2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Pseudo-LRU Not Efficient in Real World? Use Formal Verification to Bridge the Gap

Paras Gupta, Sachin Kumawat, and Kevin Bhensdadiya
Intel Corporation Pvt. Ltd.
{paras.gupta sachin.kumawat and kevin.bhensdadiya}@intel.com

***Abstract*- In hardware systems, a set associative cache is a critical component utilized to store data that has been recently accessed, in N number of ways. Whenever a read request is made, the cache can either be a hit or a miss. If there is a cache miss and the cache is already full, then one of the cache ways must evict its old data to make space for the recently accessed data. To facilitate this eviction in the cache, we employ the Pseudo Least Recently Used (PLRU) mechanism. PLRU is used to identify a way in a cache set which can be replaced by the entry reading the cache in case of cache miss. As the name "pseudo" suggests PLRU does not keep track of exact age of cache lines, but an approximation measure of age which makes PLRU more resource efficient than Least Recently Used (LRU). PLRU is prone to corner cases bugs like not accessing some ways of cache set after a set of events as it is just an approximation not absolute like LRU. One of the feasible ways to catch those corner case bugs is Formal verification (FV). FV is a technique to verify complex digital design using mathematical models. It is gaining a lot of popularity day by day because of the results it is producing on simulation clean designs. In this paper we will discuss an approach to formally verify PLRU mechanism.**

I. INTRODUCTION

PLRU mechanism is inspired from an LRU mechanism which keeps track of the least recently used entry. Whenever a new entry needs to be written to cache, the least recently used entry is evicted from the cache and is replaced by the new entry. However, LRU RTL design is concerned by the possibility of many corner case bugs which can frequently escape traditional simulation verification techniques as they are very deep and complex. Therefore, FV is one of the ways to find hidden bugs in algorithms like LRU.

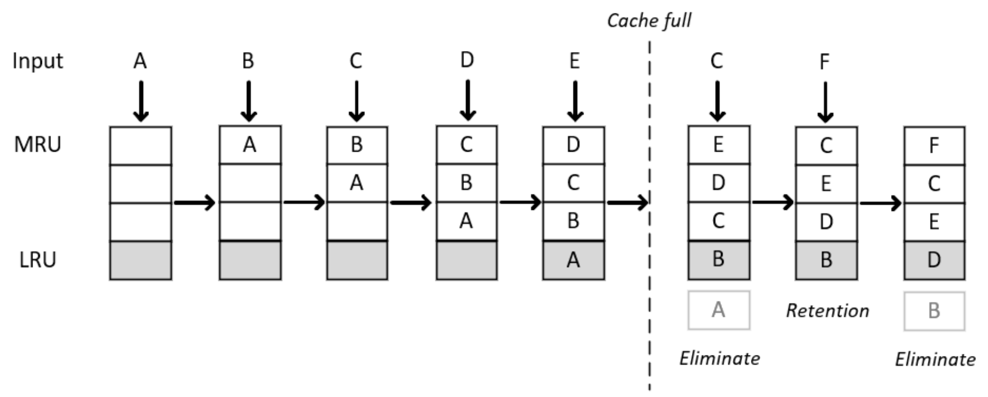

Figure 1: LRU Mechanism

A. *LRU comprehensive test planning for formal verification*

FV test planning is an initial step in exhaustive verification of LRU. Without excellent test planning, we will not be able to focus on critical buggy areas of design and overall FV Sign-off might result in an inefficient effort. Therefore, test planning can be considered one of the most important steps in the FV Sign-off process which can be done in the following steps:

1) *Identification of Design under test (DUT)*

DUT identification refers to the process of accurately identifying and isolating the specific component or system that needs to be tested within a larger design. It plays a crucial role in the design verification process by enabling focused testing, efficient debugging, and optimization of resources.

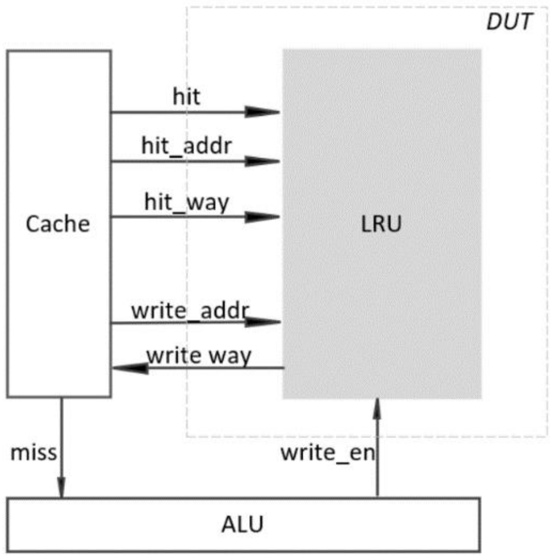

Figure 2: Identification of DUT

2) Prepare a test plan

Test plan preparation in formal verification (FV) is a process that can effectively evaluate the LRU design, validate its properties, and detect any potential issues or bugs. In order to verify an LRU exhaustively we came up with a list of properties.

i) When a cache is not full and a cache miss occurs, then write should happen at one of the invalid entries

ii) When a cache is full and a cache miss occurs, then write should happen on the oldest valid entry present

B. *Decoding PLRU: Unveiling the Working Principles*

The PLRU replacement policy offers advantages over the LRU policy in terms of efficiency and hardware requirements. Specifically, the number of state bits needed to track PLRU is N-1, where N represents the number of ways in the cache. This number is significantly lower than the state bits required to track LRU, which is the ceiling(log2(N!)).

For example, let's consider an 8-way cache line where N is 8:

- Number of PLRU bits required = N - 1 = 8 - 1 = 7
- Number of LRU bits required = ceiling(log2(N!)) = ceiling(log2(8!)) = 16

In the case of PLRU, there are only 7 state bits required to track the cache's status, whereas LRU requires 16 bits. This difference becomes more pronounced as the number of ways in the cache increases.

Figure 3 demonstrates the structure of a PLRU tree for an 8-way cache, with n0 to n6 representing nodes of PLRU tree, d1 to d8 representing ways of cache line and L0 to L1 representing levels of PLRU tree. Following arrow path from L0 to L2 we will get way to get evicted based on PLRU mechanism. The reduced number of nodes in the PLRU tree compared to the LRU implementation contributes to lower hardware complexity and power consumption.

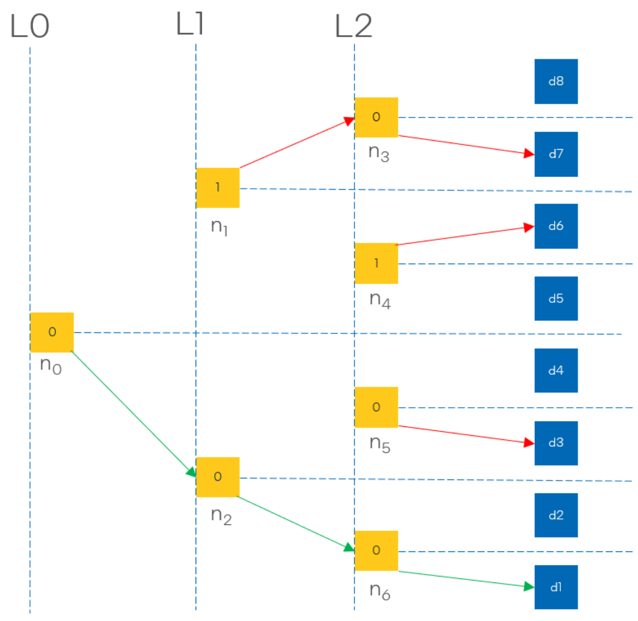

Figure **3**: PLRU tree for 8 cache line

Consequently, the adoption of the PLRU mechanism not only enhances overall hardware performance but also minimizes power consumption due to its reduced number of state bits. This efficiency makes PLRU an attractive choice for cache management in terms of both performance and power optimization.

However, unlike LRU which works on one basic principle of evicting the absolute oldest entry, PLRU works on 3 general principles listed below:

1) Just after the cache got full PLRU tree will be pointing to the oldest data in the cache i.e., d1
2) If a hit occurs on both sides or on neither side of any given node, then the polarity remains unchanged. Alternatively, if a hit occurs on only one side of a parent node, then the polarity of the parent node is examined. If the parent node is pointing towards the side that experienced the hit, then its polarity will be reversed; otherwise, the polarity remains the same. In figure 4 shown below hit is occurring on d5:
    - Polarity of node 6, 5, 3, 2 -> remains same because neither of the sides gets hit
    - Polarity of node 4, 1 -> parent node is pointing towards the side that experienced the hit, then its polarity will be reversed
    - Polarity of node 0 -> parent node is not pointing towards the side that experienced the hit, then its polarity remains the same

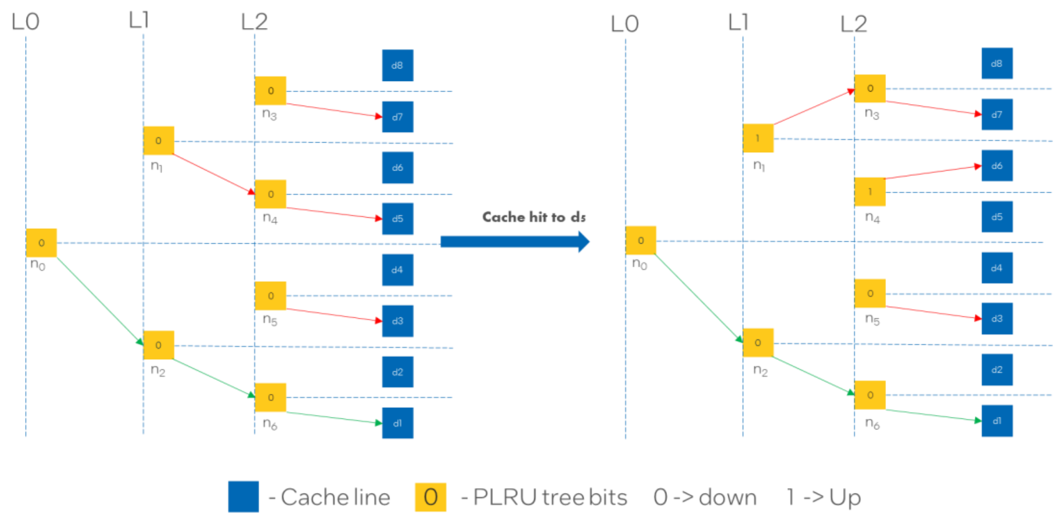

Figure 4: PLRU tree transition if there is a hit on d5

3) Cache miss is given the utmost priority i.e., in case of cache miss we only look at cache miss irrespective of any other cache hit. Below d9 replaces d1

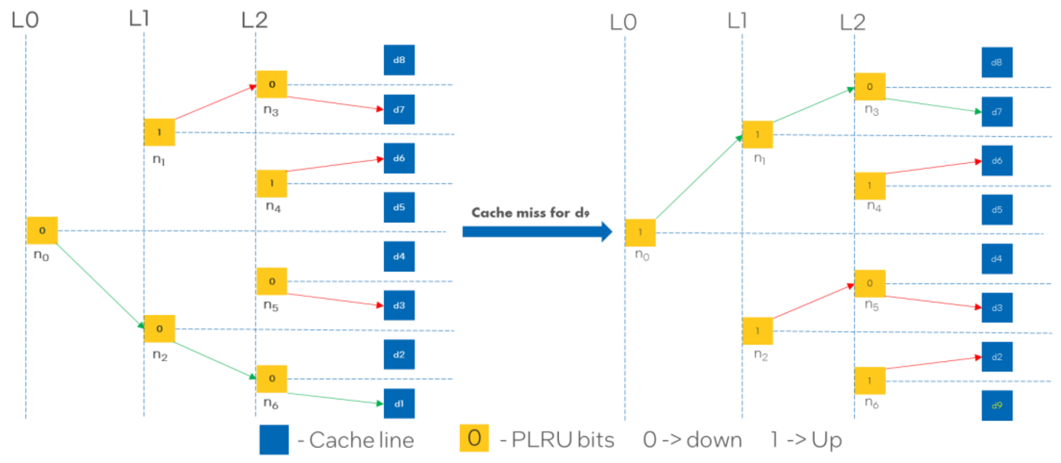

Figure 5: PLRU tree transition if there is a miss for d9

- Polarity of node 5, 4, 3, 1 -> remains same because neither of the sides gets hit
- Polarity of node 6, 2, 0 -> parent node is pointing towards the side that experienced the hit, then its polarity will be reversed

II. APPLICATION

The PLRU mechanism is derived from the LRU mechanism, resulting in its test plan to be slightly a modified version of LRU test plan.

A. Initial test plan to verify PLRU

Initially, for the PLRU mechanism, we came out with 2 generic properties that it should follow i.e.,

1) When the cache is not full and a cache miss occurs, then write should happen on one of the invalid ways

2) When the cache is full and a cache miss occurs, then write should not happen on the most recently used way present.

But based on the implementation of Cache there are certain challenges associated with the above test plan of PLRU

B. Challenges with the initial test plan

The initiation of PLRU verification using the initial test plan became increasingly difficult once the FV tool started generating various counterexamples (CEX) that posed challenges. One such instance was when property #2 failed to hold true in the presence of a multi-hit cache line, where multiple hits were supported within the same cycle and the most recently used (MRU) cache line would become the PLRU in the subsequent cycle. In Figure 6, it is evident that the PLRU tree points to d6, and we experienced cache hits for cache lines d4, d5, d6, d7, and d8. Consequently, in the next cycle, d6 becomes one of the MRU cache lines, but due to the approximate nature of the PLRU mechanism, the PLRU tree continues to point exclusively to d6.

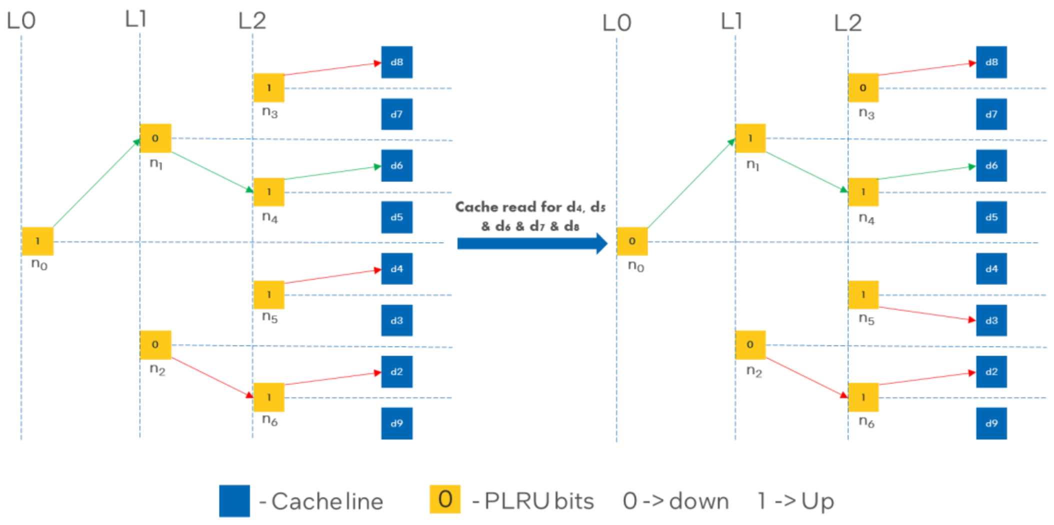

Figure 6: PLRU tree transition where MRU becoming PLRU in next cycle

Most of the CEX ended up being exceptions of PLRU, which prompted us to come up with a further relaxing of 2 generic PLRU properties.

C. Final test plan

We further relaxed our properties to deal with the above exceptions and came up with a final exhaustive set of directed properties:

1) When cache is not full then victim cache line should happen on one of the invalid cache lines

2) If there is neither cache hit nor cache miss, then victim cache line should remain same

3) If there is one cache miss followed by another cache miss, then if first victim cache line is in first half of PLRU tree then next victim cache line should be from another half of PLRU tree provided there is no cache hit or invalid ways

4) If there is a cache hit followed by cache miss, then if hit way is in first half of PLRU tree then next victim cache line should be from another half of PLRU tree provided there is no cache hit or invalid ways

5) If all the cache hits are in first half of PLRU tree, then in the next cycle victim cache line should be from another half of PLRU tree

6) If both the half of a PLRU tree get hit, then in the next cycle victim cache line should be from same half of PLRU tree

Hence verifying PLRU accuracy presents a complex challenge for verification engineers, owing to its inherent nature of approximation. Often, hidden corner cases that evade simulation verification may significantly impair the hardware design's performance. For instance, during the verification process, a corner case bug was discovered due to an erroneous implementation of the PLRU design, resulting in 50% of the cache lines being inaccessible following a set of events. Such corner cases often go undetected during simulation, necessitating the use of FV.

III. RESULTS

Formal methodology allowed us to verify the functionality of PLRU design through multiple assertion properties. Depending on the width of hit-vector in a given design both initial and final test plan could be employed to verify PLRU through FV. Through this approach, we were able to catch over 15 bugs in PLRU. Some of the bugs, being very corner case, would have affected the resource utilization of the design drastically, e.g., PLRU was evicting the Most Recently Used (MRU) entry under some scenario, eventually reducing the performance of SOC. The bugs found by FV were analyzed across the IP and some of them were found in multiple other flavors of PLRU, which allowed timely fixes and increased confidence in the IP.

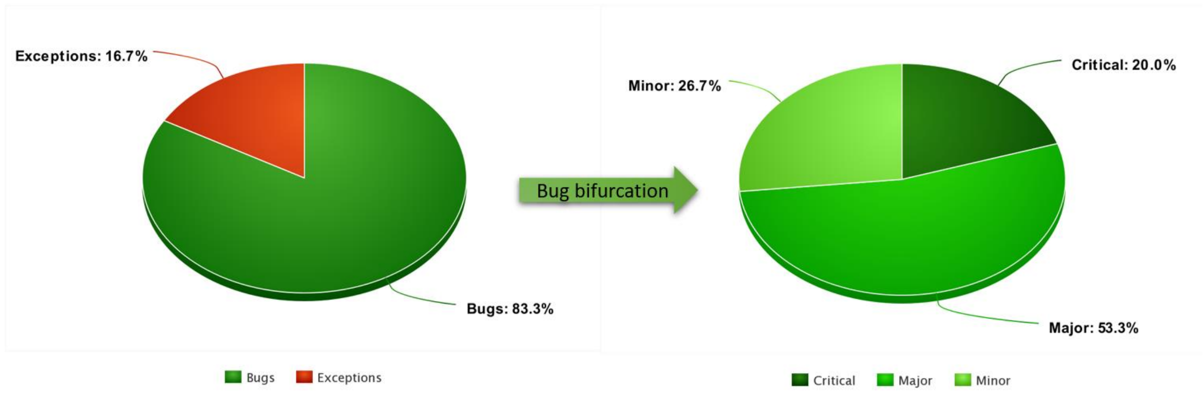

Figure 7: Total issues found

A. *Example Bug*

During the final test plan, we successfully detected and resolved more than 15 bugs in the PLRU design. One of the critical bugs, which was a rare occurrence and could have been challenging to discover through conventional simulation methods, was uncovered. Thanks to the powerful capabilities of FV tools, we were able to identify this bug within just 10 cycles by generating minimal counterexamples (CEX). This made the debugging process significantly easier. This bug was identified through the implementation of a specific property: "PLRU should not victimize the same way in consecutive victimization for same address*”*.

The property was implemented as follows-

```
//Tracker parameters
parameter track_addr = 23;
parameter track_page = 4K;

//Glue logic to store MRU way
logic [1:0] fv_mru_way;
logic fv_mru_stored;


always@ (posedge clk) begin
   if (rst) begin
      fv_mru_way <= '0;
      fv_mru_stored <= '0;
   end
   else if (wen & (wraddr==track_addr) & (page_type==track_page)) begin
```

```
      fv_mru_way <= victim_way;
      fv_mru_stored <= 1'b1;
   end
end
//Property: PLRU should not victimize the same way in consecutive victimization for same
address
PLRU_shld_not_victmz_same_way: assert property (
   @(posedge clk) disable iff(rst)
   (wen & (wraddr==track_addr) & full & (page_type==track_page) & fv_mru_stored)
   |->
   (victim_way != fv_mru_way)
);
```

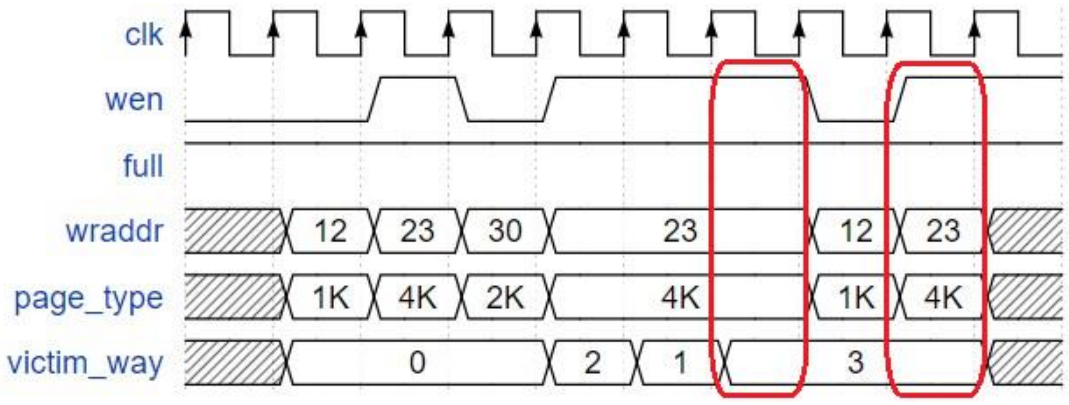

Figure 8: CEX waveform

IV. CONCLUSION

By utilizing a formal methodology, we conducted a thorough verification of the PLRU design, employing multiple assertion properties and employing both initial and final test plans to validate the PLRU through FV. This meticulous approach enabled us to detect and rectify more than 15 bugs present in the PLRU.

Some of the identified bugs were particularly challenging, as they represented rare corner cases that could have had severe consequences on the resource utilization and overall performance of the System-on-Chip (SOC). Notably, one critical bug caused the PLRU to erroneously evict the Most Recently Used (MRU) entry in specific scenarios, adversely affecting the SOC's efficiency.

Significantly, the bugs discovered through formal verification were thoroughly analyzed across the entire IP, revealing that some of them were present in multiple variants of the PLRU design. This realization facilitated prompt remediation actions and instilled greater confidence in the reliability and robustness of the Intellectual Property (IP).

In conclusion, the utilization of formal methodology, assertion properties, and FV in the verification process proved highly effective in uncovering and rectifying various bugs within the PLRU design. This comprehensive approach not only enhanced the performance of the SOC but also ensured the successful implementation of the PLRU across different versions, bolstering the overall quality of the IP.

V. REFERENCES

[1] H. Ghasemzadeh, S. Mazrouee, and M.R. Kakoee “Modified pseudo LRU replacement algorithm,” 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)

[2] Ke Zhang, Zhensong Wang, Yong Chen, Huaiyu Zhu, and Xian-He Sun, “PAC-PLRU: A Cache Replacement Policy to Salvage Discarded Predictions from Hardware Prefetchers,” 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid