2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Solving Formal Complexity for Linked List Hardware Designs

Ankit Saxena, Radheshyam Baviskar, Shubhangi Goel
Marvell Technology
Pune, India – 411014

***Abstract*— Exhaustive verification of hardware linked lists has become extremely important to ensure no hardware issues such as dead-lock or memory leakage happens due to them. Because of micro-architecture defined concurrency (for example, concurrent updates to head-pointer, tail-pointer, links and free-pointers for different linked-lists IDs) in the linked list designs, it is very hard to get sufficient coverage and confidence using traditional simulation-based approach. Formal verification, due to its exhaustive nature [1], can provide exhaustive coverage and confidence – though only after we solve formal complexity i.e., we ensure that bounded proofs have reached sufficient depths [2]. In this paper, we will discuss how we can solve formal complexity for linked list designs. First, we will discuss the micro-architecture components of linked list designs (such as free-pointers initialization sequence, various memories) which cause formal complexity either due to high-sequential depth of scenarios or due to high flip-flop count [2]. Then, we will discuss how we can write efficient checkers in formal using symbolic variables [1]. Then, we will discuss the complexity solving methods which can be used to solve the formal complexity and more importantly we will discuss how to apply them on linked list designs. Here we will discuss how to apply memory abstraction models on head-pointer memory, tail-pointer memory, data memory and free-list memory. After that we will discuss why we cannot apply memory abstraction on links memory. Then, we will discuss how to apply abstraction models for size counters and initial value abstraction for the case when free-list needs to be initialized. In the end, we will see the results where we applied this methodology on a linked list design and how we were able to achieve 100% coverage for a linked list design having a large number (2,048) of pointers.**

Keywords—hardware linked lists, formal verification, abstraction models, end-to-end formal verification

I. INTRODUCTION

Linked lists are fundamental components in hardware designs, known for their dynamic memory allocation and efficient insertion and deletion operations. In hardware design, linked lists are employed in various applications due to their flexibility and efficiency in managing variable-length data structures. A few examples of use-cases of linked lists are:

- **Task scheduling:** In Scheduler designs, linked lists are used to maintain task queues. Each task in the queue is represented as a node in the linked list, allowing for dynamic addition and removal of tasks based on priority and execution time.
- **Buffer management:** Linked lists are utilized in managing communication buffers in Buffer Manager designs. They enable efficient handling of variable-length data packets, ensuring that memory is allocated only as needed.

The correctness of linked list implementations in hardware is paramount for several reasons. For example:

- **Data integrity:** Incorrectly implemented linked lists can lead to data corruption, where data nodes may be lost or overwritten. This can have severe consequences, especially in critical systems such as medical devices or automotive control systems.
- **System stability:** Errors in linked-list operations, such as incorrect pointer manipulations, can cause system crashes or unpredictable behavior. Ensuring the correctness of linked lists helps maintain the stability and reliability of the overall system.

Since the linked list designs are critical for system, and they have concurrency in updating internal data structures such as head-pointer, tail-pointer, links and free-pointers, they should be verified using formal to ensure the exhaustive verification and no coverage holes [4]. However, verifying them using formal comes with its own challenges of solving complexity and ensuring that the bounds achieved with bounded pass are good [2]. In the next sections, we will discuss about these challenges and how to solve them.

II. LINKED LIST MICRO-ARCHITECTURE

Generally, in hardware linked list designs, several (a fixed number, say N, depending on design requirements) linked lists are maintained in a common memory pool. Size of the memory pool (say S) depends on design requirements such as performance. To maintain N linked lists in a pool of size S, following micro-architecture components are needed:

A. Head-pointer memory

This memory is used to store head-pointers of all the N linked lists. Since the width of the pointers will be $\log_2 S$, size of the head-pointer memory becomes $\log_2 S$ x N (width x depth). Since during the verification we use behavioral models of memory

which use flip-flops to implement memory behavior, this micro-architecture component adds $\log_2$S x N number of flip-flops to the formal model – causing significant formal complexity [2].

B. *Tail-pointer memory*

Like head-pointer memory, this memory is used to store tail-pointers on all the N linked lists. It has same size as head-pointer memory i.e., $\log_2$S x N (width x depth), and adds $\log_2$S x N number of flip-flops to the formal model – causing significant formal complexity [2].

C. *Links memory*

This memory, sometimes known as next-pointer memory or used-pointer memory, stores links for all N linked lists. Size of this memory is $\log_2$S x S (width x depth), and its behavioral model adds $\log_2$S x S number of flip-flops to the formal model – causing significant formal complexity [2].

D. *Data memory*

This memory stores the data for all N linked lists. Width of this memory, say W, is same as width of the incoming data that is to be stored in the linked lists. Depth of this memory is same as pool size i.e., S. Which means that its behavioral model adds W x S number of flip-flops to the formal model – causing significant formal complexity [2].

E. *Free-pointer memory*

This component stores the pointers which are not being used by any of N linked lists at a time. Depending on micro-architecture, sometimes this structure is implemented using a FIFO of size $\log_2$S x S (width x depth), sometimes it is implemented using a bitmask of width S. In case of FIFO, it adds $\log_2$S x S number of flip-flops to the formal model. In case of bitmask, the contribution is S flip-flops. In case of FIFO, the FIFO needs to be initialized to "full" state before writes to linked lists start – because at start all the pointers must be free. Typically, this initialization takes about S clock cycles – adding sequential depth to scenarios and causing formal complexity [4].

F. *Size counters*

To store the size of each linked list, typically there are per linked list counters which store size of a linked list. Width of each counter is $\log_2$(S+1), and there are N such counters – adding $\log_2$(S+1) x N flip flops.

From the concurrency point of view, typically, any of the N linked lists can send writes and reads at each clock cycle (with condition that read must be sent to non-empty linked list). For each write, tail-pointer memory is read, free-pointer memory is read, links memory is written, data memory is written, and tail-pointer memory is written. For each read, head-pointer memory is read, links memory is read, data memory is read, free-pointer memory is written, and head-pointer memory is written. Since writes and reads can come together, the design needs to concurrently handle all these reads and writes to different memories.

III. EFFICIENT CHECKERS FOR LINKED LISTS

The basic checkers to verify linked list designs will be:

A. *Data correctness checker*

For each linked list, the read data is in FIFO order i.e., it does not get corrupted, dropped, duplicated or reordered with respect to the write data.

B. *Forward progress checker*

For each linked list, its size gets updates correctly in a bounded number of cycles after a write or a read; and for each linked list, the read data is sent in a bounded number of cycles after receiving a read.

Apart from this, we might need interface checkers [3] such as valid-ready handshakes at write, read and response interfaces depending on the interface protocol – though we are not going to focus on these further in this paper.

Above checkers can be easily written using FIFO and counter in the testbench, and then can be replicated for each linked list. Though, this approach is not good for formal complexity point of view because we would need N FIFOs and N counters in formal testbench – contributing to formal complexity. Instead, we can use symbolic variables [2] to write these checkers efficiently. A symbolic variable is a testbench variable which can take any random value and remains static. We can define a symbolic linked list number n_{sym} and write above checkers only for linked list number n_{sym}. Because of exhaustive nature of formal, it will pick all the possible values of n_{sym} i.e., from 0 to N-1, and it will verify all the linked lists. This way, we will need only 1 FIFO and 1 counter to verify all the N linked lists. Further, this approach helps us with abstraction models too – which we will discuss in next sections.

IV. COMPLEXITY

Formal verification faces complexity issues where bounded proofs are not able to get sufficient depth – causing coverage holes [3]. If not solved, we cannot utilize the maximum potential of formal and cannot do formal sign-off which is achievable otherwise [2]. Formal complexity is affected by following two parameters – one, design size in number of flips flops; two, sequential depth of scenarios [2]. Due to this, following micro-architecture components will significantly contribute to complexity for formal:

- Head-pointer, tail-pointer, links, data, free-list memories – contributing to complexity due to design size.
- Size counters – contributing to complexity due to design size
- Initialization of free-list memory (if free-list memory is implemented as FIFO) – contributing to complexity due to high sequential depth

V. COMPLEXITY SOLVING METHODS

Before discussing complexity solving methods for linked lists, we need to understand why we need complexity solving. Without complexity solving, the bounded proofs may have coverage holes and may not be verifying all the scenarios. By using complexity solving methods, we want to ensure that bounded proofs reach sufficient depth to see all the scenarios. Let us discuss complexity solving method for each complexity contributing component.

A. Head-pointer memory

In case of memories, a generic approach for complexity solving is adding memory abstraction model. In memory abstraction model, generally, we maintain a row of memory, and the maintained row is chosen symbolically. For all other rows, when they are read, random data is returned - which is fine in most of the cases if we have written checkers for symbolically chosen id such as linked list number. And this is how writing efficient checkers also helps with abstraction models.

We can try adding this memory abstraction model for head-pointer memory where the abstraction model maintains the row corresponding to symbolically chosen linked list number n_{sym}. But it will cause false failures. To understand why, consider this example. Let's say linked list number n_{sym} has following links: $t_{tail} \leftarrow l_3 \leftarrow l_2 \leftarrow l_1 \leftarrow l_0 \leftarrow h_{head}$. A read happens at some non-symbolic linked list $n_{non\text{-}sym}$, causing read to head-pointer memory at row number $n_{non\text{-}sym}$. Since the memory abstraction model can return any random data, it returns l_2, and l_2 further gets written to free-list. Next, a write comes for non-symbolic linked list $n_{non\text{-}sym}$ causing read to free-list, and free-list returns l_2. Now, l_2 has become part of two linked lists at the same time: n_{sym} and $n_{non\text{-}sym}$, causing data corruption.

This issue of data corruption happens because we have a head-pointer memory abstraction model which is too abstract. To refine it, we need to ensure that for a read on $n_{non\text{-}sym}$ row, it does not return any pointer which is part of linked list number n_{sym}. To do it, we would need to create a bitmask B_{sym} of width S that tells which pointers are part of linked list number n_{sym}. This bitmask can be created by probing the reads and writes of free-list memory and mapping them with the writes and reads of linked lists so that we can determine that a read to free-list memory corresponds to write of which linked list number, and a write to free-list memory corresponds to read of which linked list number. If we can determine this, we can create B_{sym} easily – whenever there is a read on free-list due to write on linked list number n_{sym} and let's say free-list returns a pointer l_i, we set bit number l_i of B_{sym} to 1. And whenever there is a write on free-list due to read on linked list number n_{sym} and let's say free-list write data is a pointer l_i, we reset bit number l_i of B_{sym} to 0.

With this bitmask, we can update head-pointer memory abstraction model that – for a read on row number $n_{non\text{-}sym}$, it returns

```
if(rd_free_list for n_sym)
  B_sym[data_out] = 1
if(wr_free_list for n_sym)
  B_sym[data_in] = 0

Assumption: rd_free_list for n_sym → (data_out == maintained data)

Assumption: rd_free_list for n_non-sym → (B_sym[data_out] == 0)
```

any random pointer which is not set in B_{sym}.

This abstraction model adds $S+\log_2 S$ number of flip-flops and reduces $\log_2 S \times N$ number of flip-flops. For this abstraction model to be helpful i.e., to be causing reduction in flip-flops:

$$S + \log_2 S < \log_2 S \times N \quad (1)$$

Or,

$$\frac{S}{\log_2 S} < N-1 \quad (2)$$

If we take an example of S = 4096 and N = 512, LHS of (2) is ~342 and RHS is 511 – which satisfies the requirements.

B. *Tail-pointer memory*

Memory abstraction model for tail-pointer can be exactly same as memory abstraction model for head-pointer memory. In fact, we can use the same B_{sym} for tail-pointer memory too. This changes the requirements of abstraction models to be helpful to:

$$S + 2\log_2 S < 2\log_2 S \times N \quad (3)$$

$$\text{Or,} \quad \frac{S}{2\log_2 S} < N-1 \quad (4)$$

If we take an example of S = 4096 and N = 512, LHS of (4) is ~171 and RHS is 511 – which satisfies the requirements.

C. *Links memory*

Adding a memory abstraction model for links memory is extremely hard. By using B_{sym}, we can ensure that for a read on links memory due to read on linked list number $n_{non\text{-}sym}$, it returns a pointer which is not part of linked list number n_{sym}. Though, for linked list number n_{sym}, we need to ensure that the memory abstraction model maintains all the links of it – which again will require $\log_2 S \times S$ size of memory structure. Overall, it means that we won't be able to create an effective memory abstraction model for it.

D. *Data memory*

Due to similar reasons as given for links memory, it is not possible to add a memory abstraction model for data memory. Though, we can still add an abstraction here. In the data correctness checker, instead of verifying all the W bits of data, if we select a bit (out of W bits) symbolically and verify it – we will be verifying all the W bits of data due to exhaustive nature of formal. Now, we can use this information to add an abstraction model for data memory. Since the checker cares only for the symbolically selected bit, in the abstraction model, we can maintain only that bit for each row of data memory. And for all other

```
if(wr)
  mem[waddr] = data_in[sym_bit]

Assumption: rd → (data_out[sym_bit] == mem[raddr])
```

bits in a row, abstraction model can return any random data. In other words, the abstraction model maintains only a column of data memory.

This abstraction model contributes 1 x S flip flops and reduces W x S flip flops. Overall, this abstraction model helps reducing formal complexity as long as W > 1.

E. *Size counters*

Since the checkers are being written only for linked list number n_{sym}, we can blackbox/cutpoint the remaining counters and

```
Blackbox/cutpoint size counters

cnt_model = rtl.cnt_nxt[n_sym]

Assumption: (rtl.cnt[n_sym] == cnt_model)
```

maintain only one counter for linked list number n_{sym}.

This abstraction model helps reducing formal complexity by reducing $\log_2(S+1) \times (N-1)$ flip flops.

F. *Free-list memory*

Abstraction model for free-list depends on the way it has been implemented in the design. If it has been implemented using bitmask, then we cannot apply an abstraction model for it specifically. Though, in some designs, where writes are backpressured based on the free-list occupancy, we can apply abstraction model for occupancy itself to see the backpressure easily. This abstraction model allows free-list-empty signal being high even if there are pointers available in it. With this abstraction in place, we can verify that the back-pressure logic is able to handle free-list empty situation correctly. Since this is heavily dependent on the implementation, we are not going to talk about it in this paper further.

If free-list has been implemented using a FIFO, then we can add an abstraction model for it. In the abstraction model, we maintain a bitmask of available pointers. Width of the bitmask would be S. When there is a read on free-list and let's say free-list returns a pointer l_i, we reset bit number l_i of bitmask to 0. And whenever there is a write on free-list and let's say free-list

write data is a pointer l_i, we set bit number l_i of bitmask to 1. And on read, we allow to return any pointer whose corresponding bit in bitmask is 0.

```
if(rd)
  bitmask[data_out] = 0
if(wr)
  bitmask[data_in] = 1

Assumption: rd → (bitmask[data_out] == 0)
```

This abstraction model adds S flip flops and reduces $\log_2 S$ x S flip flops – effectively saving ($\log_2 S$-1) x S flip flops.

G. Free-list initialization

In case when free-list is implemented using a FIFO, it needs to be initialized before write operations to linked lists can start. The initialization is done to ensure that all the pointers are in free-list before start of writes. Most commonly, initialization is done using a counter and a state machine. Until initialization is complete, state machine remains in a "pre-initialization" state. In this state, the state machine keeps sending one write to free-list at each clock cycle. Write data is implemented using a counter which goes from 0 to S-1. Once all the S locations of free-list are written, state machine moves from "pre-initialization" state to some other state where writes to linked lists can start. Since during initialization, one entry of free-list is written each clock cycle, it takes minimum S clock cycles to complete the initialization. This adds additional sequential depth to the scenarios where linked list are written and read – causing formal complexity for these scenarios.

To solve this complexity, we can add initial value abstraction (IVA) on state machine and counter that:

- State machine can start from "pre-initialization" or "post-initialization" state.
- If state machine starts from "pre-initialization" state, counter can start from any value between 0 and S-1.
- The free-list memory is initialized for rows from 0 to counter-1 with data equal to row number.

```
IVA on: state, counter, free-list memory

Assumption: 1st_cycle → ((state == PRE_INIT) || (state == POST_INIT))

Assumption: 1st_cycle && (state == PRE_INIT) → (counter < S)

Assumption: 1st_cycle && (i < counter) → (mem[i] == i) 0 ⩽ i ⩽ counter
```

This way, we can ensure that the memory initialization does not always adds sequential depth. Before calling it done, we should notice that we have added initial value abstraction on memory and initialized it from 0 to counter-1 by using an assumption. Since we have added this assumption, we should also verify it by adding a checker that – out of reset, an address which is greater than or equal to counter should get a write on it with write data equal to the address before the initialization completes. By proving this checker, we ensure the correctness of initialization and correctness of the assumption we have added on memory.

H. Free-list memory being empty

One of the interesting and deep in sequential depth scenario is the case when free-list memory becomes empty. It will take minimum S number of writes to reach that state – making it deep in sequential depth.

We have already discussed one way to address this – by adding an abstraction model which allows free-list-empty signal being high even if there are pointers available in it. There is one more way to solve complexity for this case by extending the initial value abstraction beyond initialization i.e., by starting from a state where linked list number n_{sym} has occupied almost all the pointers. Let's understand this by taking an example for linked list number n_{sym} having size equal to S-2: $S\text{-}3_{tail} \leftarrow \ldots \leftarrow 4 \leftarrow 3 \leftarrow 2 \leftarrow 1 \leftarrow 0_{head}$. Let's say that all other linked lists are empty. In this state, only two pointer S-2 and S-1 are free. If free-list is a bitmask, then its only 2 bits at S-2 and S-1 locations must be 1 and all other bits must be 0. If free-list is a FIFO, then its occupancy must be 2 and logical difference between its read-pointer and write-pointer must be 2, and the 2 occupied locations of FIFO must contain pointers S-2 and S-1. In other words at the 1st cycle:

- Only linked list number n_{sym} allowed to have non-zero size.
- $Size_{n_{sym}}$ + free-list occupancy = S.
- Free-list has pointers $Size_{n_{sym}}$ to S-1 available in it. All other pointers are not available.

- Links memory has been initialized such that location i points to location i+1 for i < $Size_{n_{sym}}$-2.
- Row number n_{sym} of tail-pointer memory is initialized to $Size_{n_{sym}}$-1.
- Data memory can be initialized to any random value. All we need to ensure that the reference model (most likely a FIFO in testbench) is also initialized with the same values to ensure no false-failures.

With this abstraction in place, we can easily see the case when free-list becomes empty.

I. Shrunk design

If the design is parameterized, we can reduce the parameters N and S to small numbers to help with the complexity. Though, if we use this approach, we need to be very careful as we would be verifying only the shrunk design and not the actual design. In other words, we need to ensure that by verifying shrunk design, actual design "somehow" also gets verified.

VI. RESULTS

We deployed this methodology on a linked list design with S = 2048 and N = 1024 and free-list as a FIFO. We started the work when the design was almost completely verified using simulation (at a higher-level) and it was clean. We found 2 corner cases bugs for cases when free-list becomes empty. More importantly, we were able to achieve 100% stimuli coverage and 100% bound coverage for the checkers with bounded proof – which shows that checkers were able to hit all the coverage items in their cone of influence. Just to highlight, these results are for the actual design (and not the shrunk design).

TABLE I. RESULTS

Results	
Total checkers (including interface checkers)	14
Bounded pass	5
Unbounded pass	9
Coverage (code coverage)	100%
Coverage (bound coverage)	100%
Bugs found	2

ACKNOWLEDGMENT

Authors would like to acknowledge Pushkar Upadhye and Maulshree Tamhankar from the design team for their quick support during the implementation of above methodology.

REFERENCES

[1] I. Tripathi, A. Saxena, A. Verma, P. Aggarwal. "The Process and Proof for Formal Sign-off A Live Case Study", DVCon 2016.

[2] N. Kim, J. Park, H. Singh, and V. Singhal. "Sign-off with Bounded Formal Verification Proofs", DVCon 2014.

[3] S. Shrivastava, K. Han, A. Tran, C. Agarwal, A. Saxena, A. Mittal, A. Jain, R. Sabbagh. "Formal Verification of Silicon for Software Defined Networking", DVCon 2018.

[4] A. Saxena, R. Baviskar, S. Goel. "Planning Ahead for End-to-end Formal Complexity", CDNLive 2024.