Estimating Worst-case Latency of on-chip Interconnects with Formal Simulation

Freek Verbeek
Open University of the Netherlands
Radboud University, Nijmegen

Nikè van Vugt
Open University of the Netherlands

***Abstract*—Latency is a major issue in the design and validation of a Network-on-Chip (NoC). Various techniques for establishing latency bounds exist. Formal and mathematical methods, such as network calculus, can be used to analyze an NoC model. Simulation-based methods can be used to estimate latency bounds by exploring reachable states. Both have their advantages and disadvantages. This paper presents an approach that finds a middle ground between these two worlds. Our approach is based on simulation of high-level formal models. In contrast to traditional formal methods for worst-case latency, we do not require error-prone manual computation or the absence of cycles. In contrast to traditional simulation-based methods, we leverage the high level of abstraction to explore up to billions of states within a couple of hours. We apply our approach on an 8 core case study where a simple cache protocol runs on top of a ring-based Spidergon architecture. We show that deadlocks or starvations are easily found, and that for live networks a worst-case bound estimation can be produced within reasonable time.**

1. Introduction

Worst-case latency is a capricious matter: even a minor detail in the semantics of a minor element in a network may drastically impact whether some intricate and unexpected worst-case scenario can occur or not. Therefore, simulation is often done cycle-accurately and at a low level of abstraction (e.g., RTL). Cycle-accurate RTL simulation is a major research area in NoC validation [1], [2], [3]. However, finding the intricate scenario that causes worst-case latency can be tricky and may require simulation of many clock cycles.

This paper presents an approach that is based on the simulation of formal models of communication interconnects. Our approach is *high-level*: we purposefully do not simulate RTL but do simulation on a high level of abstraction. This enables fast simulation of large systems, e.g., an 8 core Spidergon with each core running a cache coherence protocol. Our approach is *formal*: the semantics of our model are completely formally defined in the Isabelle theorem prover. This enforces that each individual building block has concrete and executable semantics, and that the abstract blocks in our model can be extracted from RTL using the techniques described in [4], or that RTL can be generated from them. Finally, the building blocks of our model are *generic*: with various case studies we show that we can simulate routers, queues, virtual channels, credit-counters and state automata.

Simulation requires dealing with traffic patterns. Latency-Rate (LR) servers are a commonly accepted model of traffic injection and consumption [5] and are a basic concept in the *network calculus* [6]. The second part of our contribution consists of novel implementations for simulating network calculus traffic patterns in amortized cost $\mathcal{O}(1)$ per clock cycle. We use these algorithms to generate randomized traffic patterns – adhering to network calculus constraints – guided by heuristics that may quickly lead to a worst-case. All algorithms, models and Isabelle proofs are available online at http://www.cs.ru.nl/~freekver/fmcad17/.

Motivating example. Consider, e.g., Figure 4 from [7]. It presents a formal model of two communicating agents P and Q initiating requests and answering with responses. Each message type has its own virtual channel and a credit-based flow control ensures that a maximum number of packets can be en route at once. A packet that arrives at the opposite agent experiences a nondeterministic delay before it is sent back to its source.

The model can have a deadlock if the amount of credits is oversized, i.e., if sufficiently many packets can be injected to fill the cycle between the queues. That amount has to be at least $k + 2$ for such a deadlock to happen. However, for this deadlock to occur, a specific traffic pattern is necessary where the sources inject packets sufficiently fast, the sinks consume packets sufficiently slow, and the nondeterministic delay is sufficiently long.

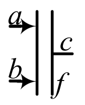
Even though that deadlock is reachable for such a traffic pattern, for many traffic patterns it is not. If the sources are bounded in their injection rate, if the sinks minimally provide some service rate, and if the nondeterministic delay is bounded, can the deadlock still occur? Our methodology can be used to show – for example – that the deadlock may occur in a setting where the sizes of the ingress queues are 2, the credit-counters are oversized to 4, the delay is maximally 10 clock ticks, the sinks are eager, and the sources inject packets at a maximum rate of 1 packet every 10 clock ticks. If we take the exact same setting but lower the maximum delay to 9 clock ticks, then the deadlock does not occur, even though the credit-counters remain oversized. In that case, the maximum latency is 23, i.e., once a packet has been injected it will maximally require 23 clock ticks before it arrives as a response at the sink. Both dx queues account for a clock tick. The packet waits 9 clocks ticks in iq_1 behind another packet. In the 11th clock cycle, the packet has reached the front of iq_1. It waits for 10 clock ticks: 9 due to the delay, 1 due to the fact that if in the last clock cycle a packet is injected, the merge may add 1 clock tick to that delay. This sums to 23: since the sinks are eager a packet will not experience any delay in its final ingress queue.

To summarize, the queue sizes, the ratio between the speeds of the sources, sinks and delays and the arbitration policy of the merges *can* be such that the credit-counters are actually not necessary, since the deadlock they are meant to prevent cannot occur anyway. A minor change in any of these elements can, however, significantly increase latency or cause a deadlock.

We have simulated 10^8 clock cycles (about $5.5 \cdot 10^7$ packets) with *random* traffic that adheres to these constraints without finding this scenario. The scenario does not occur either when traffic is regular, e.g., when the sources inject *exactly* each 10th clock tick. However, by simulating a high-level formal model of the example and using traffic patterns guided by some simple heuristics, we were able to find it within a couple of hours.

2. Executable communication interconnect modeling

We model communication networks *formally*, i.e., in such a way that the semantics are defined mathematically, and *executable*. An example of a language that allows formal and executable modelling is xMAS [7]. Figure 1 presents an example of an xMAS primitive: the join. This primitive blocks its incoming packets until at both inputs a packet has arrived, at which point it will use function f to produce a packet at the output. The language xMAS provides various primitives such as merges (for arbitration), switches (for routing), sources and sinks.



(a) The xMAS join

$$
\begin{aligned}
c{\cdot}\text{irdy} &= a{\cdot}\text{irdy} \wedge b{\cdot}\text{irdy}\\
c{\cdot}\text{data} &= f(a{\cdot}\text{data}, b{\cdot}\text{data})\\
a{\cdot}\text{trdy} &= c{\cdot}\text{trdy} \wedge b{\cdot}\text{irdy}\\
b{\cdot}\text{trdy} &= c{\cdot}\text{trdy} \wedge a{\cdot}\text{irdy}
\end{aligned}
$$

(b) Formal semantics

Figure 1. Example of an xMAS primitive

Our modelling language is basically a generalization of xMAS. Even though xMAS is executable, the primitives are too fine-grained for efficient execution. For example, a simple XY router takes about 17 primitives and modelling virtual channels is inelegant [8]. Moreover, modelling, e.g., cache protocols with xMAS is infeasible.

In our generalization, we model communication networks using generic building blocks. Blocks can be fine-grained primitives, such as arbiters or joins, but can also be abstract statefull blocks such as routers, credit-counters or a cache protocol. Each block is efficiently executable. A communication network is then modelled as a composition of executable building blocks.

We use Isabelle/HOL [9] to define the notion of “building block”, and how a communication network is composed out of these blocks.

2.1. Definition of a generic building block

A communication network consists of blocks with an ID of type *'block* connected by a set of channels with IDs of type *'chan*[1]. Each channel has an initiator and a target, and three wires, *irdy*, *trdy* and *data*. An *irdy* wire indicates the initiator is ready to transmit data, a *trdy* wire indicates the target is ready to receive, and the *data* wire is used for data transmission. We use $c{\cdot}\text{w}$ to denote wire w of channel c.

datatype $'$chan wire $=$ irdy $'$chan $|$ trdy $'$chan $|$ data $'$chan

The wires can have either Boolean values (in case of irdy/trdy), a data value, or be undefined. We use the term *color* to refer to the data, in the same fashion as colored Petrinets. We assume the existence of type *'color* that represents the set of colors.

datatype $'$color wire-value $=$ B bool $|$ C $'$color $|$ Undef

A block assigns wire values to certain wires: the irdy and data wires of its outgoing channels, and the trdy wires of its incoming channels. It does so, based on the wire values of its *environment*: the trdy wires of its outgoing channels, and the irdy and data wires of its incoming channels. A *valuation* is used to store wire values. It is a set of pairs of wires and wire values.

type-synonym $('$chan$,'$color$)$ val $=$
    $('$chan wire $\times$ $'$color wire-value$)$ set

A block is defined by two functions. The first function, *eval*, takes as input the current *internal state* of the block and a valuation. It computes new values for wires and adds these to the given valuation. For example, in case of a join (see Figure 1b), if the given valuation contains values for wires $a{\cdot}$irdy and $b{\cdot}$irdy, then the semantics of the join are able to compute a new value for wire $c{\cdot}$irdy, and that value is added to the valuation. The second function, *tick*, provides the set of possible next internal states, given the current internal state of the block and the current valuation. In case of a stateless block, this function can return the empty set.

record $('$chan$,'$color$,'$istate$)$ block $=$
    eval :: $'$istate $\Rightarrow$ $('$chan$,'$color$)$ val $\Rightarrow$ $('$chan$,'$color$)$ val
    tick :: $'$istate $\Rightarrow$ $('$chan$,'$color$)$ val $\Rightarrow$ $'$istate set

The complete *state* σ of the communication network is then simply a map of blocks to their internal state. Note that the wires are not part of the state: they are combinatorial.

type-synonym $('$block$,'$istate$)$ state $=$ $'$block $\Rightarrow$ $'$istate

1. Types preceded by an apostrophe are polymorphic, e.g., we allow any set of block IDs and any set of channel IDs.

2.2. Composition of building blocks

To build a network out of these blocks, one has to connect them using channels. A properly composed network should satisfy various properties, such as: two channels may not be connected to the same input of a block, there may be no combinatorial cycles, there may be no dangling channels, and for each wire it should be possible to derive a unique and properly typed wire value. Moreover, the algorithm for computing a valuation for the current state can be quite contrived: it should start with blocks that can provide wire values solely based on their current internal state (such as queues, sinks and sources) and then propagate these values to other blocks. That propagation is both forward (in case of irdy/data wires) and backwards (in case of trdy wire). Proving termination of this propagation is not possible in the generic case: for example, if there is a combinatorial cycle, then this propagation will not terminate.

We will show that all these problems can be dealt with at once, by formalizing the derivation of wire values as a least fixed point.

From now on, we assume the existence of a function *block* that provides the set of blocks (modelled as a map of block IDs to blocks). Moreover, we assume that for each block b, derivation of the wire values is monotonically increasing. This ensures that the valuation can only increase, i.e., if more wire values are known, then more new wire values can be computed. This assumption is formulated by requiring for each block b and for each internal state x of that block, monotonicity of its *eval* function. We use an Isabelle locale to introduce a context in which such a function *block* exists.

locale monotone-blocks $=$
  fixes block :: $'$block $\Rightarrow$ ($'$chan, $'$color, $'$istate) block
  assumes mono (eval (block b) x)

We now define a function *deriveWires* that computes, given a state σ, a valuation for all wires. This valuation is computed by the following least fixpoint:

$$deriveWires\ \sigma \equiv \mu Z \cdot eval\ (block\ b)\ (\sigma\ b)\ Z \subseteq Z$$

The valuation derived from a state σ is thus the smallest valuation Z such that Z contains all wire values computed by any block b given its current internal state (σ b).
We can now formally define the *step* function of a composition of building blocks. It is nondeterministic and returns, given the current state σ, a set of next states.

definition step ::
  ($'$block, $'$istate) state $\Rightarrow$ ($'$block, $'$istate) state set
  where step $\sigma \equiv$
  $\{\sigma' . \forall$ b . σ' b $\in$ tick (block b) (σ b) (deriveWires σ)$\}$

For all blocks b, function *tick* is used to compute the next internal state of that block. Function *tick* is given the current internal state of that block (σ b) and the current valuation of wires (*deriveWires* σ). State σ' is a next state if and only if all blocks b have "ticked", i.e., moved to some next internal state.

An Executable Communication Interconnect Model (ECIM) is defined as a set of blocks, connected in such a way that in each state there is a unique valuation for all wires. This assumption at once takes care of all issues mentioned at the beginning of this section. For example, it eliminates combinatorial cycles, since such a cycle would prevent function *deriveWires* to assign a value to the wires participating in that cycle. We extend the existing locale by adding the assumption that for each wire w there should be a unique value v derived by function *deriveWires*.

locale ECIM $=$ monotone-blocks $+$
  assumes $\exists!$ v . (w, v) $\in$ deriveWires σ

Within the ECIM context we can formulate an LTL logic and prove all kinds of sanity theorems, such as:

1) If each block is persistent (e.g., will maintain a high irdy signal once it is set, until a transfer occurs [10]) then the network as a whole is persistent.
2) If each block is correctly typed, e.g., does not assign colors to irdy/trdy wires, then this property holds always globally.
3) Each xMAS primitive is an ECIM building block. We provide a shallow embedding of a DSL that can be used to model xMAS-like primitives into ECIM.
4) Block- and idle equations [10] can be proven correct, e.g., the incoming channel of a queue is permanently blocked if and only if the queue is full and its outgoing channel is permanently blocked.

2.3. Some ECIM building blocks

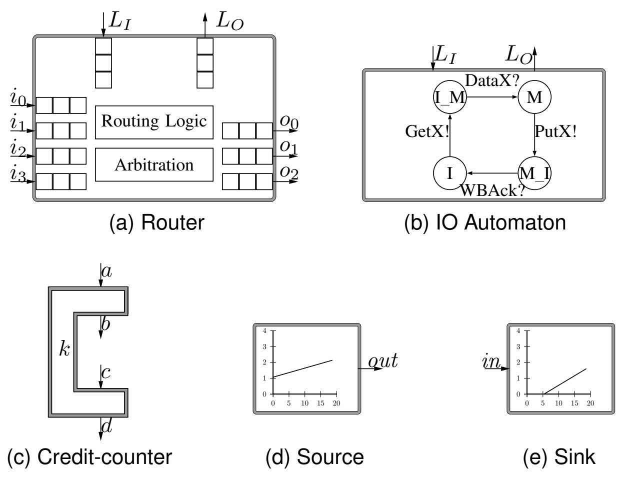


Figure 2. ECIM Building Blocks

Figure 2 shows some of the ECIM blocks that we use. First, we supply a *router* that has n inputs and m outputs. Based on a given routing logic, incoming messages will either be forwarded to some output, or – if they have arrived at their destination – be sent to the local out-port. Injection of messages can occur via the local in-port. The router is statefull: both the queues and the arbiter who resolves contention are part of the state.

Secondly, an *IO automaton* can be used to model protocols. Injection of a message (!) causes the L_O·irdy to be set to high and L_O·data to be set to, e.g., a GetX. Consumption of a message (?) is done by setting L_I·trdy high whenever L_I·data is, e.g., a DataX.

A *credit-counter* can be used to limit the amount of incoming packets "between" channels b and c. It is statefull, since it stores some integer less than k that indicates the number of counted messages.

Finally, *sources* and *sinks* can be used to generate traffic patterns. The next section discusses their implementation.

For all these blocks, we have implemented monotonic *eval* functions that determine when they consume and inject messages. Also, for each statefull block, we implement a *tick* function.

3. ECIM simulator

Assuming we have an ECIM built of efficiently executable blocks, a simulation algorithm is easily defined. We have implemented the pseudo code of Algorithm 3 in Java. The current state σ is an object with a method *step*. Each block is an object that implements a *tick* function that computes a next state. After each step, the clock is incremented.

```
function $\sigma$.STEP
    deriveWires
    for all block $b$ do
        $b$.tick
    end for
    t++
end function
```

Figure 3. ECIM simulation

In order to simulate a communication network, however, it is additionally required that traffic patterns can be efficiently simulated. For the case of *average* case latency analysis, one might use Poisson distributions to simulate random injections at sources and random consumptions at sinks. This has several major drawbacks when *worst-case* latency is considered. Firstly (regarding the sources) this does not accurately reflect the burstiness present in common traffic patterns, such as multimedia components [11]. Secondly, even with a low Poisson rate, the network might be flooded with traffic, even though the behavior of the source would not allow this. Thirdly (regarding the sinks) this does not provide a lower bound to the amount of consumptions: it theoretically allows for sinks to be dead for any period of time, thereby producing any worst-case latency as long as simulation is continued sufficiently long.

To this end, we use concepts of the network calculus to provide simulations for worst-case latency analysis [5], [6], [12]. We provide an efficient implementation to simulate *linear arrival curves*. Such curves are widely used to model traffic flows in a network and provide bursty traffic. For the sinks, we provide an efficient implementation of *linear service curves* to model the consumption behavior of sinks. Service curves (or LR servers) model an extensive class of network servers.

We only provide the definitions of the network calculus that are relevant to our simulator; for an introduction and more in-depth details, see [6], [12]. Network calculus concerns flows of traffic in the network. An *input flow* is characterized by function $R(t)$ which returns, given the current time t, the total cumulative amount of incoming traffic in the interval $[0, t]$. An *output flow* is characterized by cumulative function $R^*(t)$. We use $f_{\texttt{nc}}$ to denote the noncumulative version of a cumulative function f, i.e.,:

$$f(t) = \sum_{t' \leq t} f_{\texttt{nc}}(t')$$

3.1. Source simulation with linear arrival curves

An arrival curve can be defined by a function α that provides an upperbound to the amount of traffic.

Definition 1. *A source injects traffic adhering to arrival curve α, if and only if, for any time slot t:*

$$\forall 0 \leq s \leq t \cdot R(t) - R(s) \leq \alpha(t - s)$$

A linear arrival curve is defined by a natural number *burst* b and a real number *arrival rate* r: $\alpha(t) = rt + b$. Here b is a measure of burstiness, i.e., the amount of traffic that can be injected at once, and r is a maximally sustainable injection rate.

We provide an implementation that simulates linear arrival curves in time $\mathcal{O}(1)$ (see Figure 4). In this algorithm, we use the following variables:

Rt	the cumulative flow up to the current time slot (note that we only need to store the total flow up to the last time slot, and not per time slot).
$Rt_{\texttt{nc}}$	the non-cumulative flow in the current time slot, i.e., the number of injections
Mt	the current minimum value of $\alpha(t-s)+R(s)$ for all $s < t$

```
Require: $t > 0 \longrightarrow Mt = \min_{s<t} [\alpha(t-s) + R(s)]$
 1: function COMPUTEARRIVAL
 2:     if $t == 0$ then
 3:         $Rt_{\texttt{nc}} \leftarrow 0$
 4:     else
 5:         choose $Rt_{\texttt{nc}} \leq \lfloor Mt + r \rfloor - Rt$
 6:     end if
 7:     $Rt \mathrel{+}= Rt_{\texttt{nc}}$
 8:     $Mt \leftarrow t == 0\ ?\ b\ :\ min(Mt + r, b + Rt)$
 9:     t++
10:     return $Rt_{\texttt{nc}}$
11: end function
Ensure: $Mt = \min_{s<t} [\alpha(t-s) + R(s)]$
```

Figure 4. Simulation for linear arrival curves

After each call of function *computeArrival*, variable $Rt_{\mathtt{nc}}$ provides the number of packets that can be injected by the source in the current time slot. At Line 5, this value is computed by randomly choosing a value bounded by the current minimum value of $\alpha(t-s)+R(s)$, plus the rate, minus the current cumulative flow. Line 7 then updates variable Rt, so that it contains the current cumulative flow. Line 8 recomputes variable Mt, to preserve the invariant that variable Mt stores the minimum value of $\alpha(t-s)+R(s)$ for all $s<t$ when t is incremented at Line 9.

We prove correctness of this algorithm, by showing that Definition 1 holds invariably.

Theorem 1. *Let $R(0)=0$ and let $R(t+1)$ be the value of variable Rt after the tth call of function computeArrival. At any time slot t, we have:*

$$\forall 0 \leq t' \leq t \cdot R(t) - R(t') \leq \alpha(t-t')$$

The proof is omitted, but an Isabelle formalization can be found online.

3.2. Sink simulation with linear service curves

Service curves model a server that provides a minimal amount of service. When incoming traffic arrives at a sink, the service curve may model a delay before packets are consumed, but eventually the sink will provide a service with some rate (as long as there is sufficient incoming traffic). A linear service curve β is defined by a natural number *delay* d and a real number *service rate* r:

$$\beta(t) = \begin{array}{ll} 0 & t \leq d \\ r(t-d) & t > d \end{array}$$

The definition of a service curve is based on the notion of min-plus convolution.

Definition 2. *Let f and g be two weakly increasing functions. The min-plus convolution of f and g, notation $f \otimes g$, is defined as:*

$$(f \otimes g)(t) = \underset{0 \leq s \leq t}{inf} \; [f(s) + g(t-s)]$$

Definition 3. *A sink consumes traffic adhering to service curve β for the cumulative input flow R, if and only if, for any t:*

$$R^*(t) \geq (R \otimes \beta)(t)$$

Our algorithm is based on Propositions 1.3.1 and 1.3.2 from [6]. We here provide a corollary of these propositions:

Lemma 1. *Assume β is convex. There exists a weakly increasing function $\tau :: \mathbb{N} \mapsto \mathbb{N}$ such that for any time slot t:*

$$R^*(t) \geq R(\tau(t)) + \beta(t - \tau(t))$$

It is crucial that function τ is weakly increasing, and we leverage that fact to compute a lower bound for the service for the current time slot in amortized time $\mathcal{O}(1)$.

Remark. *For some service curves, the value of τ is computable: for a constant rate server without delay, and for strict service curves (modelling work-conserving sinks), the value of τ is the beginning of the last busy period. For linear service curves with delay that does not hold; the value of τ is unknown [6]. In the proof of Theorem 2 we will prove that in each time slot, only under a certain condition is it necessary to search for a new value for τ, and that the size of the range in which we have to search is constant.*

In the algorithm, we use the following variables:

τ	the current value such that $\tau < t$ and $R(\tau) + \beta(t-1-\tau)$ is minimal
$Rshifted$	a linked list storing values of function R, in such a way that $R(t) = Rshifted(t-\tau)$. The last value in this list always stores the total cumulative input flow $R(t)$, the first value stores $R(\tau)$. Lemma 1 shows that we can forget any value prior to τ.
Rt^*	the cumulative output flow up to the current time slot, i.e., the total number of consumptions
$Rt^*_{\mathtt{nc}}$	the non-cumulative output flow, i.e., the number of consumptions in the current time slot

Function *computeService* takes as input the current non-cumulative amount of incoming traffic in the current time slot t. It first stores that value in list $Rshifted$, by adding a new value to the end of that list (Lines 2 to 6). Then, we determine the value of the min-plus convolution by finding a value τ' for which $R(\tau')+\beta(t-\tau')$ is minimal. Currently, variable τ stores the value for which $R(\tau) + \beta(t-1-\tau)$ is minimal. Only if $t - \tau > d$ (Line 8) it is necessary to search for a new value for τ'. The search can be limited to a certain range (Lines 9 to 13). Otherwise, τ' remains the same as τ, since $R(\tau) + \beta(t-\tau)$ remains minimal.

If a new value for τ is found, we pop the elements in list $Rshifted$ until we have reached that value (Lines 15 to 18). Line 19 then computes the minimum amount of service that is to be provided in the current time slot. The actual service is then some random value greater than that amount, but bounded by the amount of incoming traffic (Lines 20 and 21). Finally, the current cumulative amount of outgoing traffic (i.e., the total cumulative amount of consumptions) is updated, and the clock is increased.

We prove correctness of this algorithm, by showing that Definition 3 holds invariably.

Theorem 2. *Let $R^*_{\mathtt{nc}}(0) = 0$ and let $R^*_{\mathtt{nc}}(t+1)$ be the value of variable $Rt^*_{\mathtt{nc}}$ after the tth call of function computeService. At any time slot t, we have:*

$$R(t) \geq R^*(t) \geq (R \otimes \beta)(t)$$

Proof: We first prove that for any n, the postcondition holds after the nth call of the algorithm:

$$R(\tau) + \beta(t-1-\tau) = \underset{s<t}{min} \; [R(s) + \beta(t-1-s)]$$

```
Require: $R(\tau)+\beta(t-1-\tau) = \min_{s<t} [R(s)+\beta(t-1-s)]$
 1: function COMPUTESERVICE(int $Rt_{\mathtt{nc}}$)
 2:     if $t == 0$ then
 3:         $Rshifted.add(Rt_{\mathtt{nc}})$
 4:     else
 5:         $Rshifted.add(Rt_{\mathtt{nc}} + Rshifted.last())$
 6:     end if
 7:     $\tau' \leftarrow \tau$
 8:     if $t - \tau > d$ then
 9:         for $s \leftarrow t-d$ to $t$ do
10:             if $f(s) \leq f(\tau')$ then
11:                 $\tau' \leftarrow s$
12:             end if
13:         end for
14:     end if
15:     while $\tau \neq \tau'$ do
16:         $Rshifted.removeFirst()$
17:         $\tau$++
18:     end while
19:     $min \leftarrow max(0, f(\tau) - Rt^*)$
20:     $max \leftarrow Rshifted.last() - Rt^*$
21:     choose $Rt^*_{\mathtt{nc}}$ st.: $min \leq Rt^*_{\mathtt{nc}} \leq max$
22:     $Rt^*$ += $Rt^*_{\mathtt{nc}}$
23:     $t$++
24:     return $Rt^*_{\mathtt{nc}}$
25: where
26:     $f(x) = Rshifted[x-\tau] + \beta(t-x)$
27: end function
Ensure: $R(\tau)+\beta(t-1-\tau) = \min_{s<t} [R(s)+\beta(t-1-s)]$
```

Figure 5. Simulation for linear service curves

The proof is by induction over n. For the base case, after the 0th call of the algorithm, we have $\tau = 0$ and $t = 1$ and the property holds trivially.

For the inductive case, the induction hypothesis is the precondition. We show that at Line 18, the algorithm has found a value for τ such that:

$$R(\tau) + \beta(t-\tau) = \min_{s \leq t} [R(s) + \beta(t-s)]$$

This implies that after incrementing time t (Line 23), the postcondition holds.

Assume there exists some $\tau' \leq t$, such that (A):

$$R(\tau') + \beta(t-\tau') < R(\tau) + \beta(t-\tau)$$

We first assume the case where (B): $t - \tau > d$ (Line 8). Assume (C): $t - \tau' > d$. Then:

$$\begin{aligned}
R(\tau') + \beta(t-\tau') &< R(\tau) + \beta(t-\tau) && \text{(A)}\\
R(\tau') + \beta(t-\tau'-1) + r &< R(\tau) + \beta(t-\tau) && \text{(C)}\\
R(\tau') + \beta(t-\tau'-1) + r &< R(\tau) + \beta(t-\tau-1) + r && \text{(B)}\\
R(\tau') + \beta(t-\tau'-1) &< R(\tau) + \beta(t-\tau-1)
\end{aligned}$$

The induction hypothesis then implies that $\tau' = t$. This implies that $d < 0$, and thus assumption (C) is false. This implies that $t - \tau' \leq d$. Hence: $t - d \leq \tau' \leq t$. Thus, in case (B), only this range of values has to be searched for candidates for a new value of τ (if any).

Case	Latency	Time (µs)	#cycles	#packets
A	14	1.5	10^9	$3.0 \cdot 10^8$
B	23	23	10^9	$1.8 \cdot 10^8$
C	∞	110	$< 10^3$	N/A
D	∞	200	$< 10^4$	N/A
E	91	187	10^7	$1.5 \cdot 10^6$

TABLE 1. SIMULATION RESULTS.

Now we consider the case where (B) is false. By Lemma 1, we have (D): $\tau' \geq \tau$. Then:

$$\begin{aligned}
R(\tau') + \beta(t-\tau') &< R(\tau) + \beta(t-\tau) && \text{(A)}\\
R(\tau') &< R(\tau) && \neg\text{(B)} \wedge \text{(D)}
\end{aligned}$$

However, since R is weakly increasing, this is a contradiction. Hence, if (B) is false, then (A) is false, meaning that there exists no τ' such that $R(\tau') + \beta(t-\tau') < R(\tau) + \beta(t-\tau)$. Hence $R(\tau) + \beta(t-\tau) = \min_{s \leq t} [R(s) + \beta(t-s)]$.

From this inductive proof, the theorem follows. It has been proven that at Line 18, the algorithm has found a value for τ such that $R(\tau) + \beta(t-\tau)$ is minimal $\forall \tau \leq t$. Thus, the number of consumptions in the current time slot $Rt^*_{\mathtt{nc}}$ is minimally that value minus the current cumulative incoming traffic, and maximally the current backlog. □

4. Experimental results

We present 5 case studies. All results (see Table 1) have been obtained on a 1,6 GHz Intel Core i5 (4 cores). For each case study we have twice run 4 simulations in parallel. Column "Latency" shows the *measured* maximum latency. Column "Time" shows the average time of simulating 4 clock cycles on 4 cores in microseconds. The last two columns show the number of clock cycles and the number of packets *per simulation*.

A) Source, queue, queue, sink. Figure 6 shows the first 80 clock cycles of in- and output flows for a simple example with one source, two consecutive queues q_0 and q_1 (resp. sizes 5 and 10) and a sink. In this example, the sink has a delay of 4 but is eventually sufficiently fast to consume the injected input flow (i.e., the service curve "overtakes" the arrival curve). We have modelled a sink that can consume from q_1 more than one packet per clock tick, so that there is no gap between the intended output flow and the actual output flow. The source injects at most 1 packet per clock tick and can thus lag behind the intended arrival flow. The maximum measured latency in the first 80 clock ticks is 9, measured between clock ticks 9 to 18.

We have run eight simulations of 10^9 clock ticks (taking about 25 minutes per four) and measured a maximum latency of 14 (see Table 1).

B) Two agents. Section 1 presents the "two agents" example of Intel. We have modelled the xMAS example using more abstract ECIM blocks such as counters and delays. We enabled the following heuristic: let the sources and sinks remain irregular but maximize the nondeterministic delay.

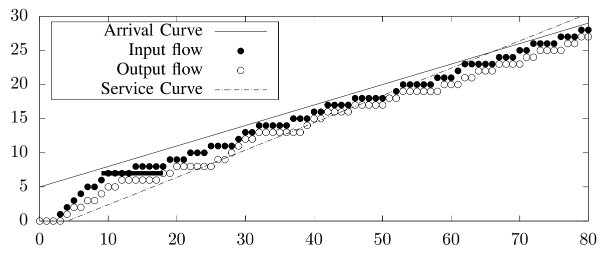


Figure 6. Source with $\alpha(5, 0.3)$, two queues, and a sink with $\beta(4, 0.4)$

All 8 simulations found a worst-case scenario that causes the 23 latency. On average, the worst-case scenario is found after about 300 million clock cycles. If the nondeterministic delay is increased to 10, simulation quickly finds a deadlock.

Spidergon case study. Figure 7 presents a Spidergon architecture with 8 nodes [13]. Packets consist of 2 bits that store a message type (GetX, DataX, PutX, or WBAck) and 3 bits that store the address of a node. The routing logic is *across first* meaning that if the shortest route to a packet requires an across channel, that channel is taken first. At each router, whenever two packets compete for the same output, a FIFO arbiter decides which packet can proceed and which not.

The protocol is a simple directory-based MI cache protocol. Caches 0 to 6 inject GetX messages to request exclusive access to a cache block. A GetX packet is accompanied by the address of the node that injects it. Its destination is always 7 and thus need not be stored in the packet. After injection of a GetX, a cache waits for data in state I_M. When a DataX is received, the cache moves to the M state where is has exclusive access to that block. To write back data, a PutX is injected, again accompanied with the address of the injecting node, and the cache moves to state M_I. Once a WBAck is received, the cache returns to state I.

Nodes 0 to 6 run this protocol; node 7 is a directory. It has two states I and M. In state I, upon receiving a GetX from node n, it injects a DataX with as destination n and moves to state M. In state M, upon receiving a PutX from node n, it injects a WBAck with destination n and returns to state I.

Finally, as shown in Figure 7, the cache protocol is connected to a source and a sink. Injection of GetX and PutX is done only when the source has a high irdy signal. We set the injection rate of the source to a Poisson distribution with $\lambda = 0.25$. This models that on average it takes 4 clock ticks for a cache to inject a packet. Dually, consumption of a packet is done only when the sink has a high trdy signal. We have set the sink to a linear service curve with $d = 4$ and $r = 0.5$, modelling that consumption of a packet can maximally be delayed 5 clock ticks, but after 6 clock ticks minimally one packet is consumed.

Note that this routing logic for the Spidergon architecture suffers from a routing deadlock. If simultaneously each node n injects packets destined for $n + 2 \mod 8$, then the clockwise circle becomes full and a circular wait occurs. The

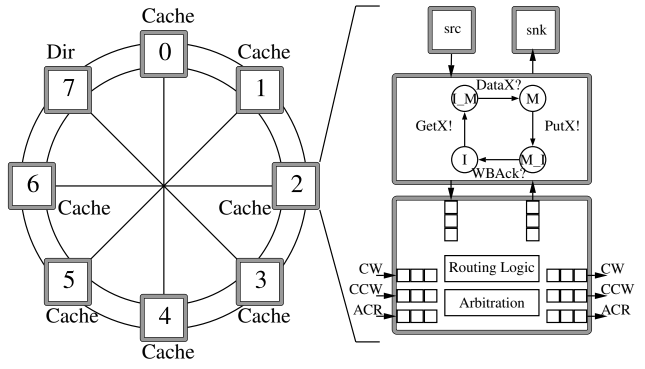


Figure 7. Spidergon Architecture with MI protocol

protocol prohibits that injection pattern and therefore – in this setup – *no routing deadlock* can occur. Moreover, there is *no protocol deadlock*. However, cross-layer deadlocks may occur.

We measure the latency for a round trip of a GetX to return back to its source as a DataX. Note that this means that the measured latency includes cases where the GetX has to wait since another cache is currently owner. We modelled three cases C, D, and E described below. For the first two cases, runs are found in which the latency grows – seemingly – to infinity, i.e., for a long period of time the latency grows linearly with time. For the third case, the maximum measured latency is 91.

C) With deadlock. Consider the case where (at least) two caches inject a GetX, say caches n and m. The directory receives 7 packets and is only able to accept the first one, say from node n. It injects a DataX and moves to state M. The cache receives the data, and moves to state M as well. The only way progression is possible, is when cache n moves to state M_I by injecting PutX. However, the local queue from router 7 to the directory currently stores a GetX packet from cache m. Since queues are FIFO, the PutX cannot overtake the GetX and a cross-layer deadlock occurs.

D) Recycling packets: no deadlock, but starvation. The deadlock can be prevented by *recycling* packets: whenever a protocol cannot consume a packet it will be recycled to the end of the queue. This allows packets that can be consumed to overtake others. Indeed, this prevents the deadlock, if the size of the local queue from the router to the node is large enough. Worst case, that queue contains a PutX from the current sharer and 6 other GetX packets. Therefore the size of the queue has to be 8 to prevent the deadlock. Recycling, however, does introduce a starvation scenario. Since the packets in the local queue are no longer handled in FIFO order, it might be the case that a GetX is continuously overtaken by other GetX packets. This starvation scenario occured on average within 10.000 clock cycles.

E) Adding a VC: no deadlock or starvation. The starvation can be resolved by splitting the local queue of the directory into two virtual channels: one for GetX packets and one for the others. Moreover, GetX packets are not recycled, and thus handled in FIFO order. This resolves both the deadlock and the starvation. Note that to prevent both the deadlock and the starvation, only *one* virtual channel is required: the remainder of the network needs none. The size of the virtual channel should be sufficient to store six packets.

5. Related work

Simulation of NoCs is an extensive research area: various cycle-accurate tools exist that target heterogeneous and generic NoC architectures. BookSim [2] is a detailed, router-based simulator for NoCs, whose underlying network model has been validated to RTL implementations. OMNeT++ [1] is a framework where generic and high-level blocks can be simulated. It has been used to simulate among others the Spidergon architecture [14]. Generally, these tools generate synthetic traffic patterns. In contrast, our approach allows modelling of the cache coherence protocol deployed by the nodes, to more accurately model realistic traffic patterns. Also, we simulate a formal model, which has been defined in such a way that blocks can be derived from Verilog, or each block can be used to generate Verilog [4].

Zhao and Lu use network calculus to analyse xMAS models and use RTL simulation to verify tightness of their bounds [15]. In [12], Zhao uses the tool Simulink of MathWorks to simulate xMAS. Zhao derives Verilog from xMAS and uses that to find a worst-case latency bound for the two agents example. The presented results show simulation of about 10.000 packets. Since we simulate a high-level model instead of Verilog, we are able to simulate significantly more packets.

Salamat et al. use Noxim [3] to analyse both latency- and fault-tolerance of a 3D chip architecture [16]. Noxim takes more properties into account such as a power and thermal model. In contrast, our approach solely focusses on worst-case latency, which allows for more efficient execution.

6. Conclusion

This paper presents an approach for finding worst-case latency estimates based on simulation of formal models. The models – whose semantics have been formalized in Isabelle/HOL – consist of generic building blocks. Each block is high-level and therefore efficiently executable. The formal semantics ensure that the blocks can be implemented in Verilog, or that they can be extracted from Verilog.

We use novel implementations of simulating latency-rate servers to generate bursty traffic patterns. We implement some simple heuristics aimed at finding the worst case, such as maximizing delays whenever possible. We did, however, find that for some examples irregular traffic was necessary to get to the worst-case scenario, and that only after simulation of millions of packets the worst case was found.

In the near future, we aim to address the gap between the Isabelle model and the implementation, by using Isabelle's code generation to generate an implementation from the model. We expect this will impact performance, but it will enable a formally verified simulator for NoCs, where the model that is simulated can be used for theorem proving or model-based testing.

References

[1] A. Varga, "OMNeT++," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Güneş, and J. Gross, Eds., 2010, pp. 35–59.

[2] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 86–96.

[3] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2015, pp. 162–163.

[4] S. J. Joosten and J. Schmaltz, "Automatic extraction of micro-architectural models of communication fabrics from register transfer level designs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 1413–1418.

[5] D. Stiliadis and A. Varma, "Latency-rate servers: a general model for analysis of traffic scheduling algorithms," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 5, pp. 611–624, 1998.

[6] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001.

[7] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[8] F. Verbeek, P. M. Yaghini, A. Eghbal, and N. Bagherzadeh, "Advocat: Automated deadlock verification for on-chip cache coherence and interconnects," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 2016, pp. 1640–1645.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[10] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, *Verifying Deadlock-Freedom of Communication Fabrics*, 2011, pp. 214–231.

[11] A. E. Kiasari, Z. Lu, and A. Jantsch, "An analytical latency model for networks-on-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 1, pp. 113–123, 2013.

[12] X. Zhao, "Network on Chip: Performance Bound and Tightness," Ph.D. dissertation, KTH Royal Institute of Technology, 2015.

[13] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra, "Spidergon: a novel on-chip communication network," in *System-on-Chip, 2004. Proceedings. 2004 International Symposium on*. IEEE, 2004, p. 15.

[14] L. Bononi and N. Concer, "Simulation and analysis of network on chip architectures: ring, spidergon and 2D mesh," in *Proceedings of the conference on Design, automation and test in Europe: Designers' forum*. European Design and Automation Association, 2006, pp. 154–159.

[15] X. Zhao and Z. Lu, "Per-flow delay bound analysis based on a formalized microarchitectural model," in *2013 Seventh IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, April 2013, pp. 1–8.

[16] R. Salamat, M. Khayambashi, M. Ebrahimi, and N. Bagherzadeh, "A resilient routing algorithm with formal reliability analysis for partially connected 3D-NoCs," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3265–3279, Nov 2016.