posedge CLK) disable iff (RESET)
  write_same_array ##16 write_same_array
)
```

Figure 11. Cover for Wr after Wr Min timing constraint

Along with targeted covers, modified cover properties **c**an also be used to view the earliest occurrence of an event within a time-period or a range of clock cycles. For convenience, they shall be referred to as range-based covers henceforth. These range-based covers look for a repeated request in a specified range of clock cycles based on performance and search depth requirements. Assuming a read request is expected in a specified range, the cover would, in a well abstracted setup, generate a trace for the earliest possible repeated read in that range. Fig. 12 shows the System Verilog code for a sample range-based cover.

```
covr_timing_same_array_wr_after_wr_range: cover property (
    @(posedge CLK) disable iff (RESET)
  write_same_array ##[16:20] write_same_array
)
```

Figure 12. Ranged Cover for Wr after Wr Min timing constraint

Using the range-based cover technique can be thought of as a modified version of using a divide-and-conquer strategy to help observe waveforms. What this means is, one can provide an estimate of bound or cycles-based range to the formal tool, during which target behavior is expected, and this range can then be modified to generate a meaningful trace as illustrated in Fig. 13. The generated trace will help one sieve through the source code for possible failures. They also help assess the current performance capability of the design.

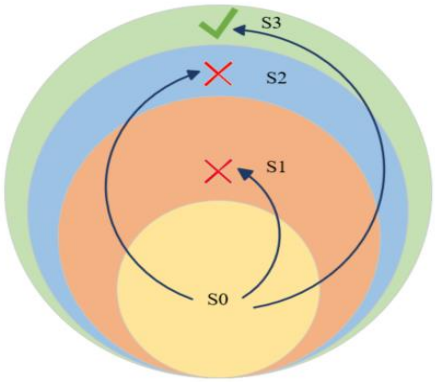

Figure 13: Devising covers for different bounds for a desired state

C. Abstractions (AM)

Unlike simulation, formal has certain limitations when it comes to the design size, precisely the number of state elements that a tool can process. This is primarily because of the nature of how formal works. With every new state element added, the state space increases exponentially, thereby leading to design complexity issues.

We encountered such complexity issues as a part of our verification effort. Some properties were not converging after being run for a long amount of time, which helped us identify this problem as a first step. Two cases when formal verification efforts hit a roadblock are, first when covers fail, and second is when properties don't converge. Both the issues can be tackled by the range-based covers approach. As mentioned earlier, writing these covers enables debugging through waveforms, and help trace driver signals through the RTL.

The generated behavioral waveforms helped identify a bottleneck matrix which was adding complexity to the tool. The matrix functioned as a hardware-based algorithm to identify and evict instructions received by the cache controller. Implementing such algorithms in hardware often requires keeping track of multiple features, such as age of an instruction, priority of an instruction, and such others, thereby requiring memory elements to be added to the design. These memory elements contribute to complexity and require a resolution. If we were to replace the matrix with something more lightweight, and yet have the same computational effect in downstream logic, we could get past this complex hurdle. This is the essence of abstraction, replacing a complex piece of logic with logic that provides similar functionality, and simplifies the formal analysis process for the tool as illustrated in Fig. 14. We identified the functionality of the matrix, and the logic it interacted with, and replaced it with an abstracted model with the same interface. The abstract model consisted of properties which acted as assumptions for the tool and complied with rest of the design logic just as the original matrix. This technique helped achieve convergence on properties within 20 minutes, which otherwise were taking over 72 hours to reach the same bounds, thereby reducing the turnaround time by ~99.5%. This can have a significant impact on left-shifting the design verification timeline and identifying early bugs in the design.

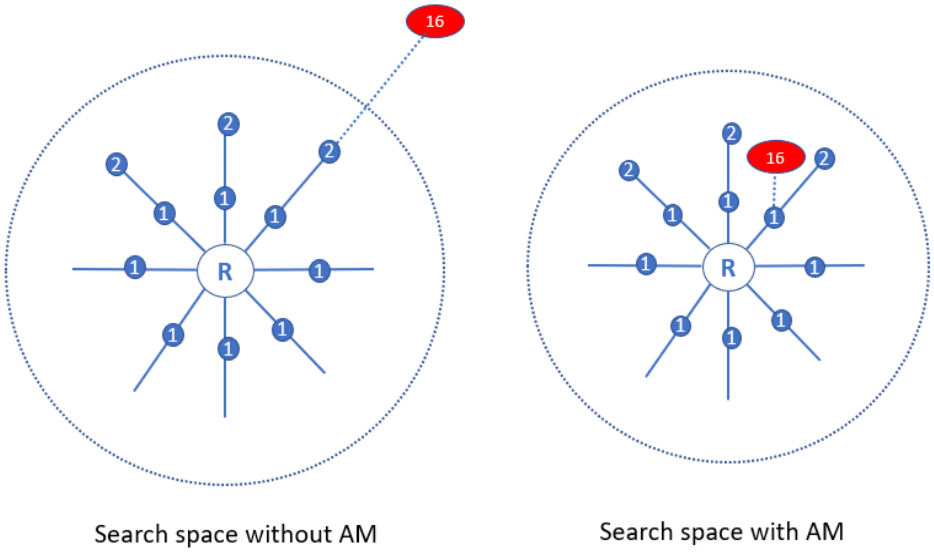

Figure 14. State space without and with Abstraction

V. Results & Discussion

Using this innovative approach, multiple hidden bugs were found in the cache controller architecture. Two bug scenarios which had a huge impact on performance are discussed below.

A. Bug Scenario 1

In the case of single clocking constraint where a repeated write request was to be blocked for clocks 1,3,4,5 after the initial write request. In other words, a consecutive write may be expected at cycle 2 and 6 onwards. The issue in the design revealed to be repeated write requests being blocked for all 5 cycles.

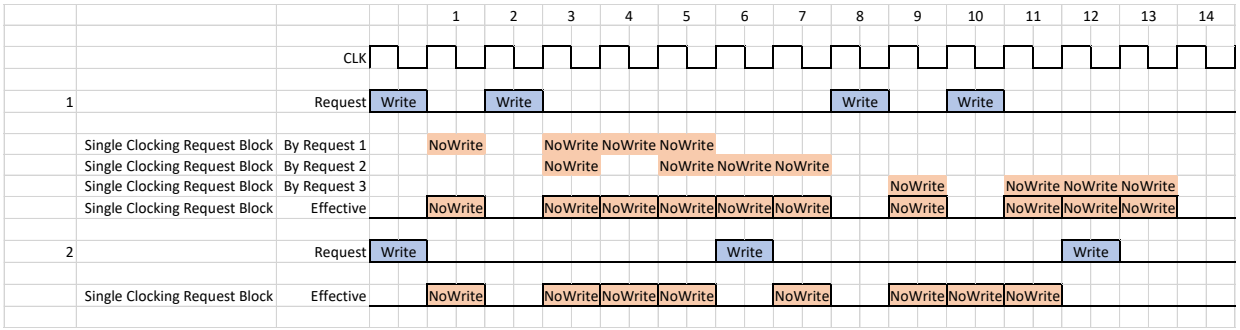

Figure 15. Bug Scenario1- Wr after Wr for single clock rule

Scenario 1 in Fig. 15 shows the peak performance expectations from the architectural specification, while scenario 2 depicts the observed bug. At peak performance, assuming the requests repeat as shown, 2 writes every 8 cycles is expected, while the implementation allows for 2 writes every 12 cycles. The missed write request resulted in a 33% (1 – 8/12) decrease in write bandwidth. Effectively 1 out of every 3 writes are being dropped, and in cache memory design this adversely affects the cache miss rate.

The implementation for verifying this functionality was a pair of assertions. The first assertion simply constraints repeated requests after an initial write but does not check in the 2nd cycle (as per the specification). The second assertion is specific to the writes request that could appear in the 2nd cycle. Together, the pair of assertions cover the entire single clocking requirement. Depending on the complexity of the single clocking requirement, more checks may be introduced that help cover the specification. The bug was discovered through a simple precondition cover

that is generated for the second of the pair of assertions. It looks for the specific write that appears in the 2nd cycle after an initial write request. Failing to see one resulted in a cover failure which exposed this critical performance issue. The above bug scenario reveals that through existing checkers, the search for performance-bug hunting widens.

B. Bug Scenario 2

Another performance requirement from the specification is that requests are scheduled as early as possible while meeting minimum legal constraints. While requests are not expected to always repeat at the minimum timing imposed by the specification, they must be allowed to do so to ensure no loss in memory access bandwidth. For example, a consecutive read request can't be scheduled within 16 cycles of each other, as a minimum timing constraint, but they may be expected at any cycle after that. An issue observed associated with these minimum constraints was that while the read requests adhered to the minimum legal constraints, a request was never scheduled at the minimum possible cycles, i.e., 16.

The verification methodology employed in these performance critical areas was through covers and ranged covers. The unreachability of the targeted cover for a consecutive read request after exactly 16 cycles present a formal proof of the bug. This is important in such scenarios where simulation-based verification techniques may not give the surety of the presence of a failure or simply a rarity in occurrence.

In cases where the covers are failing, the traces do not get generated. In such kind of bugs, where cover failures are an indication of the bug, the lack of a violation trace may be unhelpful in debugging the failures since the failures can't be pinpointed directly. There are some workarounds to this that would help trace through the source code/RTL. One is the use of the FV technique discussed previously, range-based covers, which are useful in partially combatting the issue mentioned earlier regarding the lack of violation traces.

These also have the added advantage of highlighting the extent of performance loss in this scenario. Experimenting with a variety of cover ranges to estimate the degree of failure, the earliest cover trace showed a 30-cycle difference between consecutive read requests. The performance loss here is very evident – the best-case scenario presented by the design is 1/30 requests per cycle and the expected bandwidth by the specification is 1/16 requests per cycle. A drop of almost 47% (1 - 16/30) in read bandwidth! Having such an indicator allows designers and architects to easily identify core issues and fix them with priority.

While such bugs are not critical to functionality, they impact the performance and the potential bandwidth of memory accesses.

C. Overall Results

Using this novel methodology, more than 20 bugs were found including several corner-case bugs which if not caught would have impacted the performance adversely. As discussed in the above two corner case bug scenarios, missing those bugs would have **reduced the architecture performance by 33% and 47%** respectively.

We were successfully able to validate the different timing requirements to meet the stipulated cache controller architecture performance. Covers and Range Covers with abstractions helped to identify the minimum timing supported by the design. Using different cover traces in the subsequent stages (Stage 2, stage 3) the design areas which hinder the performance were root-caused quickly. Based on the feedback, various modifications in the architecture and design implementation were carried out and several performance bottlenecks were identified while applying this innovative method.

CONCLUSION

We presented a comprehensive methodology to validate all kinds of timing performance challenges of a cache controller architecture that yields impactful results. This helps to prevent data loss, reduce data-misses and avoid other data integrity issues. Using this innovative technique, the best possible performance can be identified. Flaws in the architecture can be root caused if there is a performance setback. We could also showcase that several corner case issues can be exposed with minimal code-space properties and by reducing the design complexity. Underlying formal techniques greatly speeds up the validation process and consequently leads to better designs with high ROI. Different test scenarios are used to demonstrate the effectiveness of this methodology to bridge the gap between 'a sense of an issue' and 'a confirmed bug'. The same methodology applied to all such requirements can multiply the benefits and yield better designs with greater performance.

ACKNOWLEDGMENT

Thanks to DDG Design and Architecture Team for the support and FVCTO Team for the motivation and the valuable feedback.

REFERENCES

[1]M, Achutha KiranKumar, Erik Seligman & Tom Schubert, Book on "Formal Verification – An Essential Toolkit for VLSI Design", 2015
[2] IEEE Std 1800™-2017, IEEE Standard of System Verilog – Unified Hardware Design, Specification, and Verification Language.
[3] Pierre Wolper, "Expressing interesting properties of program in propositional temporal logic" 13th ACM SIGACTSIGPLAN symposium on Principles of programming languages, pages 184-193. ACM Press, 1986.
[4] T. Patel, "Using Formal Sign-Off to Deliver Bug-Free IPs", Oski Decoding Formal Club, Dec 2019