Formal Verification of Correctness and Performance of Random Priority-based Arbiters

Krishnan Kailas
IBM T. J. Watson Research Center
Yorktown Heights, NY
kailas@us.ibm.com

Viresh Paruthi Brian Monwai *
IBM Systems & Technology Group
Austin, TX
vparuthi@us.ibm.com, bmonwai@u.washington.edu

***Abstract*—Arbiters play a critical role in the performance of electronic systems. In this paper, we describe a novel method to formally verify correctness and performance of random priority-based arbiters. We define a property of random number sequences, called Complete Random Sequence (CRS), to characterize bounded fairness properties of random number generators and random priority-based arbiters. We propose a three step verification method utilizing the notion of CRS to establish deadlock-free operation of the arbiters, and to accurately quantify the request-to-grant delays. The proposed verification method may additionally be leveraged to tune systems composed of random priority-based arbiters and pseudo-random number generators, such as linear feedback shift registers (LFSRs), for optimal performance. We have successfully applied the approach to verify a host of cache arbiters and interconnection network controllers of commercial microprocessors.**

I. Introduction

Arbiters [1] are used extensively in electronic systems such as microprocessors, interconnection networks and other peripheral chips. The primary function of an arbiter is to restrict access requests to a shared resource, such as cache directories and buses, when the number of requests exceeds the maximum number that can be satisfied concurrently. A variety of arbitration schemes are employed by arbiters to serialize the access requests based on assigning a priority to the input requests, such that the highest priority requests are granted access to the shared resources before other pending or concurrent low priority requests.

Several priority functions [1] such as fixed-priority (certain requests always have higher priority than others), round-robin priority (strict rotation of priority assignment), priority assignment based on request arrival time (first-in first-out or least recently used), and random priority (any request can have the highest priority, at random) are commonly used by arbiters. Random-priority based arbiters are often preferred in many parts of large chip designs because they provide potentially fair arbitration while using relatively less logic and power. The latter is a by-product of sharing a few on-chip pseudo-random number generators across a number of arbiters with each arbiter tapping a subset of the bits of the pseudo-random generator to assign priorities to input requests at "random".

Regardless of the arbitration scheme used, it is important to specify and verify the desired fairness properties of the arbitration scheme. Fairness ensures that all requests are granted within a finite amount of time (based on the arbiter design specifications) and no requests are forced to starve. Formal verification techniques may be used to verify that the arbitration logic is starvation-free. In addition, formal verification may be used to prove that the arbiter design adheres to the design performance requirements, such as request-to-grant delay bounds.

In this paper we focus on the formal verification of random priority-based arbiters. Our focus will be on verifying deadlock/ starvation free operation of the arbiters. Other interesting safety properties can be verified easily with suitable methods. For eg., mutual exclusion of grants can be verified by using a counter for counting the number of grants in each cycle, and using traditional bounded-model checking techniques to check for the number of grants (counter value) never exceeding the maximum number of concurrent grants allowed by the design specifications. The ideas presented in the paper may be generalized to other types of arbiters (eg. round robin, two-level random priority-based) by suitably quantifying the fairness aspects of the respective arbitration schemes.

Typical formal verification approaches used for verifying arbiters leverage temporal logics [2] to specify liveness properties [3], which are then evaluated by an underlying decision procedure to check for the presence of deadlocks/starvation. The focus of such prior techniques is to prove that deadlocks/starvation cannot occur in an infinite execution of the machine. The decision procedure analyzes the reachable states, and looks for loops in the state transition graph. However, such a proof of liveness simply guarantees that a grant will be issued eventually, and does not provide any insights into the upper bound on the number of clock cycles between a request and a grant. The grant can be delayed for an unacceptable number of cycles, making such a proof less valuable from the standpoint of the performance requirements/specification of the design. Hence, there is a need for an improved methodology to (formally) verify arbitration logics inclusive of the performance/quality-of-service aspects of the logics.

We present such a verification methodology in this paper. We articulate the absence of deadlocks as a bounded liveness check, which has several advantages. It enables precisely computing the bounds on the request-to-grant delay, giving insights into the performance of the logic, and greatly im-

* Brian Monwai is currently at the University of Washington, Seattle.

proves computational efficiency as safety property checking is easier to evaluate algorithmically compared to liveness. It may be noted that while it is common to cast a liveness check as a bounded safety check, our proposed methodology uniquely quantifies the fairness and request-to-grant delay characteristics of the arbitration logic.

The main advantages of our verification scheme are as follows:

- The method effectively decouples the fairness logic from the actual arbitration logic, allowing checking the bounded fairness properties of each independently. Hence, the proposed scheme scales well to verification of large arbiters.
- The scheme provides a method to quantify the fairness properties of pseudo-random number generators [4] and arbiters that use random priority-based arbitration schemes.
- The approach improves upon prior art verification schemes, which guarantee deadlock-free operation of arbiters, by accurately computing the request-to-grant delay of the arbiters as well, thus enabling verification of performance aspects of the design.
- The technique can be applied to the verification of RTL directly ensuring correctness of the real logic, and does not require building any specialized models.

Organization. The rest of the paper is organized as follows. In the next section we briefly review the related work and highlight the novel aspects of our proposed solution. Section III provides a general background about random priority-based arbitration scheme. In section IV, we discuss the issues in specifying a fairness property for random number sequences, and introduce the notion of Complete Random Sequence. We describe the details of the proposed formal verification method in section V, and discuss the results in section VI, followed by some conclusions.

II. Related Work

Arbiters are routinely verified using formal verification techniques (eg. [5], [6]) such as Symbolic Model Checking [7], [8]. Typical approaches check for starvation, specified as a temporal logic liveness formula, which checks for eventual grant of resources to the requesters in an infinite execution of the machine - a property which is impossible to verify using prevalent underapproximate techniques such as simulation. Liveness properties are easy to formulate and verify against the logic, which has helped establish formal verification as the key to verifying arbiters. As stated above, such techniques do not give much insight into the performance of the arbitration logic, such as whether the requests are granted access to shared resources within a specified number of cycles.

Checking for unbounded liveness though poses a challenge due to higher computational complexity than other classes of properties. Hence, practical approaches cast the unbounded liveness check as a bounded check for liveness [9], [10] and bounded fairness [11]. This necessitates the re-tooling of the bounded liveness check to take into account fairness constraints imposed on the logic. While such an approach does allow for verification of liveness using a wider range of decision procedures (such as SAT) and gives some insights into the performance, it still is somewhat of an approximation as the fairness has to be "artificially" built into the property, and not really checked for against the actual fairness logic. Alternately, the liveness property may be converted into a safety property using transformations such as [12], [13]. This enables leveraging a rich set of verification algorithms (eg. Transformation-based [14]) to alleviate the capacity problems somewhat [15].

Our proposed ideas go beyond the above techniques by way of precisely characterizing the performance of the arbiter and the fairness logic, which can then be compared against the design specification for conformance, or used to ascertain system performance. Moreover, the insights we provide can be used to optimize the fairness logic, eg. fine tune the LFSR logic to generate all possible unique random numbers within a specified number of cycles, or select the right subset of bits to tap for optimal performance. We decompose the verification task into checks on the fairness logic and the arbitration logic separately which further addresses computational complexity of the task.

III. Random priority-based arbiters

Random priority-based arbiters are commonly used for granting a subset of several concurrent read and write requests to access a shared resource such as a cache directory or a shared bus in every cycle. In random priority-based arbitration, any request can become the highest priority request at random. For example, as shown in Figure 1, request with ID i gets its turn at time t when the value of random number $r(t) = f(i)$, where $f(i)$ is a function of i. The random numbers are usually generated using a pseudo-random number generator such as a Linear Feedback Shift Registers (LFSR) [4]. The goal of this arbitration scheme is to provide unbiased service to all requests.

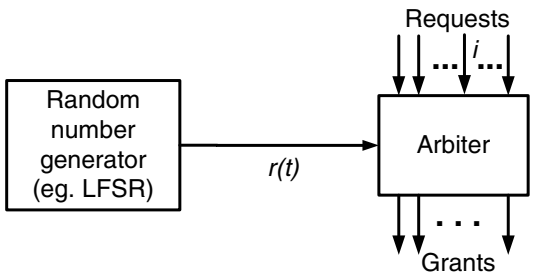

Fig. 1. Random priority-based arbiter.

In such random priority-based arbiters, one of the pending requests is granted access to the shared resource based on the random number generated by the LFSR logic in the current cycle. A request will be starved (i.e., its grant will be delayed for a long time) if the specific random number $r(t)$ that allows a request i to be granted is not generated by the LFSR logic for a long time. Clearly, delaying a request beyond a certain

number of cycles can have serious performance impact, for example, when such an arbiter is used in a cache directory access control logic. Therefore, in real applications, it is not sufficient to prove that the random number generator produces a random number sequence containing all possible values because it only proves that any given random number will be generated *eventually* (i.e., any given request will be granted *eventually*). We additionally need to define fairness properties for the random number generator in order to reason about and prove the request-to-grant delay bounds of the arbiter.

In the next section, we explain the key insights that led to our formal verification method.

IV. Specifying bounded fairness property for random number sequences

True random numbers are hard to generate. This has led to the creation of a host of pseudo-random number generators in use today with different characteristics. Random number generators, and the sequence of random numbers generated by them, are characterized mainly by a few key properties such as predictability and distribution. Predictability measures the randomness of the pseudo-random generator by characterizing occurrences (or lack thereof) of repeating sequences of random numbers generated. Distribution measures the frequency of occurrence of each number in an infinitely long sequence of random numbers generated; it is desirable to have a uniform frequency distribution, i.e., each random number must have the same frequency. However, the property of greatest importance in ensuring that a random priority-based arbiter will grant a request within an acceptable finite time interval is a different one – this fairness property must ensure that any unique random number will be generated within a fixed time interval. We use the notion of *Complete Random Sequence* to characterize the fairness property of a random number sequence, and the logic (such as LFSR) used to generate such a number sequence.

A. Complete Random Sequence

We define a Complete Random Sequence (CRS) in a random number sequence as a contiguous sequence of random numbers that has *all* the possible unique random numbers at least once. There are 2^N unique numbers in the output of a random number generator (e.g. LFSR) that can generate an N bit random number. Therefore, if the random number generator can generate one random number in every cycle, the length of the shortest CRS will be 2^N cycles. The longest CRS generated by a pseudo-random number generator can be infinitely[1] long. In other words, a random number generator may take a variable number of cycles from 2^N to infinity to generate a CRS.

V. Formal verification method

In the following we present a formal verification method that uses CRS (defined above) to specify the bounded fairness properties of random priority-based arbiters, and to accurately quantify and verify the request-to-grant delays.

[1]The CRS can be infinitely long if one (or more) of the 2^N random values is missing from the shift register sequences [16] used by the arbiter.

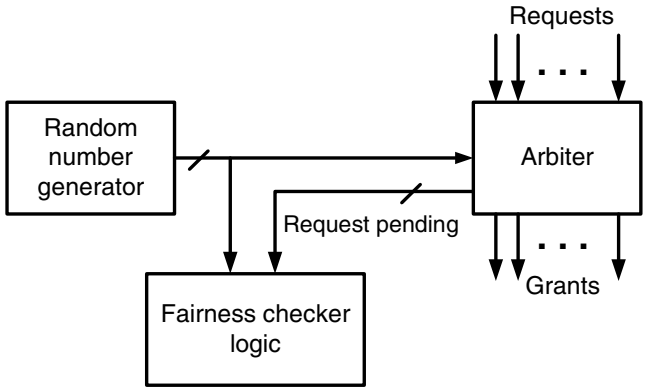

Fig. 2. Block schematic of the testbench with arbiter and fairness checker.

We use a 3 step process to quantify and verify the fairness properties of the random priority-based arbiter design under verification. In the first step, the request-to-grant delay bounds of the arbiter are determined in terms of CRS (instead of in cycles). In the second step, the lower and upper bound (in cycles) of the length of CRS generated by the random number generator (used to impose fairness by the arbiter, eg. LFSR) is determined. In the third step, the lower and upper bound values determined in step 2 are combined to compute the request-to-grant delay bounds (in cycles) of the arbitration logic as a whole. This is an accurate characterization of the performance of the arbiter, and can be used to verify that the design meets the specification. Each one of the above steps can potentially uncover a number of bugs in the logic design. A description of each of the 3 steps follows in the sections below.

A. Determining request-to-grant delays in CRS

In our first verification step, we quantify the upper bound on the request-to-grant delay, in terms of the number of CRSes in the random input number sequence, in the time interval from the issue of a request until it is granted. The idea is to prove a bounded liveness property – i.e. a request will be granted within a bounded period of time if the random number sequence meets certain fairness constraints.

This may be done by detecting the CRSes in the random number sequence and determining how many CRSes are needed to grant a request using a testbench consisting of a fairness checker logic, the arbiter and a set of non-deterministic input bits replacing the random number generator output signals as shown in Figure 2. Typically a subset of the outputs (via tap points on registers) of the random number generator is used by the arbiter; we replace these register outputs with non-deterministic input signals with the same timing characteristics. Note that we are not including the actual logic implementing the random number generator, such as an LFSR logic containing several registers, because in this step we are trying to prove a property of the arbiter independent of the characteristics of the particular random number generator

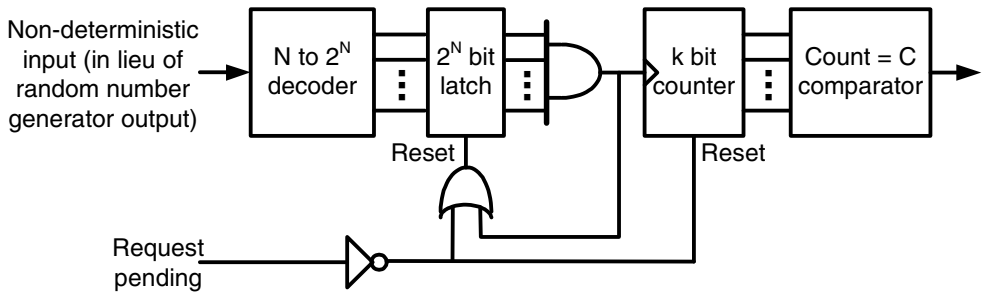

Fig. 3. Logic for checking fairness requirements of one input request of random priority arbiter.

driving the arbiter.

Figure 3 shows the fairness checker logic for one request. It uses a 2^N bit latch to keep track of the occurrences of the 2^N unique numbers in the random number sequence generated by a N bit non-deterministic input source (in lieu of the actual random number generator). A k bit counter is used to count the number of CRSes in the random number sequence while a request is pending (i.e., the time when a request is issued until the request is granted). The latch and the counter will be reset until a request becomes active. Whenever a CRS is detected in the input sequence, all the bits in the latch will be set, causing the output of the AND gate to become active. The k bit counter is incremented and the latch is reset after each time a CRS is detected. When a request is granted the *Request pending* signal becomes inactive. Thus, the k bit counter keeps track of the number of CRSes in the random number sequence while the request is pending. The size of the counter, k, may be selected based on the desired worst-case request-to-grant delay as required by the design specifications of the arbiter.

The k bit counter output values are compared with a constant value C using a comparator. Constant value C represents the request-to-grant delay in terms of the number of CRSes that the model checker will attempt to verify against in each one of the iterative proof steps. The initial value of constant C may be set to any value from 0 to $(2^k - 1)$ by the fairness checker logic. Regardless of the initial value of C selected, our verification method systematically searches the possible range of values of C in an iterative manner to find the largest value of C for which the following property holds for all possible sequences that can be generated by a random number generator:

Request pending $\wedge$ *(number_of_CRSes = C) = TRUE,*

where $C < (2^k - 1)$. The largest value of C can be determined in at most k verification runs using a binary search [17] approach for selecting the values of C in each run.

B. Determining the length of CRS

The second verification step accurately quantifies the upper and lower bounds of the length (in number of clock cycles) of CRSes generated by a specific implementation of random number generator, such as an LFSR, used by the arbiter. The basic idea is to nondeterministically sample the random number sequence for a fixed number of cycles L and varying L in each proof step until the sampled random number sequence has exactly one CRS. The maximum and minimum values of such a sampling window of length L gives the upper and lower bounds of the length of the CRSes, respectively, that can be generated by the random number generator.

A sampling window gating signal can be generated using a fixed width pulse generator comprising a counter that can count up to a fixed number of cycles (say L cycles) and triggered by a random start window signal as shown in Figure 4. Such a sampling window gating signal can be used to sample the random number sequence produced by a random number generator at random instances for a fixed time interval as shown in Figure 5. The sampled random number sequence (the output of 2-input AND gate in Figure 5) is monitored to detect whether the sequence contains one CRS in it using a logic similar to the fairness checking logic described in the last verification step (see section V-A).

Fairness checker logic allows using formal verification techniques to accurately determine the upper and lower bounds of the length of CRSes in the random number sequence by varying the fixed sampling window size L and proving that the fixed sampling window contains exactly one CRS. For a given fixed length L of the sampling window, value of the output signal "CRS found" of fairness checker logic shown in Figure 5 at the end of the sampling window may be used to prove that the random number generator may generate a CRS of length L. The property (absence of exactly *one* CRS in the random number generator output)

$$CRSfound = FALSE$$

holds only when $L < L_{min}$ and $L > L_{max}$, where L_{min} is the length of shortest CRS and L_{max} is the length of longest CRS. In other words, the length of the longest CRS can be determined by observing the values of $L > L_{min}$ for which the property holds. The length of the shortest CRS can be determined as the smallest value of L for which the property

$$CRSfound = TRUE$$

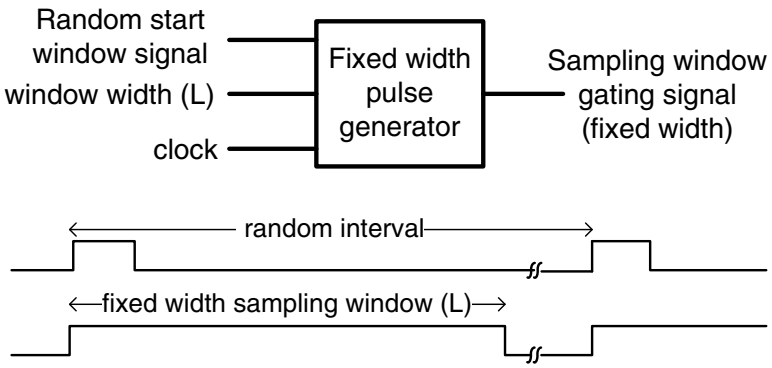

Fig. 4. Fixed width pulses with random periodicity.

holds for all possible window start cycles. The iterative process of determining L_{min} begins with the fairness checker logic initializing the sampling window gating signal to an initial fixed width $L = 2^N$. The value of L is incremented in each proof step until the above property holds.

The upper bound of the length of the CRS L_{max} can be determined in a similar way. The length of the sampling window is varied in each proof step such that the smallest value of the length of CRS $L > L_{min}$ for which the property

$$CRSfound = FALSE$$

holds for all possible random window start cycles. The value of L determined is $1 + L_{max}$.

C. Computing request-to-grant delay bounds

In the third step, the results of the first step and second step are combined to determine the worst case request-to-grant delay bounds in terms of the number of clock cycles of the entire arbitration logic, including the arbitration logic and the pseudo-random number generator logic driving the arbiter. In this third step, the upper and lower bounds of the length of the CRS computed in the second verification step may each be combined with the result (largest value of C) computed in the first step to obtain the worst-case request-to-grant delay bounds in cycles of the arbiter as:

(largest value of C) $\times$ (length of CRS in cycles)

The results obtained can be validated against the design specification to make sure that the arbiter conforms to the performance requirements, and no requests are starved for more than a fixed number of cycles.

Discussion The length of a CRS is determined by the last missing-random-number in a random number sequence regardless of the technique used for generating random numbers (such as LFSR logic). In the worst-case scenario, this last missing-random-number is the one that would cause a request to get a grant. Therefore, if the request arrives at the beginning of a CRS, it has to wait until such a last missing-random-number shows up to provide a grant, regardless of the way in which this last missing-random-number is delayed by the LFSR logic. This is the reason why the length of the longest CRS determines the worst-case request-to-grant delay.

Even though multiplying min/max length of a CRS with the number of CRSes can provide the range of request-to-grant delays corresponding to the best- and worst-case scenarios, it need not be an accurate representation of the actual worst-case request-to-grant delay bounds unless the LFSR can generate CRSes of the same (or similar) length back-to-back. It can be observed by analyzing the random number sequences generated by traditional LFSR designs that most of the time a series of back-to-back CRSes differ only by 1 cycle in length, followed by an abrupt change in the length of the CRS (see Figure 6), because of the way the LFSR is constructed with shift-registers and XOR gates in the feedback paths. Moreover, there are many possible permutations of random numbers ending with a specific missing-random-number - a phenomenon that is hard to avoid in traditional LFSR designs unless newer designs such as a *provably-fair random number generator* [18] is used.

However, for all practical purposes, the product of min/max lengths of a CRS and number of CRSes should provide us a fairly tight (+/- a few cycles) bound on the request-to-grant delay. If the design of the arbiter is such that any request can be starved for a fixed number (N) of CRSes, then in order to determine the *precise* request-to-grant delay bounds, we need to monitor the minimum and maximum length of N *contiguous* CRSes in the random number sequence in step 2. The CRS detection logic and method described in section V-B may still be used for determining the bounds of the length of N CRSes; the only difference is that we will be looking for the lower- and upper- bound of length of N contiguous CRSes instead of *one* CRS.

VI. Results

The formal verification method described has been used for verifying several random priority-based arbiters used in

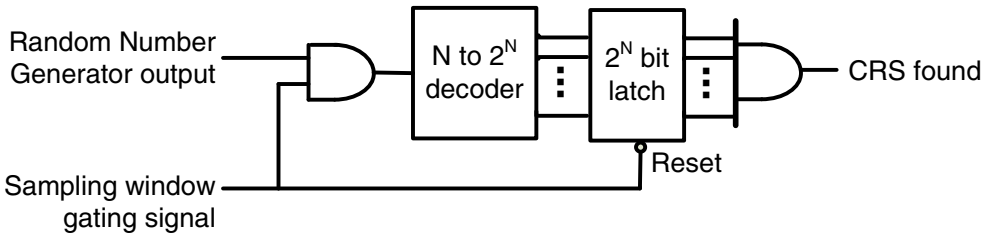

Fig. 5. Logic for checking the fairness of random number generator.

Design	Testbench Type	Traditional				CRS-based			
		Problem Size		Total Time	Peak Memory	Problem Size		Total Time	Peak Memory
		ANDs	Registers	(h:m:s)	(GB)	ANDs	Registers	(h:m:s)	(GB)
8to1ARB_RC	Any	2049	395	24:00:00	6.2	1738	333	0:0:27	0.071
	Multiple	3006	576	24:00:00	5.3	2625	513	0:01:43	0.091
	Bug	2880	554	24:00:00	16.9	2427	464	0:0:06	0.043
4to1ARB_SN	Any	2226	428	24:00:00	7.6	1284	302	0:2:40	0.144
	Multiple	3303	632	23:45:00	22.9	1971	455	0:7:03	0.168
10to1PBARB	Any	3852	770	17:33:07	18.1	2133	394	0:18:53	0.02
	Multiple	3855	770	24:00:00	15.2	3453	682	1:23:40	1.1

TABLE I
RUNTIMES AND MEMORY USAGE FOR DIFFERENT ARBITERS

```
12 | 735210000046
12 | 352100000467
12 | 521000004673
12 | 210000046735
12 | 100000467352
14 | 00000467352521
13 | 0000467352521
12 | 000467352521
11 | 00467352521
10 | 0467352521
10 | 4673525210
10 | 6735252104
10 | 7352521046
46 | 3525210463521421425252563142525210046314252567
45 | 525210463521421425252563142525210046314252567
44 | 25210463521421425252563142525210046314252567
43 | 5210463521421425252563142525210046314252567
42 | 2104635214214252525631425252100463142525 67
41 | 1046352142142525256314252521004631425256 7
40 | 0463521421425252563142525210046314252567
39 | 4635214214252525631425252100463142525 67
38 | 6352142142525256314252521004631425256 7
37 | 3521421425252563142525210046314252567
36 | 5214214252525631425252100463142525 67
35 | 2142142525256314252521004631425256 7
```

Fig. 6. A snippet of 3 bit random number sequence generated by a 16 bit LFSR showing one CRS per line. Each row depicts the length of the CRS followed by the actual CRS.

caches (eg. cache directory port arbitration logic) and on-chip interconnection network controllers (eg. command arbitration logic) of a commercial processor. Table I shows the experimental results on 3 of such industrial designs that use different types of random priority-based arbiters, and an LFSR to generate pseudo-random numbers. There are 3 types of testbenches: type *Any* is used to prove bounded liveness of any input request, whereas type *Multiple* is used for checking starvation of multiple requests. The *Bug* type is similar to the *Any*, except that a larger counter was used to generate a counter-example trace to help the designer fix a bug. All experiments were run on a 1.65 GHz POWER5+ processor with 384 GB memory using IBM's SixthSense (semi-) formal verification tool [14], a state-of-the-art industrial formal verification tool. We used a 2 bit counter ($k = 2$) in all the testbenches, except in the testbenches for 10to1_PBARB and 8to1ARB_RC (bug) in which we used a 4 bit counter. The requests are made non-deterministically to the arbiter with additional constraints on the arrival rate of the requests as per the design specifications. The problem size in terms of number of ANDs and registers, as reported by the SixthSense tool (since it uses AIGs [14] to represent the problem internally), shown is after initial cone-of-influence (COI) reduction and before unrolling the design.

The results are shown for both the proposed CRS-based and traditional bounded-liveness checking (counter-based) approaches. In the traditional approach, we included both the arbiter and the LFSR logic.

The request-to-grant delays determined were found to be in the range of 1 to 7 CRSes long. The bounds of the length of CRSes in the LFSR output used by the arbiters varied from a few tens of cycles to a few hundreds of cycles. For 8to1ARB_RC (bug) we were able to generate a short, yet illustrative, counter-example trace for the designer to identify the bug quickly. Note that using a counter in lieu of the LFSR would not have caught the bug, and in general should not be used as it does not analyze different permutations of random number sequences.

Each of the arbiter designs uses 16-stage LFSR logic. The size of the testbench used for computing the length of CRSes (as described in section V-B) is approximately 3000 ANDs

and 600 registers after initial COI reduction. Each run for determining the length of the CRS took anywhere from a few seconds to 6+ hrs, with an average overall runtime of 1.2 hrs to converge on the final value using a binary search approach.

We set a time limit of 24 hrs for each of the experiments. All the testbenches using the traditional bounded liveness approach were unsolved due to either exceeding the time, or memory limit (data structures exceeding bounds, eg. BDD node index). The total time reported for the latter ones is under 24 hrs. All the CRS based testbenches completed in a very short time using very little memory. It is clear from the experiments that the CRS based scheme significantly outperforms the traditional verification approach for random priority-based arbiters.

Our experience suggests that the proposed three step verification process helps to significantly reduce testbench complexity of random priority-based arbiters, thereby making them amenable for verification using formal verification. Moreover, the arbitration logic is verified in its entirety including the requirements imposed on the random priority-based fairness scheme.

In addition to verifying the correctness of the arbiter, the second verification step described in section V-B was used for tuning the pseudo-random number generators as well. Typically random priority-based arbiters are implemented using large LFSRs, and only a subset of the bits of the LFSR are used to assign priorities to the input requests. By monitoring the upper and lower bounds of the length of the CRS, one can easily choose the optimal tap points of an LFSR logic such that the logic/functional block using the arbiter meets the design specifications.

Furthermore, the request-to-grant delay bounds computed from the RTL-level design can be used for accurately modeling higher levels of hierarchy and larger systems (for example, the processor model) in which random priority-based arbiters are used. This in turn helps to more accurately estimate performance of the processor and systems, and contributes to overall performance verification.

It may be noted that though our main goal is to prove that none of the requests can be starved and to verify the request-to-grant delay specifications, the proposed verification method in effect proves that the arbiter cannot deadlock as well.

VII. Conclusions

A scheme to verify the fairness properties of pseudo-random number generators, and arbiters that use random priority-based arbitration is described. The presented approach uses a logic decomposition and formal verification based method to accurately characterize the worst- and best-case request-to-grant delay of such an arbiter inclusive of the fairness logic. The scheme first determines an upper bound on the request-to-grant delay of the arbiter in terms of the number of complete random sequences (CRS) independent of the pseudo-random number generator. It then separately determines, in terms of the number of clock cycles, an upper bound and a lower bound on the length of a complete random sequence in the random number sequence generated by a fairness logic used by the arbiter, i.e. the random number generator. Finally, the scheme determines a worst-case request-to-grant delay bounds of the arbiter system, in terms of the number of clock cycles, by combining the upper bound of the request-to-grant delay of the arbiter with the upper bound of the length of the CRS and the lower bound of the length of the CRS. The scheme has been successfully used to verify several random priority-based arbiters in the cache units and the on-chip interconnection network controllers.

Acknowledgments

Authors would like to thank Hari Mony for providing optimized SixthSense engine configurations and helpful discussions.

References

[1] W. J. Dally and B. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, 2004, ch. 18, pp. 350–362.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[3] L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Trans. Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977.

[4] P. Horowitz and W. Hill, *The Art of Electronics*, 2nd ed. Cambridge University Press, 1989, pp. 665–667.

[5] A. Goel and W. R. Lee, "Formal Verification of an IBM CoreConnect TM Processor Local Bus Arbiter Core," in *Design Automation Conference (DAC)*, 2000, pp. 196–200.

[6] K. Wasaki, "A Formal Verification Case Study for IEEE-P.896 Bus Arbiter Using A Model Checking Tool," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 7, no. 3, pp. 184–191, 2007.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit verification using Symbolic Model Checking," in *Design Automation Conference (DAC)*, 1990, pp. 46–51.

[8] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[9] R. M. Gott, J. Baumgartner, P. Roessler, and S. I. Joe, "Functional formal verification on designs of pSeries microprocessors and communication subsystems," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 565–580, 2005.

[10] T. Le, T. Gloekler, and J. Baumgartner, "Formal Verification of a Pervasive Interconnect Bus System in a High-Performance Microprocessor," in *Design, Automation and Test in Europe (DATE)*, 2007, pp. 217–224.

[11] N. Dershowitz, D. Jayasimha, and S. Park, "Bounded Fairness," *Lecture Notes in Computer Science*, vol. 2772, pp. 304–317, 2004.

[12] A. Biere, C. Artho, and V. Schuppan, "Liveness Checking as Safety Checking," in *Proc. 7th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)*, ser. ENTCS, vol. 66, no. 2. Elsevier, 2002.

[13] V. Schuppan and A. Biere, "Liveness Checking as Safety Checking for Infinite State Spaces," in *Proc. 7th Intl. Workshop on Verification of Infinite-State Systems (INFINITY '05)*, ser. ENTCS, vol. 149, no. 1. Elsevier, 2006.

[14] H. Mony, J. Baumgartner, V. Paruthi, R. L. Kanzelman, and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2004, pp. 159–173.

[15] J. Baumgartner and H. Mony, "Scalable Liveness Checking via Property-Preserving Transformations," in *Design Automation and Test in Europe*, 2009.

[16] S. W. Golomb, *Shift Register Sequences*. Aegean Park Press, 1982.

[17] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley, 1997, vol. 3: Sorting and Searching.

[18] K. K. Kailas, B. C. Monwai, and V. Paruthi, "Method and Structure for Provably Fair Random Number Generator," U.S. Patent application 12/101,734, April 11, 2008, IBM docket No. YOR920080134US1.