Science of Computer Programming 118 (2016) 24–39

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

An improved fault-tolerant routing algorithm for a Network-on-Chip derived with formal analysis [☆]

Zhen Zhang [a,*], Wendelin Serwe [b,c,d], Jian Wu [e], Tomohiro Yoneda [f], Hao Zheng [g], Chris Myers [a]

[a] *Dept. of Elec. & Comp. Eng., Univ. of Utah, Salt Lake City, UT, USA*
[b] *Inria, France*
[c] *Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France*
[d] *CNRS, LIG, F-38000 Grenoble, France*
[e] *Toshiba America Electronic Components, Inc., San Jose, CA, USA*
[f] *National Institute of Informatics, Tokyo, Japan*
[g] *Dept. of Comp. Sci. & Eng., Univ. of S. Florida, Tampa, FL, USA*

ARTICLE INFO

Article history:
Received 20 January 2015
Received in revised form 23 December 2015
Accepted 8 January 2016
Available online 14 January 2016

Keywords:
Fault-tolerant routing
Formal methods
Model checking
Network-on-chip
Process calculus

ABSTRACT

A fault-tolerant routing algorithm in Network-on-Chip (NoC) architectures provides adaptivity for on-chip communications. Adding fault-tolerance adaptivity to a routing algorithm increases its design complexity and makes it prone to deadlock and other problems if improperly implemented. Formal verification techniques are needed to check the correctness of the design. This paper describes the discovery of a potential livelock problem through formal analysis on an extension of the link-fault tolerant NoC architecture introduced by Wu et al. In the process of eliminating this problem, an improved routing architecture is derived. The improvement simplifies the routing architecture, enabling successful verification using the CADP verification toolbox. The routing algorithm is proven to have several desirable properties including deadlock and livelock freedom, and tolerance to a single-link-fault.

1. Introduction

Cyber-physical systems (CPS) have ubiquitous applications in many safety critical areas such as avionics, traffic control, robust medical devices, etc. As an example, the automotive industry makes active use of CPS: modern vehicles can have up to 80 *electronic control units* (ECUs), which control and operate everything from the engine and brakes to door locks and electric windows. Currently, each ECU is statically tied to its specific sensors and actuators. This means that processing power between different ECUs cannot be shared, which degrades the performance of the chip due to imbalanced workload

[☆] This material is based upon work supported by the National Science Foundation under grants CNS-0930510 and CNS-0930225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Part of this work was performed during a visit of the first author at the Inria Grenoble–Rhône-Alpes research centre. A subset of the experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

[*] Corresponding author.
E-mail addresses: zhen.zhang@utah.edu (Z. Zhang), Wendelin.Serwe@inria.fr (W. Serwe), bjwujian@gmail.com (J. Wu), yoneda@nii.ac.jp (T. Yoneda), zheng@cse.usf.edu (H. Zheng), myers@ece.utah.edu (C. Myers).

http://dx.doi.org/10.1016/j.scico.2016.01.002
0167-6423/© 2016 Elsevier B.V. All rights reserved.

on each ECU. More importantly, this structure is susceptible to faults since if an ECU fails, it causes a malfunction in the corresponding sensor and/or actuator. With advances in semiconductor technology, it is now possible to have multiple cores on a single chip that communicate using the *Network-on-Chip* (NoC) paradigm. A NoC approach allows a flexible mapping of ECUs to processing elements, which makes it possible for ECUs to share processing power and tolerate faults by having spare units.

Wormhole routing has been utilized in many NoC designs, such as Aethereal [1], Hermes [2], and QNoC [3]. It is a switching technique that routes a packet of data in small units, known as *flits*. A packet travels through the network like a worm, and it typically consists of a *header flit* with the packet's destination, *body flits* carrying the packet's information, and a *tail flit* indicating the end of the packet. This paper presents the verification of an asynchronous NoC architecture that supports a link-fault tolerant routing algorithm [4] extended to a multiflit wormhole routing setting. A unique feature of the routing algorithm for this NoC architecture is that it uses a *deadlock avoidance* rather than a *deadlock prevention* scheme. In other words, rather than creating a routing algorithm that is proven to be deadlock-free, potential deadlocks are detected and avoided by dropping packets. To the best of our knowledge, this work is the first formal verification of a deadlock avoidance routing function. The verification takes advantage of the CADP (*Construction and Analysis of Distributed Processes*) toolbox [5] and its process-algebraic modeling language LNT (formerly *LOTOS New Technology*) [6].

Previous work on the verification of this routing algorithm [7] applies data abstractions to enable model checking of deadlock freedom and single-link fault tolerance. It, however, is flawed in that the abstract model does not include certain routing failure cases that are present in its concrete counterpart. This paper uses an example to illustrate the flaws in the abstraction and provides a refined, corrected abstraction. During this correction process, redundant fault-tolerance behaviors are found to cause livelocks in the presence of multiple faults. Through a series of diagnostic examples on the concrete model, an improvement to avoid livelocks is made on the NoC architecture and model. Deadlock and livelock freedom, single link-fault tolerance, and packet delivery are formally analyzed. Finally, this paper describes several remaining challenges to the verification of this and similar systems.

This paper is organized as follows. Section 2 describes the NoC architecture and routing algorithm. Section 3 discusses a livelock problem discovered through formal analysis. Section 4 presents an improvement of the NoC routing algorithm to avoid livelocks. Section 5 presents verification results for deadlock and livelock freedom, and properties of single-link fault tolerance and packet delivery. Section 6 surveys related work. Section 7 discusses the insights obtained from using model checking in the design of the NoC architecture and some future research directions.

2. Network-on-Chip architecture and routing algorithm

A fully functional NoC system has to be fault-tolerant and free of deadlocks. The Glass/Ni fault-tolerant routing algorithm [8], guarantees deadlock freedom by disallowing certain turns (i.e., changes in routing direction) in the network, so that communication cycles cannot occur. However, the Glass/Ni routing algorithm uses the *node-fault model*, where a fault in an incoming link is interpreted as the complete node failing. Not only does this mean losing the ability to route to an otherwise functional node, but if the node does not actually stop operating, it can potentially introduce deadlock in the network. Imai et al. proposed a modified version [9] that achieves one link-fault tolerance by introducing a mechanism to forward link fault locations to a neighboring routing node allowing for a route selection that avoids the faulty link. This fault forwarding method though can still result in a deadlock at the edges of a mesh network, so in these cases, it must revert to the node-fault model. Wu et al. described an improvement [4] that is capable of handling link faults anywhere in the network. A feature of this routing algorithm is its flexible fault-tolerance mechanism. It handles transient link failures and allows illegal turns whenever there is no danger of a cyclic communication dependency creating a deadlock. In case a potential cyclic communication dependency is detected, deadlock is avoided by dropping the packet that attempts to make an illegal turn, which effectively breaks the cycle. In other words, this routing algorithm only drops packets whenever there is a potential to form a cycle of dependencies. It is this deadlock avoidance scheme that this paper attempts to improve upon and to formally verify.

There are nine types of router nodes for this architecture as depicted in the three-by-three mesh shown in Fig. 1. Namely, there are four types of corner nodes (i.e., nodes 00, 02, 20, and 22), four types of edge nodes (i.e., nodes 01, 10, 12, 21), and one type of center node (i.e., node 11). A larger network would include duplicates of the routers shown here. This architecture implements an extended version of the routing algorithm [4] described by Wu et al. The original algorithm assumed single-flit packets and that each node could route only a single packet at a time, while this modified architecture allows each node to potentially have several multi-flit packets in flight at a time. For example, node 00 may be routing a packet from node 01 to node 10, while simultaneously routing a packet from node 10 to node 01. To accomplish this, each node xy is composed of several independent *routers* (r_D_xy) and *arbiters* (arb_D_xy), where $D \in \{\mathrm{PE}, \mathrm{E}, \mathrm{N}, \mathrm{S}, \mathrm{W}\}$ corresponds to the direction the packet is coming from in the case of routers and going to in the case of arbiters. Notice that because arbiters need storage capacity to avoid deadlocks [7, Section 4.2], it is necessary to keep them separate from the routers.[1]

[1] One might argue that it is not necessary to include the arbiters arb_PE_xy in the model, because we assume that a processing element is always ready to consume a packet, so that there is nothing to arbitrate. However, to be closer to the real circuit, we prefer to keep these processes, in particular because they do induce only a small performance penalty in verification execution time.