Probabilistic Verification for Reliable Network-on-Chip System Design

Benjamin Lewis[1(✉)], Arnd Hartmanns[2], Prabal Basu[1], Rajesh Jayashankara Shridevi[1], Koushik Chakraborty[1], Sanghamitra Roy[1], and Zhen Zhang[1]

[1] Utah State University, Logan, UT, USA
{benjamin.lewis,prabalb,rajesh.js}@aggiemail.usu.edu,
{koushik.chakraborty,sanghamitra.roy,zhen.zhang}@usu.edu
[2] University of Twente, Enschede, The Netherlands
a.hartmanns@utwente.nl

Abstract. The design of modern network-on-chip (NoC) systems faces reliability challenges due to process and environmental variations. Peak power supply noise (PSN) in the power delivery network of a NoC device plays a critical role in determining reliable operations: PSN typically leads to voltage droop, which can cause timing errors in the NoC router pipelines. Existing simulation-based approaches cannot provide rigorous, worst-case reliability guarantees on the probabilistic behaviors of PSN. To address this problem, this paper takes a significant step in formally analyzing PSN in modern NoCs. Specifically, we present a probabilistic model checking approach for the rigorous characterization of PSN for a generic central router of a large mesh-NoC system, under the Round Robin scheduling mechanism with a uniform random network traffic load. Defining features for PSN are extracted at the behavioral level to facilitate property formulation. Several abstract models have been derived for the central router's concrete model based on the observations of its arbiter's conflict resolution behavior. Probabilistic modeling and verification are performed using the Modest Toolset. Results show significant scalability of our abstract models, and reveal key PSN characteristics that are indicative of NoC design and optimization.

Keywords: Probabilistic model checking · Network-on-chip · Reliability analysis · Power supply noise

1 Introduction

The advancement in probabilistic model checking has enabled its applications in a wide range of domains, including cryptography [11], systems biology [22], network protocols [21], game theory [6], and distributed systems [20]. Likewise, in recent times, the growing demand for robust and secure digital system design has challenged the potential for innovation in formal methods. In this work, we

© Springer Nature Switzerland AG 2019
K. G. Larsen and T. Willemse (Eds.): FMICS 2019, LNCS 11687, pp. 110–126, 2019.
https://doi.org/10.1007/978-3-030-27008-7_7

venture into the probabilistic model checking of the reliability evaluation of a complex and distributed digital system—the on-chip communication network, *network-on-chip* (NoC), deployed in a many-core system.

NoC—the de-facto standard for on-chip communication in modern many-core systems—*generally* comprises of several topologically homogeneous routers operating synchronously in a decentralized control system. Despite the conceptual similarity with conventional computer networks, a NoC is subject to several unique reliability challenges, e.g., process and environmental variations, that are vastly dissimilar to conventional network communication. Over a decade of simulation-based research has gone into NoC design exploration and reliability analysis [1–3, 27]. However, simulation-based ecosystems fail to provide worst-case reliability and safety guarantees. Consequently, formal verification is necessary to ensure the correctness of specific functionality of the NoC components.

The primary challenge of applying automated verification, specifically, model checking, is the notorious state explosion issue, as evidenced by a recent work on model checking an asynchronous NoC [31] where the intermediate state space corresponding to only 13 out of the 66 components in a 3×3 NoC consists of several hundred million states. Consequently, accurate modeling of the reliability issues (e.g., power supply noise, quality-of-service guarantees, etc.) is poised to further aggravate its computational complexity.

This paper presents a probabilistic model checking method for the analysis of *power supply noise* (PSN) for a generic central router of a large mesh-NoC system and its impact on the router's reliability under uniform random traffic loads. To enable an accurate and efficient analysis and a convenient formulation of the probabilistic properties, we extract the key characterizing features of the router at the behavioral level. We present formal models for the central router with four full-duplex channels, operated under uniform random packet injection with the starvation-free Round Robin conflict resolution scheduling. To tackle the state space explosion challenge, abstract models have been derived based on critical observations of conflict resolution patterns. Transition probabilities between abstract states are inferred from exhaustive executions of the underlying concrete models with limited steps. We use the high-level formal modeling language Modest [13] to formulate our models, the state spaces of which are large discrete-time Markov chains (DTMC), and the Modest Toolset's [15] probabilistic model checker mcsta for the analysis. We check *reward-bounded* properties, for which mcsta implements scalable analysis techniques [12]; in particular the state elimination approach resulted in significant analysis speedups. The final verification results show significant scalability of our abstract models, and reveal key relations between traffic loads and PSN.

2 Motivation

PSN in the power delivery network of an integrated circuit is composed of two major components: (a) *resistive noise*, which is estimated by the product of the current drawn and the lumped resistance of the circuit; and (b) *inductive noise*

that is caused by the inductance the power grid and is proportional to the rate of change of current through the inductance ($\frac{\Delta i}{\Delta t}$). For a distributed system such as a NoC, the latter takes a central component [2].

A high inductive noise is responsible for the intermittent peaks in the cycle-wise noise profile of a NoC. Basu et al. have recently demonstrated that, in an 8×8 NoC, the peak PSN can increase from 40% of the supply voltage at the 32-nm technology node to about 80% of the supply voltage at the 14-nm technology node, while running a uniform-random synthetic traffic pattern [2]. Voltage droop due to PSN can radically degrade the delay of various on-chip circuit components. Such increase in the delay has the potential to engender *timing errors* in the pipe-stages of the NoC routers, thus severely impacting the reliability as well as the performance of the overall on-chip communication.

Although recent works [2,27] tackle the PSN problem in NoCs to some extent, they do not guarantee the worst-case peak PSN—a determinant of the NoC reliability—across different operating conditions, realistically conceivable, for any parallel workload. Moreover, these works do not provide any bounds on the temporal PSN profile for a router, given an application execution. Consequently, existing approaches to mitigate PSN are a far cry from a truly reliable NoC design paradigm that can be deployed in mission-critical systems. On the contrary, this work shows that probabilistic model checking, despite its inherent challenges, can offer precise bounds on the performance and reliability with common environment assumptions, leading the way to future reliable NoC design.

3 Related Work

Reliable and energy efficient communication is the backbone of many-core systems. Significant recent research exploring *reactive*, *proactive* and *predictive* techniques has focused on addressing the challenges of fault tolerance in NoC [4,5,7,17,28]. However, a wide majority of these works are simulation-based analyses, which cannot provide rigorous reliability and performance guarantees.

Formal verification in NoC architectures has largely been focused on functional correctness of routing algorithms [26,31,32]. Zhang et al. investigate properties of deadlock and livelock freedom and tolerance to link failure, and use model checking to enhance an existing routing protocol [31,32]. Based on theorem proving techniques, the DCI2 developed by Verbeek demonstrates significant scalability in proving properties of deadlock and livelock freedom and topology violations of statically determined routing logic [29]. Accurate assessment of NoC reliability has to incorporate quantitative aspects depicting the inherent distributive and reactive nature of NoC. Coste et al. presents in [8] a translation procedure to convert existing functional model into Markov chains for the evaluation of the latency of memory accesses over a *Globally Asynchronous Locally Synchronous* (GALS) NoC. Nevertheless, the scope of existing literature in probabilistic verification of the NoC is minimal.

On the other hand, researchers have extensively employed probabilistic verification to assess and improve reliability, resilience, and security of computer

hardware designs [10,24,25,30]. For example, Han et al. demonstrate how to obtain the fundamental error bounds by using bifurcation analysis based on probabilistic models of unreliable gates [14]. Kumar et al. propose an automatic compositional reasoning technique to improve the scalability of probabilistic model checking of hardware systems [19]. Mundhenk et al. propose probabilistic model checking for the security analysis of automotive architectures at the system level [23]. However, the dividends of these works have not yet been carried forward to the NoC domain.

4 Conflict Resolution in Central Routers

Figure 1 depicts the central router of an 8×8, two-dimensional mesh NoC network [2]. It has four full-duplex channels with the bandwidth of one flit of a network packet, where each channel has a buffer with the capability of storing four flits. The router simultaneously transmits and receives flits in all four directions. Assume that each flit carries the next forwarding direction, and that a flit is not diverted back to its incoming direction. The forwarding direction is used for the arbiter in the central router to detect possible conflicts. The arbiter resolves conflicts, created by multiple flits originated from the four buffers attempting the same output direction. The order of conflict resolution relies on the Round Robin scheduling mechanism to guarantee fairness and starvation-free arbitration. The input interface handles flits arrived from all of four directions, and accommodates them in the first available space in the corresponding buffer. The output interface directs flits from the arbiter that are ready to be dispatched to the neighboring routers. The rest of this section describes details of the arbiter's conflict resolution mechanism.

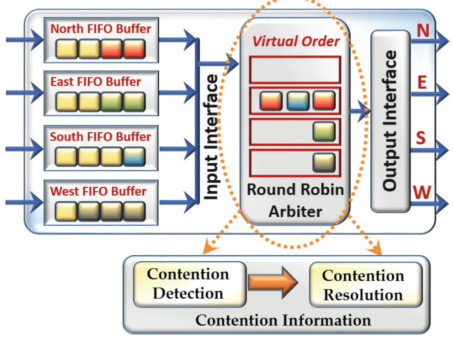


Fig. 1. NoC router model.

Since the bandwidth of all outgoing channels allows only one flit at a time, conflicts are resolved inside the central router. Conflicts affect the performance of each individual router and hence the entire NoC. During each clock cycle, the arbiter first examines each buffer's front flit's outgoing direction to detect conflicts. If no conflict exists, all the buffers can forward their front flits to their respective outputs in one cycle to maximize the throughput. Otherwise, the arbiter has to resolve all conflicts, requiring one or more additional cycles. Figure 2 demonstrates three representative scenarios of conflicts and their resolution. For simplicity, we ignore the incoming packets to all four buffers at each cycle, and only illustrate conflict resolution. In Scenario A, only one conflict exists between the east and west buffers at cycle t_n, and the east buffer has higher priority. The arbiter, therefore, serves the east buffer at cycle t_n, and

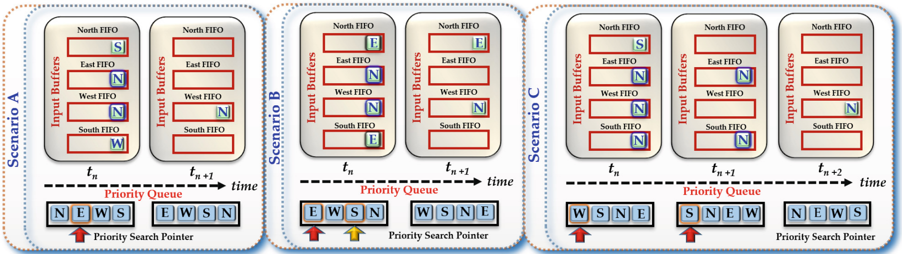


Fig. 2. Conflict scenarios.

it also directs the north and south buffers in this cycle, as their flits do not conflict. In the following cycle t_{n+1}, the priority gets updated by shifting the priority queue, and the west buffer is served. Scenario B demonstrates two pairs of conflicts: the north and south buffers compete for the east output, while east and west compete for the north output in cycle t_n. Following the priority queue, the arbiter serves the east and south buffers in cycle t_n, and the west and north buffers in the next cycle. Scenario C illustrates a three-way conflict: all front flits of the east, west, and south buffers compete for the north output. The arbiter serves the west buffer first, and simultaneously serves the north buffer as it is not conflicting with others. In the following cycle t_{n+1}, the south buffer gets serviced, as it has higher priority than the east, leaving the east buffer to be serviced in cycle t_{n+2}.

5 Formal Model of the Central Router

The formal model implements all potential conflicts in the central router. The Modest language [13] is used to model the router as shown in Fig. 1. We introduce a datatype `buffer` shown in Listing 1.1. Integer variable `dest` represents the front flit's destination in each buffer: 0 (north), 1 (east), 2 (south), or 3 (west). Value -1 indicates an empty buffer. The field `id` stores the buffer location in the central router, with the same encoding as the flit's destination. Variable `serviced` is `true` if the front flit was serviced in the current cycle, and `false` otherwise. The `priority` field represents the priority position each buffer will occupy in the next clock cycle. Lastly, the actual buffer, `buff`, is modeled as an integer linked list. The size of `buff` is set to four for all models presented in this paper, but its length can be set to any finite integer. The arbiter model `arb` is an array of four `buffer` values. The position of `buffer` in `arb` represents the current priority for servicing all four buffers, array index 0 being the highest priority and 3 being the lowest. For example, if `arb[1]` refers to the east buffer with `id = 1` and `priority = 3`, then at the beginning of the next cycle the buffer will have been moved to position `arb[3]`. Two internal integer variables, namely, `unserviced` and `totalUnserviced`, are used in the Round Robin scheduling mechanism: `unserviced` counts the number of unserviced buffers in one cycle due to

Listing 1.1. Buffer model.

```
datatype buffer = {int(-1..3) dest, int(0..3) id,
    bool serviced, int(0..3) priority, intlist option buff};
buffer north, east, south, west;
buffer[] arb = [north, east, south, west];
int(0..2) unserviced;
int(0..2) totalUnserviced;
```

Listing 1.2. Procedure for updating `serviced` and `unserviced`.

```
arb[0].serviced = true;
if (arb[1].dest != -1 && arb[1].dest == arb[0].dest) {
    arb[1].serviced = false;
    unserviced++;
}
else {
    arb[1].serviced = true;
}
if(arb[1].dest != -1 && (arb[2].dest == arb[1].dest || arb
    [2].dest == arb[0].dest)) {
        arb[2].serviced = false;
        unserviced++;
}
else {
        arb[2].serviced = true
}
if(arb[1].dest != -1 && (arb[3].dest == arb[2].dest || arb
    [3].dest == arb[1].dest || arb[3].dest == arb[0].dest)) {
        arb[3].serviced = false;
        unserviced++;
}
else {
        arb[3].serviced = true;
}

totalServiced = unserviced;
```

conflict, and decrements as the unserviced buffer's priority values are calculated; and `totalUnserviced` tracks the total number of buffers unserviced in one cycle.

The `serviced` field for each buffer and `unserviced` are updated by the procedure shown in Listing 1.2. It automatically sets `serviced` to `true` for the buffer in position 0, because the arbiter will definitely serve this buffer in the current cycle. It then moves on to the buffers in all remaining positions. If a buffer is non-empty and is in conflict with another buffer with higher priority, then the latter will be serviced in the current cycle and the former has to wait for its chance in the next cycle. Therefore, `serviced` of the former is set to `false` and `unserviced` is incremented. Otherwise `serviced` is set to `true`. Lastly, the arbiter assigns `totalUnserviced` the updated `unserviced`.

Next, priority for each buffer gets updated using the procedure shown in Listing 1.3. It should be noted that priority update is assumed to strictly follow the order shown in the procedure, starting with the buffer in position 0 of the arbiter array `arb`. If the buffer at `arb[i]` was serviced, its `dest` is updated by peeking the front of the corresponding buffer, followed by an update of the buffer

Listing 1.3. Update priority.

```
for(int i = 0, i < 3, i++) {
    if (arb[i].serviced == true) {
        arb[i].dest = peekFront(arb[i].buff);
        arb[i].buff = dequeue(arb[i].buff);
        arb[i].priority = i + unserviced;
    }
    else {
        arb[i].priority = totalUnserviced - unserviced;
        unserviced--;
    }
}
```

itself. If no element is in `buff`, the `peekFront` function will return -1 to indicate an empty buffer. The `priority` is updated to the sum of its current priority `i` and the number of unserviced buffers, whose priorities have not been updated. Intuitively, buffers not serviced in the current cycle will be given higher priority in the next, and those serviced receive priority corresponding to their position in the arbiter array. If the buffer was not serviced, the `priority` is determined by subtracting `unserviced` from the total number of unserviced buffers in the current cycle, after which the `unserviced` is decremented. We use this method to keep track of the order for buffers that did not get serviced in the current cycle.

As an example, assume `arb=[north, east, south, west]`, and `serviced` are `true`, `false`, `true`, and `false`, respectively. Both `unserviced` and `totalUnserviced` are set to 2, because the east and west buffers were not serviced in the current cycle. Priority updates start with `arb[0]`, i.e., the north buffer. Since it was serviced in the current cycle, its priority is updated to $0 + 2 = 2$. The value of `unserviced` remains at 2. Next, the priority is updated for the east buffer. Because it was not serviced in the current cycle, its priority is set to $2-2 = 0$, giving itself the top priority for the next cycle. The value of `unserviced` then decrements from 2 to 1, indicating that one remaining unserviced buffer is scheduled for the next cycle. Similarly, priorities for the south and west buffers are updated to $2+1 = 3$ and $2 - 1 = 1$, respectively. The variable `unserviced` decrements to 0 after all priory updates. The resulting arbiter array is `[east, west, north, south]`.

To model incoming flits to all four buffers, we randomly assign their `dest` fields using the discrete uniform distribution, with the exception that a buffer does not receive a flit destined to its incoming direction. Probabilistic model checking on this routing node model incurs exponential state space growth as cycles increase, quickly becoming too large to be handled. For 100 clock cycles, MCSTA explored 400 million states with another 100 million queued for expansion when 132 GB memory were filled. This is mainly due to the combinatorial explosion of flit values. To address this issue, we present several abstract router models next.

6 Abstract Models and Refinement

Abstract models presented in this paper are based on an ad-hoc method specifically for the central router. The initial abstraction is based on the observation

that rather than specific scenarios of conflicts formed by the `dest` field of four buffers, the arbiter's behavior is *only* determined by a few conflict patterns that can co-exist in one cycle, including non-conflicting scenarios. This observation leads to four abstract states: (1) no conflicts, where all buffers are serviced in the current cycle; (2) one pair of conflicts, where the only unserviced buffer is the one with lower priority in the pair; (3) two pairs of conflicts with two unserviced buffers, both of which have lower priority compared to their conflicting counterparts; and (4) three buffers in conflict, where the two buffers with low priorities are not serviced in the current cycle. Four buffers cannot all be in conflict as it is assumed that a flit is not diverted back to its incoming direction.

Since the abstract model is formulated at the behavioral level without circuit-level details, one has to project the measure of PSN onto the same abstract level. We know that the inductive noise, a major source of PSN, is proportional to the rate of change of current in the circuit. An abrupt change in the router activity in two consecutive cycles directly leads to a high rate of current change [2]. A low router activity is characterized by the arbiter serving no routers in a cycle, as all buffers are empty; while a high router activity is indicated by the arbiter serving three or more buffers in a cycle. The relative frequency of both high-to-low and low-to-high activities over a given timespan can, therefore, accurately reflect the state of the local noise and hence PSN in the NoC routers. For this purpose, we consider the following two probabilistic properties: (1) the probability that the number of high router activity cycles is lower-bounded by $k \cdot N$ within N overall cycles; and (2) the probability that the total number of high-to-low and low-to-high activities is lower-bounded by $k \cdot N$ within N overall cycles, where $k \in (0, 1)$. High router activity, as indicated by property (1), can potentially create a high local congestion in the network, leading to a high PSN due to an unbalanced power density [9]. On the other hand, property (2) reflects a large and sudden load change in a router that can lead to a large inductive drop in the power delivery network of the NoC [2]. Collectively, understanding these properties is essential to ascertain the minimum voltage guardband for the NoC, sufficient to ensure a fault-free communication in a many-core system. To facilitate checking of these properties, two variables are created, namely, `optimalRuns`, which increments if all four buffers are serviced in a cycle, and `noiseRuns`, which accumulates cycles with high-to-low or low-to-high activities. Formulation of these properties is presented in Sect. 8.

The initial abstract model, however, is incorrect in that after two clock cycles, the accumulation of `optimalRuns` diverges from that obtained from the concrete model. This is because the probability varies when transitioning between two states with two-pair conflict. Specifically, different scenarios of two-pair conflict result in different probabilities. Table 1 illustrates some examples. Each entry listed under columns `arb[i]` shows the buffer location and the destination of its front element. For example, "n(e)" under column `arb[2]` means that the north buffer's front flit is destined for the east output. The entry "w(n, e, s)" in the same column indicates that the west buffer can receive a flit destined to any other three directions. For state "2a", if the arbiter has the two-pair conflict in

cycle k, then `arb[0]` and `arb[1]` are serviced, allowing "n(e)" and "e(n)" to move to `arb[0]` and `arb[1]`, respectively, in the next cycle. Observe that at cycle $k+1$, two-pair conflict scenarios include (n(e), e(n), w(n), s(e)) and (n(e), e(n), w(e), s(n)), and the possible three-way conflicts are (n(e), e(n), w(e), s(e)) and (n(e), e(n), w(n), s(n)). For state "2b" with a different two-pair conflict pattern at cycle k, the next cycle can only form the two-pair conflict (w(n), s(e), n(e), e(n)), and no three-way conflict can exist. For state "2c", the only two-pair conflict is (w(s), s(e), n(e), e(s)), and the only three-way conflict is (w(s), s(e), n(s), e(s)). Our analysis reveals that such discrepancies exist in other abstract states.

Table 1. History-dependent conflict examples.

State	Cycle	arb[0]	arb[1]	arb[2]	arb[3]
2a	k	w(n)	s(e)	n(e)	e(n)
	$k+1$	n(e)	e(n)	w(n, e, s)	s(n, e, w)
2b	k	n(e)	e(n)	w(n)	s(e)
	$k+1$	w(n)	s(e)	n(e, w, s)	e(n, w, s)
2c	k	n(e)	e(s)	w(s)	s(e)
	$k+1$	w(s)	s(e)	n(e, w, s)	e(n, w, s)

The four-abstract model is refined based on an analysis of all possible inputs into the arbiter and their respective behaviors. The possible inputs can be grouped into thirteen behaviors which are defined as states as shown in Table 2. Each refined state is conditioned on the number of unserviced buffers at the end of a cycle and where the flit's destination points, specifically, buffer locations (i.e., the `id` field) the destinations at the arbiter's positions 0 and 1 point to. This table shows predicates defining these refinement conditions. Notations have been simplified as follows: $dest_i$ represents the front flit's destination of the buffer at index i of the arbiter array, i.e., `arb[i].dest`.

To calculate transition probabilities among the thirteen abstract states, we modify the concrete model to include two variables: s_{prev} and s. For every clock cycle, s_{prev} first updates to s and then all predicates in Table 2 are evaluated and s is updated accordingly. Assuming the model starts with no conflicts ($s_{prev} = 0$), we observe that for *up to* two transitions, which corresponds to two clock cycles, *every* one of the thirteen states in Table 2 is reachable. Transition probability emanating from state 0 to an abstract state, say 1b, is calculated by summarizing all probabilities of transitioning from the concrete state 0, which is the same as the abstract state 0, to all concrete states that satisfy the predicate for state 1b, which is $unserviced = 1 \wedge dest_0 = id_2$. For this calculation, we added to the model a variable clk that is incremented with every clock cycle. The calculation is then performed by first using MCSTA to query for

$$\mathbf{P}_{=?}(\diamond\,(clk = 2 \wedge s_{prev} = 0 \wedge s = 2)),$$

i.e., the probability to eventually ($\diamond$) reach a state in the model after two clock cycles where $s_{prev} = 0$ and the new abstract state is $s = 2$, i.e., 1b in Table 2. We then divide the result by the sum of probabilities out of state 0. Other transition probabilities are calculated similarly. Another observation is that the next states and transition probabilities from states 2b and 2c are identical, so we combine them into state 2b to form a twelve-state abstract model as shown in Table 3.

Table 2. Refined abstract model with thirteen states.

State	Predicate
0	$unserviced = 0$
1a	$unserviced = 1 \wedge dest_0 = id_1$
1b	$unserviced = 1 \wedge dest_0 = id_2$
1c	$unserviced = 1 \wedge dest_0 = id_3$
2a	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_1 \wedge dest_1 = id_0$
2b	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_2 \wedge dest_1 = id_3$
2c	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_3 \wedge dest_1 = id_2$
2d	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_1 \wedge dest_1 = id_2$
2e	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_1 \wedge dest_1 = id_3$
2f	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_2 \wedge dest_1 = id_0$
2g	$unserviced = 2 \wedge dest_0 \neq dest_1 \wedge dest_0 = id_3 \wedge dest_1 = id_0$
3a	$unserviced = 2 \wedge dest_0 = dest_1 = id_2$
3b	$unserviced = 2 \wedge dest_0 = dest_1 = id_3$

7 Including Idle Cycles in the Abstract Model

The twelve-state abstract model shown in Table 3 assumes that a flit is injected to all buffers in *every* cycle. This is, however, not quite realistic as it is common that one or more buffers do not receive an incoming flit. Such situations change the conflict patterns and hence the arbiter's resolution behavior. From [2], we know that the cycle-wise and intermittent PSN is a direct result of a significant change of buffers served. Precisely three or four buffers are serviced by the arbiter between two consecutive clock cycles. Counting these changes allows us to accurately reflect the state of the local noise and hence PSN in the NoC routers. This implies that change from serving zero to four buffers and vice versa needs to be modeled. This section describes a modified abstract model to include idle cycles for each buffer. Using similar method as described in Sect. 6, refinement is applied to the twelve-state abstract model to account for scenarios with three, two, one, and none serviced buffers in one cycle. This leads to the twenty-five-state abstract model provided in Table 4. The state notation in this table represents the conflict scenario and the number of buffers with incoming flits in a given state. For example state 2_4^b represents the state with the conflict scenario $2b$ in Table 2 in which four buffers have incoming flits. Note that this refinement does not change the fact that probabilities in this table can be calculated by checking its underlying concrete model for two clock cycles as described in Sect. 6.

Table 3. Twelve-state abstract model with transition probabilities.

	0	1a	1b	1c	2a	2b	2d	2e	2f	2g	3a	3b
0	$\frac{1}{9}$	$\frac{4}{27}$	$\frac{16}{81}$	$\frac{20}{81}$	$\frac{1}{27}$	$\frac{3}{81}$	$\frac{2}{81}$	$\frac{1}{81}$	$\frac{1}{81}$	$\frac{2}{81}$	$\frac{1}{27}$	$\frac{1}{9}$
1a	$\frac{1}{9}$	$\frac{4}{27}$	$\frac{8}{27}$	$\frac{4}{27}$	0	$\frac{2}{27}$	0	$\frac{1}{27}$	0	$\frac{1}{27}$	$\frac{1}{27}$	$\frac{1}{9}$
1b	$\frac{1}{9}$	$\frac{4}{27}$	$\frac{2}{9}$	$\frac{2}{9}$	$\frac{1}{27}$	$\frac{1}{27}$	0	0	$\frac{1}{27}$	$\frac{1}{27}$	$\frac{1}{27}$	$\frac{1}{9}$
1c	$\frac{1}{9}$	$\frac{4}{27}$	$\frac{2}{27}$	$\frac{10}{27}$	$\frac{2}{27}$	0	$\frac{2}{27}$	0	$\frac{1}{81}$	0	$\frac{1}{27}$	$\frac{1}{9}$
2a	$\frac{1}{9}$	$\frac{2}{9}$	$\frac{2}{9}$	0	0	$\frac{2}{9}$	0	0	0	0	$\frac{1}{9}$	$\frac{1}{9}$
2b	$\frac{2}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{4}{9}$	$\frac{1}{9}$	0	0	0	0	0	0	0
2d	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{3}$	$\frac{2}{9}$	0	0	0	0	0	$\frac{1}{9}$	0	$\frac{1}{9}$
2e	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{3}$	$\frac{2}{9}$	0	0	0	$\frac{1}{9}$	0	0	0	$\frac{1}{9}$
2f	$\frac{1}{9}$	$\frac{1}{3}$	$\frac{1}{9}$	$\frac{2}{9}$	0	0	0	0	$\frac{1}{9}$	0	$\frac{1}{9}$	0
2g	$\frac{1}{9}$	$\frac{1}{3}$	$\frac{1}{9}$	$\frac{2}{9}$	0	0	$\frac{1}{9}$	0	0	0	$\frac{1}{9}$	0
3a	0	0	$\frac{4}{9}$	0	0	$\frac{1}{9}$	0	0	0	$\frac{1}{9}$	0	$\frac{1}{3}$
3b	0	0	0	$\frac{4}{9}$	$\frac{1}{9}$	0	$\frac{1}{9}$	0	0	0	0	$\frac{1}{3}$

8 Verification Results

All experiments have been performed on the abstract central router models, which are constructed as DTMC models using the high-level compositional modeling language MODEST. The explicit-state probabilistic model checker MCSTA in the MODEST TOOLSET has been used for verification. Properties (1) and (2) are bounded probabilistic reachability queries for the transient behavior up to N clock cycles, with N being a rather large number. Implementing the cycle counter *clk* as a state variable, which we did for the computations in Sect. 6 with bound 2, would *unfold* the model over the cycle count up to the (now large) bound, exacerbating the state space explosion problem. To avoid this problem now, we made *clk* a *transient* variable that is set to 1 when moving from one clock cycle to the next and to 0 otherwise. A transient variable is only "live" during the assignments executed when taking a transition; it is not part of the state vector. In this way, clock cycle progress becomes a *reward* annotation to certain transitions instead of being encoded in the structure of an (unfolded) state space. We can then formalize properties (1) and (2) as reward-bounded reachability queries:

$$(1)\quad \mathbf{P}_{=?}(\diamond^{[\text{accumulate}(clk)\leqslant N]}\ \texttt{optimalRuns} \geqslant k\cdot N)$$

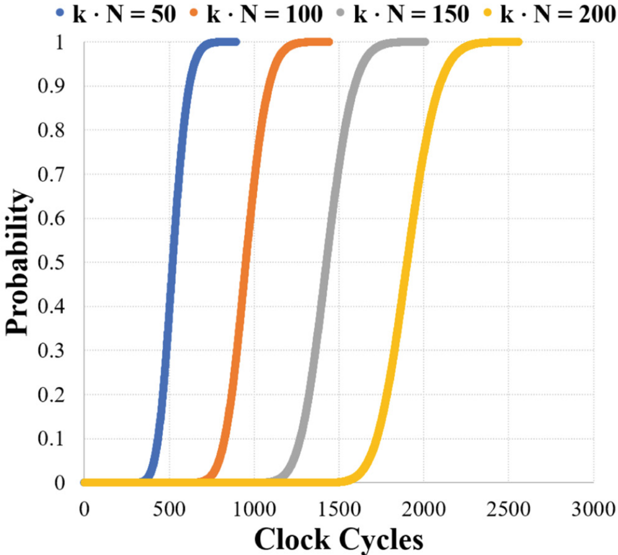
$$(2)\quad \mathbf{P}_{=?}(\diamond^{[\text{accumulate}(clk)\leqslant N]}\ \texttt{noiseRuns} \geqslant k\cdot N)$$

We use the state elimination method [12] implemented in MCSTA for the reward-bounded property checking reported in this section. For our experiments, it provides a significant scalability and efficiency improvement over the classic unfolding-based approaches, but also over the default modified-iteration method, both of which we attempted to use in earlier versions of this model. In this way, our experience mirrors the performance behaviour observed earlier in [12,16,18].

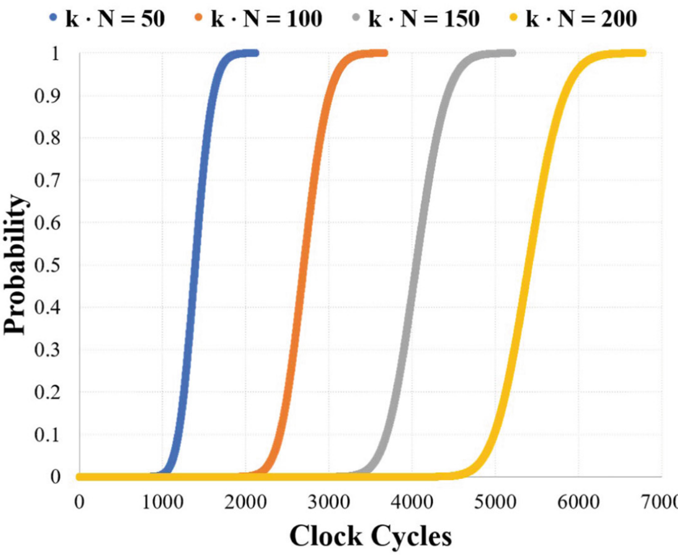
Table 4. Twenty-five-state abstract model with transition probabilities.

	0_4	1_4^a	1_4^b	1_4^c	2_4^a	2_4^b	2_4^c	2_4^d	2_4^e	2_4^f	2_4^g	3_4^a	3_4^b	0_3	1_3^a	1_3^b	1_3^c	3_3^a	3_3^b	0_2	1_2^a	1_2^b	1_2^c	0_1	0_0
0_4	$\frac{9}{256}$	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{5}{64}$	$\frac{3}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{3}{256}$	$\frac{9}{256}$	$\frac{11}{64}$	$\frac{15}{256}$	$\frac{5}{64}$	$\frac{25}{256}$	$\frac{1}{256}$	$\frac{3}{256}$	$\frac{21}{128}$	$\frac{3}{256}$	$\frac{1}{64}$	$\frac{5}{256}$	$\frac{1}{64}$	$\frac{1}{256}$
1_4^a	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$	0	$\frac{1}{64}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{5}{32}$	$\frac{1}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{32}$	0	$\frac{1}{64}$	0
1_4^b	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{3}{32}$	$\frac{3}{32}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{3}{32}$	$\frac{3}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	0
1_4^c	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{5}{32}$	$\frac{1}{32}$	0	0	$\frac{1}{32}$	0	0	0	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{1}{64}$	$\frac{11}{64}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	0	$\frac{1}{32}$	$\frac{1}{64}$	0
2_4^a	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{8}$	0	0	$\frac{1}{16}$	$\frac{1}{16}$	0	0	0	0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{32}$	$\frac{1}{32}$	0	0	0	$\frac{1}{64}$	0	0	0	0	0
2_4^b	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	0	0	0	0	0	0	0	$\frac{1}{4}$	0	0	$\frac{1}{8}$	0	0	$\frac{1}{16}$	0	0	0	0	0
2_4^c	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{4}$	$\frac{1}{16}$	0	0	0	0	0	0	0	0	$\frac{1}{4}$	0	0	$\frac{1}{8}$	0	0	$\frac{1}{16}$	0	0	0	0	0
2_4^d	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{1}{16}$	0	0	0	0	0	0	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{1}{8}$	$\frac{3}{16}$	0	$\frac{1}{8}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	0
2_4^e	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{1}{16}$	0	0	0	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	$\frac{3}{16}$	0	$\frac{1}{8}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	0
2_4^f	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	0	0	0	$\frac{1}{16}$	0	$\frac{1}{16}$	0	$\frac{3}{16}$	$\frac{1}{8}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	0
2_4^g	$\frac{1}{16}$	$\frac{3}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	$\frac{3}{16}$	$\frac{1}{8}$	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	0
3_4^a	0	0	$\frac{1}{4}$	0	0	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	$\frac{3}{16}$	0	0	$\frac{5}{16}$	0	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0
3_4^b	0	0	0	$\frac{1}{4}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	$\frac{3}{16}$	0	0	0	$\frac{5}{16}$	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	0
0_3	$\frac{9}{256}$	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{5}{64}$	$\frac{3}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{3}{256}$	$\frac{9}{256}$	$\frac{11}{64}$	$\frac{15}{256}$	$\frac{5}{64}$	$\frac{25}{256}$	$\frac{1}{256}$	$\frac{3}{256}$	$\frac{21}{128}$	$\frac{3}{256}$	$\frac{1}{64}$	$\frac{5}{256}$	$\frac{1}{64}$	$\frac{1}{256}$
1_3^a	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$	0	$\frac{1}{64}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{5}{32}$	$\frac{1}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{32}$	0	$\frac{1}{64}$	0
1_3^b	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{3}{32}$	$\frac{3}{32}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{3}{32}$	$\frac{3}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	0
1_3^c	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{5}{32}$	$\frac{1}{32}$	0	0	$\frac{1}{32}$	0	0	0	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{1}{64}$	$\frac{11}{64}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	0	$\frac{1}{32}$	$\frac{1}{64}$	0
3_3^a	0	0	$\frac{1}{4}$	0	0	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	$\frac{3}{16}$	0	0	$\frac{5}{16}$	0	0	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0
3_3^b	0	0	0	$\frac{1}{4}$	$\frac{1}{16}$	0	0	$\frac{1}{16}$	0	0	0	0	$\frac{3}{16}$	0	0	0	$\frac{5}{16}$	0	$\frac{1}{16}$	0	0	0	$\frac{1}{16}$	0	0
0_2	$\frac{9}{256}$	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{5}{64}$	$\frac{3}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{3}{256}$	$\frac{9}{256}$	$\frac{11}{64}$	$\frac{15}{256}$	$\frac{5}{64}$	$\frac{25}{256}$	$\frac{1}{256}$	$\frac{3}{256}$	$\frac{21}{128}$	$\frac{3}{256}$	$\frac{1}{64}$	$\frac{5}{256}$	$\frac{1}{64}$	$\frac{1}{256}$
1_2^a	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{16}$	0	$\frac{1}{64}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{5}{32}$	$\frac{1}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{32}$	0	$\frac{1}{64}$	0
1_2^b	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{3}{32}$	$\frac{3}{32}$	$\frac{1}{64}$	0	$\frac{1}{64}$	0	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{3}{32}$	$\frac{3}{32}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	$\frac{1}{64}$	$\frac{1}{64}$	$\frac{1}{64}$	0
1_2^c	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{5}{32}$	$\frac{1}{32}$	0	0	$\frac{1}{32}$	0	0	0	$\frac{1}{64}$	$\frac{3}{64}$	$\frac{11}{64}$	$\frac{3}{64}$	$\frac{1}{64}$	$\frac{11}{64}$	0	$\frac{1}{64}$	$\frac{7}{64}$	0	0	$\frac{1}{32}$	$\frac{1}{64}$	0
0_1	$\frac{9}{256}$	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{5}{64}$	$\frac{3}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{3}{256}$	$\frac{9}{256}$	$\frac{11}{64}$	$\frac{15}{256}$	$\frac{5}{64}$	$\frac{25}{256}$	$\frac{1}{256}$	$\frac{3}{256}$	$\frac{21}{128}$	$\frac{3}{256}$	$\frac{1}{64}$	$\frac{5}{256}$	$\frac{1}{64}$	$\frac{1}{256}$
0_0	$\frac{9}{256}$	$\frac{3}{64}$	$\frac{1}{16}$	$\frac{5}{64}$	$\frac{3}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{256}$	$\frac{1}{128}$	$\frac{3}{256}$	$\frac{9}{256}$	$\frac{11}{64}$	$\frac{15}{256}$	$\frac{5}{64}$	$\frac{25}{256}$	$\frac{1}{256}$	$\frac{3}{256}$	$\frac{21}{128}$	$\frac{3}{256}$	$\frac{1}{64}$	$\frac{5}{256}$	$\frac{1}{64}$	$\frac{1}{256}$

Results are generated on a desktop computer with an AMD Ryzen Threadripper 12-Core 3.5 GHz Processor and 132 GB memory, running Ubuntu Linux. One core is used at any time. All results presented in this section assume uniform random packet arrival at all four buffers. Verification results for property



(a) Cumulative probability for **optimalRun**.



(b) Cumulative probability for **noiseRun**.

Fig. 3. The two probabilistic properties denote high activity (Fig. 3a) and high change in activity (Fig. 3b) in the central router of a mesh NoC, experiencing a uniform-random traffic (Sect. 6). The steep curves reveal a high probability of a heavy congestion, as well as, a sudden and large change in the traffic, which can cause a high PSN in the NoC power delivery network.

(1) are presented for the twelve-state abstract model described in Sect. 6, and it is expected that these results are over-approximations for the abstract model with idle cycles in Sect. 7. For checking property (2), the latter model is used.

Figure 3a shows the cumulative probability for property (1) with several lower bounds. The formal model used for checking this property does not consider any empty buffer, in order to demonstrate the worst case scenario. At the architecture level, a high activity denotes the reception of three or more flits in one cycle. The steep slope of the curves indicates that the central router of a mesh NoC is likely to experience a heavy surge of the traffic load at a relatively short span of time. Such high load of traffic can engender a local hostspot in the network, which in turn, can lead to a large peak PSN.

Figure 3b depicts the cumulative probability for property (2) with several lower bounds. In this case, we consider empty buffers in the formal model of the central router. A high probability of such transitions within a short time span, as seen in this figure, denotes a bursty nature of the traffic encountered by the central router. As a result, there is a large inductive noise in the power delivery network of the NoC. Collectively, these two properties are pivotal in determining the minimum voltage guardband for the central router, because a more conservative guardband marks a power inefficient design, while a smaller one will be prone to intermittent timing errors in the NoC, aggravating its reliability.

Table 5 shows the peak memory usage and the total run-time reported by the MCSTA tool. Model checking property (2) requires significantly more memory than that for property (1). This is due to the increased complexity of the twenty-five-state abstract model depicting idle cycles over the twelve-state model, as well as, that checking property (2) requires more cycles to converge.

Table 5. Performance results.

Property	$k \cdot N$	Peak memory usage (MB)	Total run-time (s)
(1)	50	795	6.6
	100	1393	14.5
	150	2293	24.5
	200	2965	36.3
(2)	50	858	20.7
	100	2993	92.2
	150	6302	249
	200	11522	528.8

9 Conclusion

This paper presents a probabilistic model checking method for the reliability analysis for a generic central router of a large mesh NoC design under uniform random traffic loads. To combat the notorious state explosion problem,

abstract models have been derived based on critical observations of conflict resolution using the Round Robin scheduling mechanism. Probabilistic properties are derived by identifying the frequency of abrupt changes in router activities, which causes the inductive noise of PSN. To enable efficient checking, the clock cycle counter variable is set as transient and is treated as a reward annotation only to certain transitions, instead of part of the state space. Verification results reveal crucial PSN behaviors that allow the minimal voltage guardband to be determined for the central router, providing insights in NoC designs with improved reliability.

For future work, we plan to extend the central router model with increased number of channels and variants of Round Robin scheduling mechanisms. Incorporating routing protocols in the router model is also important, as it enables us to model a full NoC and better evaluate its reliability with respect to PSN. Additionally, we plan to investigate probabilistic predicate abstraction techniques to automate the abstraction and refinement of larger NoC models, and evaluate how they may affect the verification of PSN-related properties.

Acknowledgments. Arnd Hartmanns was supported by NWO VENI grant 639.021.754. Benjamin Lewis, Prabal Basu, Rajesh Jayashankara Shridevi, Koushik Chakraborty, and Sanghamitra Roy were supported in part by National Science Foundation (NSF) grants CAREER-1253024, CNS-1421022, and CNS-1421068. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. Ancajas, D.M., Chakraborty, K., Roy, S., Allred, J.M.: Tackling QoS-induced aging in exascale systems through agile path selection. In: IEEE/ACM International Conference on Hardware/software Codesign and System Synthesis, pp. 1–10 (2014)
2. Basu, P., Shridevi, R.J., Chakraborty, K., Roy, S.: IcoNoClast: tackling voltage noise in the noc power supply through flow-control and routing algorithms. IEEE Trans. VLSI Syst. **25**(7), 2035–2044 (2017)
3. Bhardwaj, K., Chakraborty, K., Roy, S.: An MILP based aging aware routing algorithm for NoCs. In: IEEE/ACM Design Automation & Test in Europe (DATE), pp. 326–331 (2012)
4. Bogdan, P., Marculescu, R.: Hitting time analysis for fault-tolerant communication at nanoscale in future multiprocessor platforms. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. (TCAD) **30**(8), 1197–1210 (2011)
5. Chaix, F., Avresky, D., Zergainoh, N.E., Nicolaidis, M.: A fault-tolerant deadlock-free adaptive routing for on chip interconnects. In: IEEE/ACM Design Automation & Test in Europe (DATE), pp. 909–912 (2011)
6. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Des. **43**(1), 61–92 (2013)
7. Chou, C.L., Marculescu, R.: FARM: fault-aware resource management in NoC-based multiprocessor platforms. In: IEEE/ACM Design Automation & Test in Europe (DATE), pp. 673–678 (2011)

8. Coste, N., Hermanns, H., Lantreibecq, E., Serwe, W.: Towards performance prediction of compositional models in industrial GALS designs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 204–218. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_18
9. Dahir, N., Mak, T.S.T., Xia, F., Yakovlev, A.: Modeling and tools for power supply variations analysis in networks-on-chip. IEEE Trans. Comput. (TC) **63**(3), 679–690 (2014)
10. Fang, L., Yamagata, Y., Oiwa, Y.: Evaluation of a resilience embedded system using probabilistic model-checking. arXiv preprint arXiv:1405.1703 (2014)
11. Gay, S., Nagarajan, R., Papanikolaou, N.: Probabilistic model-checking of quantum protocols. arXiv preprint arXiv:quant-ph/0504007 (2005)
12. Hahn, E.M., Hartmanns, A.: A comparison of time- and reward-bounded probabilistic model checking techniques. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 85–100. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_6
13. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods Syst. Des. **43**(2), 191–232 (2013)
14. Han, J., Gao, J., Jonker, P., Qi, Y., Fortes, J.A.: Toward hardware-redundant, fault-tolerant logic for nanoelectronics. IEEE Des. Test Comput. **22**(4), 328–339 (2005)
15. Hartmanns, A., Hermanns, H.: The Modest Toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
16. Hartmanns, A., Junges, S., Katoen, J.-P., Quatmann, T.: Multi-cost bounded reachability in MDP. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 320–339. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_19
17. Hosseini, A., Ragheb, T., Massoud, Y.: A fault-aware dynamic routing algorithm for on-chip networks. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2653–2656 (2008)
18. Klein, J., et al.: Advances in symbolic probabilistic model checking with PRISM. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 349–366. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_20
19. Kumar, J.A., Vasudevan, S.: Automatic compositional reasoning for probabilistic model checking of hardware designs. In: 2010 Seventh International Conference on the Quantitative Evaluation of Systems (QEST), pp. 143–152. IEEE (2010)
20. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic verification of Herman's self-stabilisation algorithm. Formal Aspects Comput. **24**(4–6), 661–670 (2012)
21. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. Inf. Comput. **205**(7), 1027–1077 (2007)
22. Milazzo, P.: Formal Modeling in Systems Biology: An Approach from Theoretical Computer Science. VDM Verlag (2008)
23. Mundhenk, P., Steinhorst, S., Lukasiewycz, M., Fahmy, S.A., Chakraborty, S.: Security analysis of automotive architectures using probabilistic model checking. In: Proceedings of the 52nd Annual Design Automation Conference, p. 38. ACM (2015)
24. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S., Gupta, R.: Using probabilistic model checking for dynamic power management. Formal Aspects Comput. **17**(2), 160–176 (2005)

25. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S.K.: Evaluating the reliability of defect-tolerant architectures for nanotechnology with probabilistic model checking. In: Proceedings of the 17th International Conference on VLSI Design, pp. 907–912. IEEE (2004)
26. Salamat, R., Khayambashi, M., Ebrahimi, M., Bagherzadeh, N.: A resilient routing algorithm with formal reliability analysis for partially connected 3D-NoCs. IEEE Trans. Comput. **65**(11), 3265–3279 (2016)
27. Shridevi, R.J., Ancajas, D.M., Chakraborty, K., Roy, S.: Tackling voltage emergencies in NoC through timing error resilience. In: ACM International Symposium on Low Power Electronic Devices (ISLPED), pp. 104–109 (2015)
28. Tsai, W.C., Zheng, D.Y., Chen, S.J., Hu, Y.H.: A fault-tolerant NoC scheme using bidirectional channel. In: IEEE/ACM Design Automation Conference (DAC), pp. 918–923 (2011)
29. Verbeek, F.: Formal verification of on-chip communication fabrics (2013)
30. Welke, S.R., Johnson, B.W., Aylor, J.H.: Reliability modeling of hardware/software systems. IEEE Trans. Reliab. **44**(3), 413–418 (1995)
31. Zhang, Z., Serwe, W., Wu, J., Yoneda, T., Zheng, H., Myers, C.: An improved fault-tolerant routing algorithm for a network-on-chip derived with formal analysis. Sci. Comput. Program. **118**, 24–39 (2016)
32. Zhang, Z., Serwe, W., Wu, J., Yoneda, T., Zheng, H., Myers, C.: Formal analysis of a fault-tolerant routing algorithm for a network-on-chip. In: Lang, F., Flammini, F. (eds.) FMICS 2014. LNCS, vol. 8718, pp. 48–62. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10702-8_4