Formal Verification of Arbiters using Property Strengthening and Underapproximations

Gadiel Auerbach Fady Copty
IBM Haifa Research Laboratory, Haifa, Israel. e-mail: *gadiel,fadyc@il.ibm.com*
Viresh Paruthi
IBM Systems and Technology Group, Austin, TX, USA. e-mail: *vparuthi@us.ibm.com*

***Abstract*—Arbiters are commonly used components in electronic systems to control access to shared resources. In this paper, we describe a novel method to check starvation in random priority-based arbiters. Typical implementations of random priority-based arbiters use pseudo-random number generators such as linear feedback shift registers (LFSRs) which makes them sequentially deep precluding a direct analysis of the design. The proposed technique checks a stronger bounded-starvation property; if the stronger property fails, we use the counter-example to construct an underapproximation abstraction. We next check the original property on the abstraction to check for its validity. We have found the approach to be a very effective bug hunting technique to reveal starvation issues in LFSR-based arbiters. We describe its successful application on formal verification of arbiters on a commercial processor design.**

I. Introduction

Arbiters [4] are widely used in electronic systems such as microprocessors and interconnects. Arbiters restrict access to shared resources when the number of requests exceeds the maximum number of requests that can be satisfied concurrently. For example, an arbiter that regulates access to a bus selects which requestors would be granted access to the bus if there are more concurrent requests than the bus can handle. Arbiters use various arbitration schemes in the form of a priority function to serialize access to the shared resource by the requestors. The priority function decides which requestor to grant next. Examples of priority functions include round robin (rotate priority amongst requestors), queue-based (first-in first-out), or random priority (select next requestor randomly).

Random priority-based arbiters [8] have been gaining in popularity because of their high potential for fair arbitration, unlike other techniques such as round robin or queue-based which can be unfair because of their fixed order of arbitration. This arbitration scheme allows any request to have the highest priority at random. A random priority-based arbiter uses a pseudo-random number generator to select or influence the selection of the next requestor. A common implementation of such arbiters uses a Linear Feedback Shift Register (LFSR) [7] to generate a pseudo-random sequence of numbers. An LFSR is a cyclic shift register whose current state is a linear function of its previous state, and it generates a sequence of numbers which is statistically similar to a truly-random sequence. In this paper we focus on formal verification of such LFSR-based random priority arbiters.

The main concern in verification of an arbiter is checking for starvation. Starvation is a special case of liveness properties,

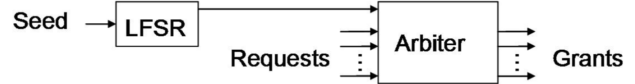

Figure 1. LFSR-based arbiter

in which any request must have a grant eventually. Liveness properties are often computationally hard to verify even on medium-sized designs. To alleviate this, it is common to check for starvation by replacing liveness properties with bounded properties – “request will be granted within N cycles”, for some constant N. If a bounded property passes, it implies the correctness of the original liveness property. Even so, the sheer size of LFSR-based industrial arbiters may preclude an exhaustive analysis of the bounded property.

We describe a method to uncover bugs leading to long latencies before requestors are granted in such complex arbiters. If the bounded property fails, we study the counter-example and attempt to either fix the problem by increasing the bound, or to use the information from the counter-example to underapproximate the original design. The concepts presented in this paper can be easily generalized to other schemes (besides LFSRs) to implement a random priority function. The presented technique can, in fact, be generalized to model checking of general-purpose systems, and we briefly present such a generalization.

II. LFSR-based arbiters

An LFSR-based arbiter grants access to a pending request based on the random number generated by the LFSR at any given point in time. Figure 1 shows a schema of an LFSR-based arbiter. An LFSR of length N generates a deterministic cyclic sequence whose period is $2^N - 1$, where all numbers from 1 to $2^N - 1$ are visited. The initial value of an LFSR is called the seed, and the sequence of numbers generated by the LFSR is completely determined by the value of its seed. An LFSR of length N may be used to arbitrate between M requestors, where $M \ll 2^N$, by sampling a subset $log(M)$ bits of the LFSR to select the next request to be granted. Such a scheme helps to amortize the cost of implementing an LFSR in hardware by way of the same LFSR serving multiple arbiters with different tap points. E.g., N may be 16, while M is 8 requiring 3-bits of the 16-bits of the LFSR to be tapped.

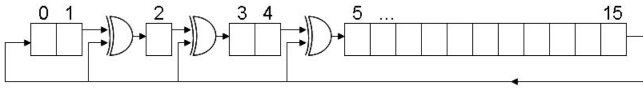

Figure 2. 16-bit LFSR

Figure 2 depicts a 16-bit LFSR from one of our case studies, the I_arbiter. The register shifts bits from left to right with some bits XORed with the most significant bit. The LFSR seed is configurable, and may be assigned any value between 1 and $2^{16} - 1 = 65535$. Formal verification environments typically assign a non-deterministic value to the seed.

III. Formal verification of LFSR-based arbiters

Verification of arbiters entails checking for starvation, which may be formulated as a liveness property. E.g., the following PSL [6] property specifies that whenever signal `request` is asserted, signal `grant` is asserted some time in the future.

```
always request -> eventually! (grant)
```

A counter-example for such a property is a trace showing a path leading to an infinite loop. In an LFSR-based arbiter, this constitutes a cycling through of all the valuations of the LFSR. The LFSR minimal loop length is $2^N - 1$, thus any loop showing a counter-example of the liveness property must be at least of that length. Hence, finding a trace for such a property of an LFSR-based arbiter is very hard. An easier yet more useful alternative to the above correctness property is to check for a request to be granted within a specified number of cycles, determined by the arbiter specification. In other words, we check to see if the request is granted within k cycles [8], [5]. In addition to verifying that a request is granted, such a formulation gives insights into the performance aspects of the arbiter, which is quite useful given the critical role arbiters play in the overall performance of electronic systems. The following property expresses a bounded-starvation condition.

```
always request-> next_e[1..k](grant)
```

Exhaustive verification of above properties to guarantee lack of bugs on is becoming increasingly challenging, if not impossible, for arbiters in real-world systems due to their sheer size and complexity. This calls for bug hunting methods to detect as many bugs as possible using scalable underapproximate techniques and (semi-) formal analysis. Such methods are more practical and provide concrete traces, rather than a suspicious bounded pass due to suspect abstractions.

Related work

As stated above, typical approaches to verify arbiters check for eventual grant of resources to the requests without much attention to performance aspects. Krishnan et. al. [8] studied starvation and performance of random priority-based arbiters extensively. They proposed a three-step verification process for computing an upper bound on the request-to-grant delay. In the first step they compute the maximum length Complete Random Sequence (CRS) comprising all random numbers (in the context of the sampled bits) the LFSR can assume. Next they compute the maximum number of CRSes needed for a request to be granted by the arbiter standalone with replacing the LFSR with a random-number generator. In the third step, the two values computed are combined to give the worst-case request-to-grant delay in clock cycles. A drawback of this method is the decoupling of the LFSR from the arbiter in the second step; a CRS can complete without being sampled by the arbiter. This produces a theoretical worst-case request-to-grant delay yielding very high bounds at times, much higher than the bounds stated in the model specification to be useful. Moreover, the trace produced by this technique is not representative of the overall system comprising the LFSR and the arbiter.

Our proposed technique compliments the above solutions by providing an effective bug hunting method for the actual LFSR-based arbiter, without any simplification thereof which may render the treatment (results) removed from the real logic. The effectiveness of the method has been proven on highly complex arbitration systems where it was leveraged to find real bugs. The method dynamically chooses between property strengthening and underapproximations in order to find a failure faster. The method can be easily generalized to create property-based underapproximations.

IV. Bug hunting in LFSR-based arbiters

The complexity of property checking is a function of the property and the design-under-test (DUT). Our bug hunting approach considers both the property and the DUT. In this section we describe how we construct easier-to-check underapproximate abstractions of LFSR-based arbiters.

Underapproximation and overapproximation techniques are commonly used to falsify properties or prove their correctness [3]. An abstract system is easier to check than the concrete system because it has fewer states and fewer transitions. Since our focus is bug hunting of safety properties we leverage underapproximations to obtain traces falsifying the property, which are then validated on the concrete/original model.

The seed of an N-bit LFSR may range between 1 and $2^N - 1$. The seed fully determines the LFSR sequence, so a run of the arbiter is based on one of $2^N - 1$ possible seeds. To underapproximate the arbiter we fix the LFSR seed by assigning it a constant N-bit number. A fixed-seed arbiter underapproximates the nondeterministic-seed arbiter as every run of a fixed-seed arbiter corresponds to a single LFSR sequence. If a bounded-starvation property fails in a fixed-seed arbiter then it definitely fails in the nondeterministic-seed arbiter; additionally, a counter-example that demonstrates a fail of a safety property in a fixed-seed arbiter is valid in the nondeterministic-seed arbiter. If a bounded-starvation property holds in a fixed-seed arbiter we cannot ascertain if it holds in the concrete system.

Falsification of a k-cycle-starvation property in an N-bit LFSR arbiter requires checking runs of depth k in a model that allows $2^N - 1$ possible LFSR sequences. Our method addresses the inherent hardness by alternating checking easier-to-check properties on the original system, and checking the original property on abstract systems. We iteratively check starvation with lesser bounds on the original system, and starvation with

the original bound on fixed-seed arbiters. We use property strengthening to seek interesting seeds that generate sequences that are likely to cause long starvation.

We define the following properties that express lower request-to-grant delays

$p_j \doteq$ `request -> next_e [1..j] (grant)`

for $1 \leqq j < k$. It is obvious that checking any of the properties p_j can be done in a shorter period of time than the original property. Clearly, every run that starves a request for k cycles starts with a starvation of j cycles, but a starvation of j cycles does not necessarily end with a starvation of k cycles. If a property p_j fails in the concrete system and a counter-example is generated, we underapproximate the arbiter by restricting it to the very same LFSR sequence that the counter-example reveals. Since LFSR sequences are determined by their seed it is enough to confine the arbiter's non-deterministic seed to the same seed that is exposed by the counter-example. Checking the fixed-seed arbiter is easier and likely to uncover a k-cycle long starvation.

Our method is outlined in Algorithm 1. We denote the original nondeterministic-seed LFSR arbiter by M, the maximal number of cycles allowed between a request and a grant as determined by the specification by k, and for some constant number c, we denote by $M[seed \leftarrow c]$ the arbiter M whose seed is the constant number c.

Algorithm 1 Checking bounded starvation on LFSR-based arbiters

1) check $M \overset{?}{\models} p$
2) **if** pass or fail **then return** result
3) $j_{min} \leftarrow 1; j_{max} \leftarrow k$
4) **while** ($j_{min} \lneqq j_{max}$) **do**
   a) $j \leftarrow \lfloor \frac{j_{min}+j_{max}}{2} \rfloor$
   b) check $M \overset{?}{\models} p_j$
   c) **if** pass **then return** "pass"
   d) **if** timeout **then** $j_{max} \leftarrow j$
   e) **if** fail **then**
      i) $M_j \leftarrow M[seed \leftarrow seed_j]$; $j_{min} \leftarrow j$;
      ii) check $M_j \overset{?}{\models} p$
         A) **if** fail **then return** "fail"

The algorithm checks bounded starvation with different bounds and creates underapproximations of the original arbiter by initializing it with different seeds. We iteratively check property p_j with arriving at the next value of j using a binary search. If checking of a bounded-starvation property p_j times out, we next check another bounded-starvation property with a lower bound. If a property p_j fails, we extract the LFSR seed from the counter-example, denoted by $seed_j$. Next we restrict the arbiter's seed to $seed_j$, and check if the original property fails in the fixed-seed arbiter. If the property does not fail we narrow the seed space by checking a weaker property with a higher bound.

The algorithm halts after $\log(k)$ steps at the most. Let us examine an extreme case where all runs of the strengthened properties on the concrete model, $M \models p_j$, time out. This indicates that the arbiter is extremely complex and beyond the capabilities of our formal-verification tools. We note that the method is an effective bug hunting heuristic, but does not guarantee a bug free design, nor does it cover all LFSR seeds.

V. Bug Hunting Method – A Generalization

We generalize the presented heuristic to general purpose model checking. The rationale is straightforward – check strengthened properties on the original model to aid in finding an efficient underapproximation for bug hunting on the original model. If any of the strengthened properties pass on the original model, it implies that the original property passes as well. If it fails then, heuristically, it has some information leading to a fail of the original property. This information can be extracted, and used to guide the search for a failure on the original property. This is achieved by defining an underapproximation of the model and checking for the validity of the property on it.

Intuitively, a safety property asserts that something bad never happens, while a strengthened property asserts that something "not-as-bad" never happens. Formally, for two properties p and q we say that property p is *stronger* than property q if $p \rightarrow q$. Consequently, given system M and two properties p and q such that p is stronger than q, we have $M \models p \rightarrow M \models q$, i.e., if p holds in M then q holds in M.

Falsification of a strengthened property tends to be easier than falsification of the original property because it defines more bad states in the system. If falsification of the original property is infeasible then we check a strengthened version of the property. If the strengthened property fails, we restrict the concrete system to the valuations provided by the obtained counter-example, and see if the original property fails.

It is not easy to determine how to strengthen a property in a useful manner. Hence, we restrict the discussion to a subset of properties whose strengthened versions enable an efficient and exhaustive search. A straightforward example for such properties is PSL parameterized properties that have a single parameter that serves as a sequence consecutive-repetition operator or as a bound of the next_e or next_a families of operators (formal definitions can be found in [1]). These widely-used operators are similar to the next_e operator used in our test case, and the practice of binary search over a bounded range of integers readily applies to them.

VI. Experimental Results

The bug hunting method described in section IV has been used to verify several random priority-based arbiters used in an interconnect unit, and a router of a complex commercial processor. Table I shows the experimental results on 3 such industrial designs that use different types of random priority-based arbiters, and different LFSR sizes to generate pseudo-random numbers. The first arbiter, referred to as C_arbiter, is a command arbiter using a 32-bit LFSR. It arbitrates 27 requestors going to a single target. Its specification states the starvation bound to be 600 cycles. It uses a compound priority scheme combining LFSR-based arbitration and round robin to combinatorially compute the next granted requestor.

Design	*Random seed run time (h:m)*	*Fixed seed run time (h:m)*	*Vars before Redn*	*Gates before Redn*	*Vars after Redn*	*Gates after Redn*
C_arbiter	48:00 (Timeout)	8:56	2361	90397	812	7883
I_router	48:00 (Timeout)	21:09	104575	4223285	34070	1413519
I_arbiter	21:34	19:50	104575	4223285	30766	876328

Table I
RUN TIMES AND MEMORY USAGE FOR DIFFERENT ARBITERS

The second design, referred to as I_router, is a router of 56 requestors to 56 targets. The router is a more complex case of arbitration. It cannot starve an input from getting a request, and it cannot block an output from receiving a request. This router has a 16-bit LFSR, and it uses three of its bits for arbitration. It is a very large design with hundreds of thousands of variables (inputs and Flip-Flops) with multiple arbitration stages. The third arbiter, I_arbiter, is a simpler case of this router, with only one target available, thus checking arbitration only. The specification of I_router and I_arbiter requires a starvation bound of 1000 cycles.

All experiments were run on a 2x2.4GHz AMD dual core processor with 8 GB RAM memory, using IBM's RuleBase PE [2] and SixthSense [9] state-of-the-art industrial formal verification tools. The problem size is in term of gates and variables as reported by the RuleBase PE tool, shown before and after running RuleBase PE automatic model-size reductions. *Vars* denotes the numbers of registers and inputs.

For each of the designs we first applied the CRS technique [8]. The results yielded request-to-grant bounds higher than the starvation bounds in the specification. E.g., for the router arbiter it showed that the max length of CRS is 95 cycles; and we found that the request-to-grant delay is at least 50 CRSes – while trying to find a higher bound of 100, the tool timed out, implying a best case request-to-grant upper bound to be at least 4750 cycles.

Table I shows the run time of runs of the original property on fixed-seed arbiters that yielded traces (the last step in Algorithm 1). The various runs to compute an initial LFSR seed took anywhere from few minutes to 4 hours. We used parallel capabilities of our toolset to run a large number of rules with different starvation bounds, with a total run-time of 8 hours. The highest bounds on which the properties p_j failed were 375 for the C_arbiter and 687 for the I_arbiter. We gathered all LFSR seed values from the failing traces, seeded the LFSR of the original design with those, and ran the original formula. For benchmark purposes, the results above show the run time of RuleBase PE without using the parallel feature.

The verification timed out on the nondeterministic-seed runs of the C_arbiter, while a specification violation with a fixed seed was found in 9 hours. For the I_router design, the nondeterministic-seed runs timed out as well, while a trace for a fixed seed was obtained after 21 hours. As for the I_arbiter, the nondeterministic-seed finished in 21-1/2 hours while the fixed seed finished in 20 hours. In the I_router and I_arbiter designs the trace was found after the first run of algorithm 1, while on the C_arbiter the algorithm ran more than once and timeout increased for the run of stronger properties on the original model.

Clearly the fixed seed method shows a significant advantage on the more complex designs. It was able get past the huge complexity barrier of these designs. Note that even if the nondeterministic-seed runs were to finish easily, the initial state of the LFSR from these runs can be used as a seed for future runs that try to falsify proposed fixes. Another interesting fact was that the initial LFSR seed for the I_router and I_arbiter traces was different. In addition to finding the bounded starvation traces, our method was able to give us a large number of interesting traces which provided insights into the relationship between the LFSR and the arbiter.

VII. Conclusion and Future Work

We presented an effective method for computing smart property-based underapproximations. The technique dynamically converges on underapproximations which yield useful results in the form of bugs or interesting insights into the workings of the logic. This method has been successfully applied to LFSR-based arbiters and provided results which otherwise would not have been obtained with other techniques.

The described approach can be further generalized to other types of properties. Other directions include developing more general ways to construct underapproximations from counter-examples. The search for underapproximations can be improved by considering additinal seeds provided by the underlying decision procedure. The method can be enhanced further to be a proof-oriented approach by extracting reasons for pass results of the strengthened properties from the solving engines.

Acknowledgments

The authors would like to thank Alexander Ivrii and Hana Chockler for their helpful comments.

References

[1] *IEEE Standard for Property Specification Language (PSL). IEEE Std 1850*. 2010.
[2] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at ibm. *Formal Methods in System Design*, 22(2):101–108, 2003.
[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
[4] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, San Francisco, 2003.
[5] N. Dershowitz, D.N. Jayasimha, and S. Park. Bounded Fairness. *Lecture Notes in Computer Science*, 2772:304–317, 2004.
[6] C. Eisner and D. Fisman. *A Practical Introduction to PSL*. Integrated Circuits and Systems. Springer-Verlag, 2006.
[7] P. Horowitz and W. Hill. *The art of electronics*. Cambridge University Press, 2nd edition, 1989.
[8] K. Kailas, V. Paruthi, and B. Monwai. Formal verification of correctness and performance of random priority-based arbiters. In *FMCAD*, 2009.
[9] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable automated verification via expert-system guided transformations. In *FMCAD*, pages 159–173, 2004.