J Supercomput
DOI 10.1007/s11227-017-2035-5

AnyNoC: new network on a chip switching using the shared-memory and output-queue techniques for complex Internet of things systems

Jia-Yang Lin[1] · Yi-Ting Hsieh[2] · Trong Nghia Le[3] · Wen-Long Chin[3]

© Springer Science+Business Media New York 2017

Abstract Recently, the Internet of things (IoT) has attracted a lot of attention owing to its versatile applications by enabling numerous things/objects to collect and exchange data via Internet. Despite the promising role of IoT, there exists the problem of integrating many heterogeneous functions into an embedded and complex IoT system. Meanwhile, in the past decade, we have also envisioned a paradigm shift in the embedded system market toward the system on a chip (SoC) by integrating all components into a single chip. But the on-chip communications of IoT systems remain an important and challenging issue. This work proposes a new network on a chip (NoC) switching, AnyNoC, employing the shared-memory and output-queue techniques implemented using the efficient dynamic link list, particularly suitable for the IoT SoC. The proposed high-level design can achieve the optimal performance by sharing the data buffer among all ports and eliminating the head-of-line blocking problem, resulting in a virtual point-to-point characteristic without the interruption of slow devices or congestion conditions in other ports. Moreover, the proposed architecture can minimize the required memory size by virtually sharing all buffers among all ports, resulting in one queue needed for each outbound port and totally N queues are required, where N denotes the number of ports. Therefore, compared to the famous wormhole switching, the proposed NoC architecture features lower cost and higher performance, which can approach the theoretical upper bound. Moreover, for a 16×16 network, the per-

✉ Wen-Long Chin
johnsonchin@pchome.com.tw

1 Himax Corp., No. 26, Zilian Road, Xinshi Dist, Tainan City 74148, Taiwan

2 Chunghwa Telecom Corp., No. 21-3, Sec. 1, Xinyi Rd., Zhongzheng Dist., Taipei City 100, Taiwan

3 Department of Engineering Science, National Cheng Kung University, Tainan 70101, Taiwan

Published online: 06 April 2017

formance gain of the throughput of the proposed switching compared to the popular wormhole switching is about 40%.

Keywords Internet of things (IoT) · Network on a chip (NoC) · Shared-memory and output-queue switching · System on a chip (SoC)

1 Introduction

Recently, the Internet of things (IoT) has received a lot of attention owing to its versatile applications, e.g., Industry 4.0 and Smart Factory, by enabling numerous devices to collect and exchange data via Internet [1]. Data gathered through IoT can be analyzed to take autonomous decisions. The IoT is the network of physical objects including all kinds of devices, vehicles, and other items embedded with electronic components, software, sensors, and communication interfaces. Based on communication capability, intelligence can be built into those physical objects around our lives. However, future IoT system will include versatile things/objects with low, moderate, or high complexity. Hence, the large scale and autonomy features of the IoT make the problem of integrating many heterogeneous functions into an embedded system an opening issue. Other challenges of the IoT include technologies, applications, and standardization [2].

Modern system on a chip (SoC) design shows a clear trend toward integration of many homogeneous and/or heterogeneous intellectual-property (IP) cores [3]. SoC designs provide integrated solutions to challenging design problems in the telecommunications, multimedia, automotive, and consumer electronics market. The integration of numerous components into a single system gives rise to new challenges. To keep pace with the advanced semiconductor process of 20 nm technology node and beyond, a high-level, component-based methodology for large-scale SoC architectures that can reduce design time receives lots of attention in the last decade. A key component in modern SoCs is the interconnection technology enabling reliable communications and interconnect existing components in a plug-and-play fashion [4].

Traditional on-chip buses, e.g., advanced high-performance (AHB) [5], open core protocol (OCP) [6], and CoreConnect [7] buses, can cost effectively connect a few tens of components [8], while point-to-point connections can only connect less components. In contrast, the switching technique is usually preferred because it allows low channel setup time, and reduces the dependence between latency and inter-node distance [9]. To manage the complexity of designing a SoC containing tens or even hundreds of IP cores, scalable on-chip communication infrastructure is playing an increasingly dominant role in SoC designs [10]. The routing mechanism for the large-scale SoC still remains an important and challenging issue [11].

We design a new network on a chip (NoC) as the scalable and high-performance communication infrastructure for the IoT SoC featuring easy integration of miscellaneous IoT devices. Various NoC designs have been elucidated. A good survey can be found in [12]. Interesting readers can refer to it and its references therein. The proposed NoC router, named AnyNoC, achieves the optimal performance by allowing multiple outstanding packets and/or flits, resulting in a virtual point-to-point characteristic

without the interruption of slow devices or congestion conditions in other ports, and has the following new features compared to conventional switching techniques:

- Shared-memory switching architecture
  All the flits initiated by source devices are stored in the shared memory and waiting for transmission once destination devices are available. The flits destined to the same device obey the first-come-first-served (FCFS) policy. The shared-memory switching architecture can optimally utilize the flit buffer.
- Output-queue structure
  Output-queue structure can avoid the head-of-line (HOL) blocking [13], thereby enabling true parallel transmission paths between different router ports.

This work employs the shared-memory and output-queue techniques implemented using the dynamic link list./queryBoth the terms 'dynamic link list' and 'dynamic linked list' are used throughout the article. Please suggest whether they are distinct or interchangeable. Moreover, the proposed high-level design can minimize the required memory size by virtually sharing all buffers among all ports, resulting in one queue needed for each outbound port and totally N queues are required, where N denotes the number of ports. Compared to conventional switching techniques with a complexity of $\mathcal{O}(N^2)$, the proposed architecture has a complexity of $\mathcal{O}(N)$ owing to the shared-buffer architecture. AnyNoC using the proposed flow control mechanism achieves the statistical multiplexing gain compared to conventional switching techniques employing dedicated buffers. To fully utilize the whole data buffer, we propose to use the dynamic linked list to efficiently build up the data structures of all queues.

The major contributions of this work are outlined as follows.

1. We point out the problem of integrating many heterogeneous functions into a complex and embedded IoT object.
2. We propose a new architecture for the IoT SoC, which can be easily expanded to accommodate plenty of devices in a plug-and-play fashion.
3. The proposed architecture provides a virtual point-to-point characteristic by employing the shared-memory and output-queue techniques. As such, the HOL blocking is eliminated and the memory buffer can be fully utilized. The novel dynamic linked list is devised to efficiently build up the data structures of all queues.
4. The performance gain of the proposed NoC router compared to the popular wormhole switching is about 40%.

The rest of this paper is organized as follows. Section 2 introduces the proposed NoC architecture and its design principles. Section 3 summarizes the experimental results. Conclusions are drawn in Sect. 4.

2 AnyNoC: router based on shared-memory and output-queue switching techniques for the IoT SoC

A single broadcast medium, such as AHB bus, which performs in a time-division multiplexing can no longer provide the required bandwidth and latency for modern SoCs. The popular wormhole switching, insufficient for the IoT SoC, has the input

queue structure, suffering from the HOL blocking, thereby, reducing the performance a lot. By contrast, the shared-memory switching can fully utilize the flit buffer and reduce memory used to store flits. Moreover, the output-queue structure can eliminate the HOL blocking. The statistical multiplexing gain can be obtained by adopting the proposed architecture and flow control mechanism.

2.1 Architecture

We propose to utilize the dynamic linked list to efficiently build up the data structures of all queues. Figure 1 demonstrates the block diagram and labeled operating sequence of the proposed NoC router. Similar to a conventional NoC switch, there are five network interfaces (NIs), but the router core is different. There are three sub-blocks in the router core: slave manager, data manager, and queue manager.

The slave manager is used to look up the egress port of the incoming flit by the slave table; hence, the connection is virtually setup in the slave table.

The data manager contains the flit buffer, i.e., (synchronous) static random access memory (SRAM), called data_ram, used to store the flits queued in the output queues. The flit buffer is completely shared by all output queues. To save the cost, the data_ram is implemented using a single-port SRAM, instead of flip-flops. To fully utilize the bandwidth of flit buffer, the linked list is maintained using another SRAM, called link_ram, separated from the flit buffer. The flit buffer is logically partitioned into blocks of the flit size. Each block is one-to-one corresponding to the entry in link_ram.

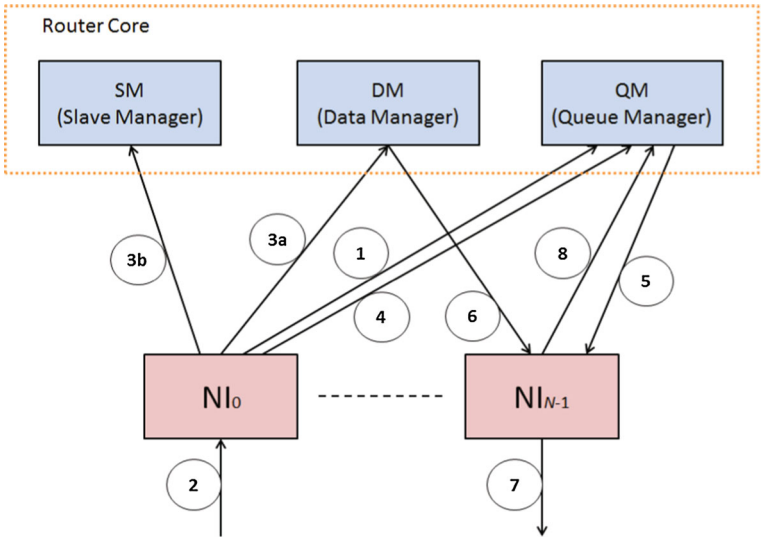

Fig. 1 Block diagram and labeled operating sequence of the proposed NoC router, $N = 5$. Four ports are used for traffics destined to the eastern, western, southern, and northern directions, respectively. Another port connects the IP core. There are eight steps for the switching operation of each flit. But those steps for adjacent flits are not necessarily in order, e.g., step 1 is performed once the floating queue is not empty. The steps processing different flits are pipelined

The queue manager maintains the dynamic linked list. The node of the linked list represents its corresponding block in the flit buffer. Three kinds of queues employ the linked list data structure: output queues, an available-block queue, and floating queues. The output queues store the flits waiting for transmission. The available-block queue indicates those unused blocks, while the floating queues are short queues used by ingress ports to store blocks used by current or future incoming flit. Each port has one floating queue, which stores those blocks that are $prefetched$ by an ingress port.

Once the incoming flit has been completely received, it is enqueued to its destination port, thereby composing an output-queued structure. If there are total P blocks in the router and, at a certain time, $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{F}$ are, respectively, the sets of total blocks in the output queues, available-block queue, and floating queues, then

$$P = |\mathcal{O} \cup \mathcal{A} \cup \mathcal{F}| = |\mathcal{O}| + |\mathcal{A}| + |\mathcal{F}| \tag{1}$$

should be guaranteed, as $\mathcal{O} \cap \mathcal{A} = \emptyset$, $\mathcal{A} \cap \mathcal{F} = \emptyset$, and $\mathcal{F} \cap \mathcal{O} = \emptyset$, where $|\cdot|$ denotes the cardinality and $\emptyset$ denotes the empty set.

In Fig. 1, the circled label 1 prefetches blocks when the floating queue is not full. Doing so is to ensure that there are available blocks to receive incoming flits in a non-blocking fashion. Prefetched blocks are obtained from the available-block queue. The circled label 2 indicates receiving (RX) of flits. The circled label 3a stores received data into the flit buffer. Concurrently, the circled label 3b looks up the slave table to determine the output port. The circled label 4 enqueues the received flit indicated by the floating queue to destined output queue. The circled label 5 notifies the destination port the status of output queues and provides the block identification (id) to be transmitted. The circled label 6 reads the flit data from the flit buffer. The circled label 7 transmits (TX) flits. Finally, once the flit processing is finished, the circled label 8 returns the occupied block to the available-block queue.

2.2 Principle of queue operations

We propose to utilize the dynamic linked list to build up the data structures of all queues. The stored number represents the block id of a flit and its corresponding buffer. The entry in link_ram stores the $next$ block id. Figure 2 illustrates the $logical$ queue and the $physical$ contents of the memory representing it, where head and tail denote the head and tail pointers of a queue, respectively. The flit buffer is fully shared by all router ports and the NoC router needs only one link_ram to process the dynamic linked list structure constituting all queues.

As an example, Figs. 3 and 4 present two basic linked list operations used to establish the linked list for the output queue: enqueue and dequeue operations, respectively. Figure 3 enqueues Q_2 into Q_1. There are two micro-operations. First, write the content of Q_2 head pointer into the address indicated by Q_1 tail pointer. Then update Q_1 tail pointer as Q_2 tail pointer. Only one memory access is required. Figure 4 dequeues first two blocks from the linked list. There are also two micro-operations. First, read the content of the dequeue tail pointer indicated by Q_2 tail. Then, it is used to update the Q_1 head pointer. Dequeue operation requires one memory access.

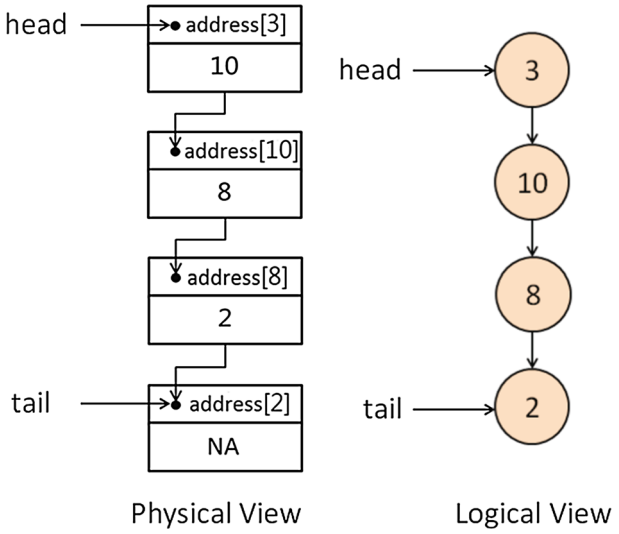

Fig. 2 *Logical* queue and the *physical* contents of the memory representing it, where head and tail denote the head and tail pointers, respectively. NA denotes not applicable

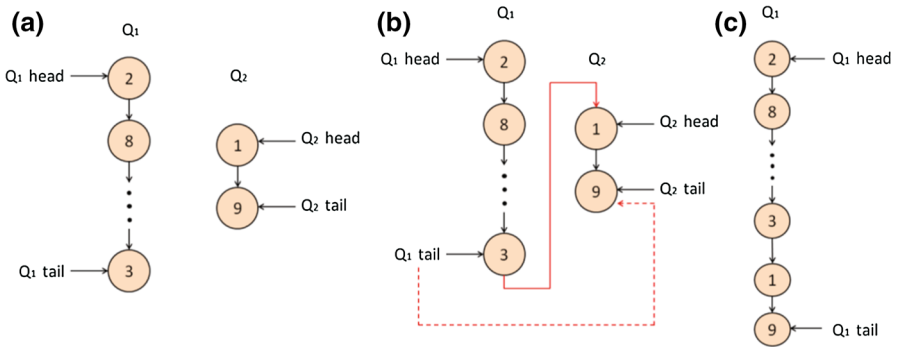

Fig. 3 Enqueue operation: **a** Q_2 will be enqueued into Q_1. **b** First, write the content of Q_2 head pointer into the address indicated by Q_1 tail pointer. Then update Q_1 tail pointer as Q_2 tail pointer. **c** Final view of logical queue

Four circled labels of the operating sequence, i.e., 1, 4, 5, and 8, relate to the maintenance of all three queues. The circled label 1 gets blocks from the available-block queue, one at a time. One read memory access is required to update the available-block queue. The circled label 4 enqueues the floating queue blocks into an output queue, which needs one write memory access for each flit. When sending a flit, the circled label 5 reads its block id, one at a time. The circled label 8 releases the occupied blocks to the available-block queue, which needs one write memory access for each flit.

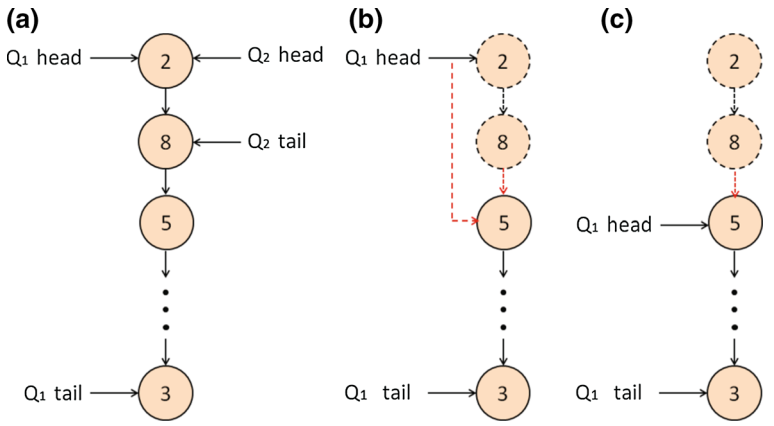

Fig. 4 Dequeue operation: **a** Q_2 will be dequeued from Q_1. **b** First, read the content of the dequeue tail pointer indicated by Q_2 tail. Then, it is used to update the Q_1 head pointer. **c** Final view of logical queue

2.3 Flow control mechanism

Well-designed flow control can eliminate flit loss and avoid congestions. To improve the performance and utilization of flit buffer while reduce implementation complexity, we propose a simple flow control mechanism that can suit well for the proposed switching architecture. The proposed flow control mechanism can let hotspot output queues in use occupy most of the flit buffer, provided that the number of available blocks is not lower than some predefined threshold. On the contrary, when remaining resources are limited, the transmission of greedy devices will be paused until the congestion condition is relived. The threshold is defined such that those output queues contributing to little or no occupance of flit buffer can still use the flit buffer without interruption by those congested ports.

Therefore, the incoming flit destined to the i-th output queue is paused once the flow control enable

$$\text{fc\_en}[i] = \text{ab\_fc\_en} \cap \text{ab\_fc\_en}[i] \tag{2}$$

becomes true, where $\cap$ denotes the intersection, the flow control enable signal for the available block $\text{ab\_fc\_en} = |\mathcal{A}| < \text{TH\_AB}$, TH_AB denotes the threshold of the available block, $\text{ab\_fc\_en}[i] = |\mathcal{O}_i| > \text{TH\_OQ}$, $\mathcal{O}_i$ denotes the set of available blocks used by the i-th output queue, and TH_OQ denotes the threshold of the output queue.

3 Experimental results

For the purpose of high-level evaluation of the switching performance, we use a self-designed traffic generator with perfectly controlled traffics destined to random and/or fixed devices. For all NoCs, simulations were conducted for a 32-bit wide system and 10-flit packets. The performance results are compared for the conventional wormhole

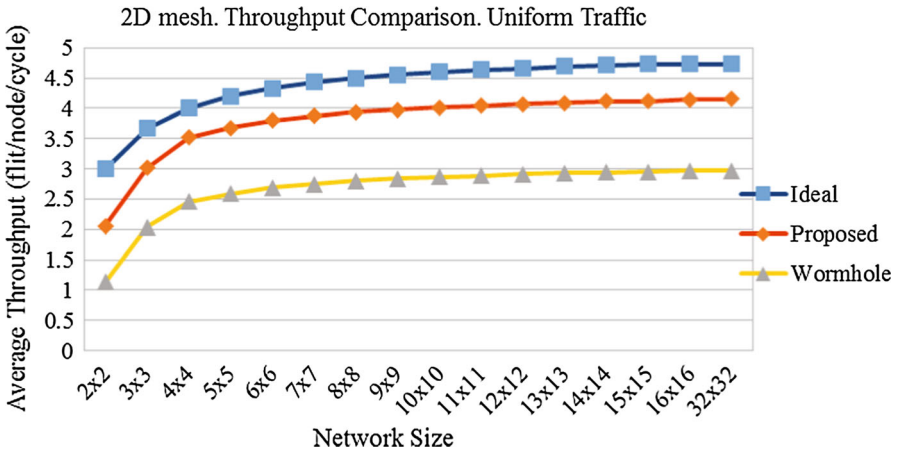

Fig. 5 Performance of the average throughput (flit/node/cycle) plotted as a function of network size. The traffic injection rate is 45%

switching and the proposed one using the same number of data (flit) buffer, i.e., 80-flit deep buffer.[1] In the proposed NoC switching, after extensive experiments, we configure TH_AB = 40 and TH_OQ = 30. Notably, there exists tradeoff for the utilization of buffers and fairness among all ports under various thresholds. The packets are forwarded using the X-Y routing scheme.

Figure 5 presents the throughput of the proposed switching (AnyNoC), conventional wormhole switching (Wormhole), and the average number of links per router (Ideal) under traffics uniformly destined to different IP cores. To achieve the cycle accurate simulation, the designs are implemented using Verilog HDL, and we focus on the popular wormhole switching and the proposed approach. The traffic injection rate is 45%. The number of links L represents the maximum throughput,[2] i.e., the number of flits per cycle of a node, the network can sustain and can be expressed as

$$\begin{aligned} L &= \frac{4 \times 3 + [4(n-2)] \times 4 + (n-2)^2 \times 5}{n^2} \\ &= 5 - \frac{4}{n} \end{aligned} \tag{3}$$

where $\times$ denotes multiplication and $n \times n = n^2$ denotes the number of routers in a network or network size. In the numerator, the first term 4×3 represents 4 router nodes in the corner with 3 links for each node; the second term $[4(n-2)] \times 4$ represents $[4(n-2)]$ router nodes on the edge, except those 4 corner nodes, with 4 links for each node, and the last term $(n-2)^2 \times 5$ represents the remaining $(n-2)^2$ interior router

[1] It will be shown later that the proposed design can outperform the popular wormhole technique under the constraint of the same buffer size. In another viewpoint, to achieve the same performance as the wormhole switching, the proposed design requires less buffer size.

[2] Ideally, each link can transmit one flit in every cycle.

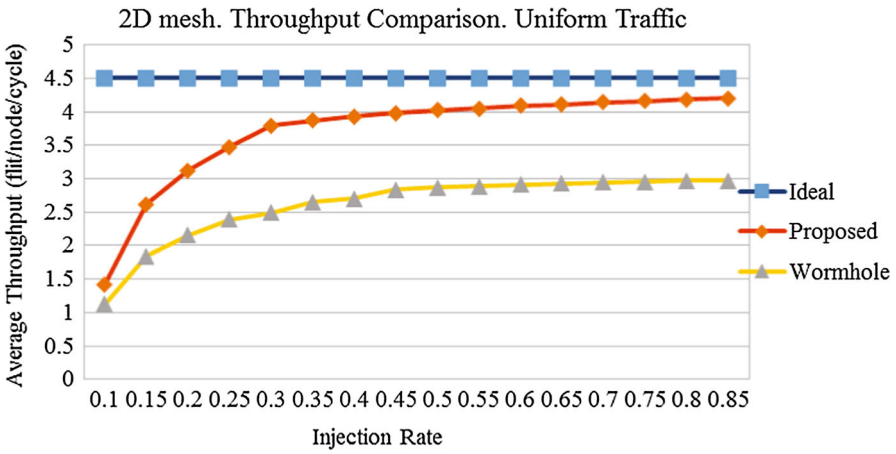

Fig. 6 Performance of the average throughput (flit/node/cycle) plotted as a function of traffic injection rate. The network size is 8 × 8

nodes with 5 links for each node. When $n \rightarrow \infty$, the number of links approaches 5, which is the upper bound of the throughput. As shown, the throughput increases consistently with the increase in the network size, because, for a larger network, the number of interior $BUSY$ router nodes increases as well. As expected, the proposed architecture outperforms the wormhole switching by eliminating the HOL blocking and fully utilizing the data buffer. For a 16 × 16 network, the performance gain of the proposed switching compared to the wormhole switching is about 40%.

Figure 6 plots the average throughput (flit/node/cycle) as a function of traffic injection rate. The network size is 8 × 8. As shown, the proposed architecture outperforms the wormhole switching for various traffic injection rates, particularly for the high traffic injection rate. When the traffic injection rate is low, the performances of compared approaches are close because the throughput is bounded by the traffic injection rate.

For both proposed and wormhole switchings, Fig. 7 presents the average latency (clock cycles) from the source device to the destination device plotted as a function of traffic injection rate for 4 × 4 and 8 × 8 networks. The latency is influenced by many factors, including switching technique, buffer size, and routing algorithm. As displayed, the latency increases when the network size becomes large. When the switch saturates, the latency will exponentially increase. For the 4 × 4 NoC, the latency saturation points for the proposed and wormhole switchings are roughly under the traffic injection rates of 0.33 and 0.4, respectively, while, for the 8 × 8 NoC, the latency saturation points for the proposed and wormhole switchings are roughly under the traffic injection rates of 0.23 and 0.29, respectively. Therefore, under the same buffer size, the proposed switching has a higher saturation point than the wormhole switching, which means that the congestion condition using the proposed technique happens later than the wormhole switching. Moreover, the latency of the proposed technique increases gradually.

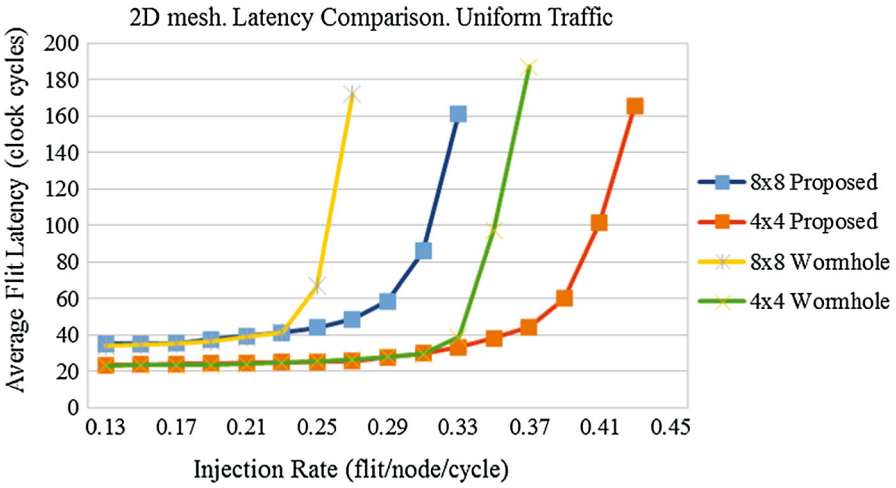

Fig. 7 Performance of the average latency (clock cycles) from the source device to the destination device plotted as a function of traffic injection rate for 4×4 and 8×8 networks

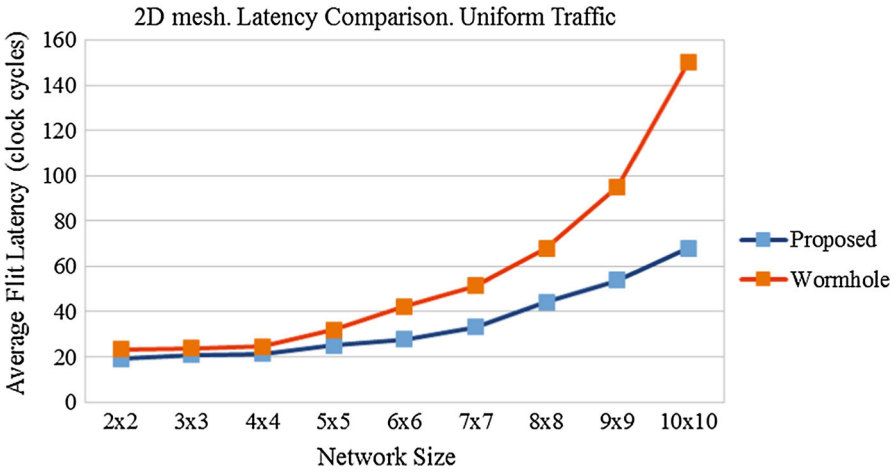

Fig. 8 Performance of the average latency (clock cycles) from the source device to the destination device plotted as a function of network size. The traffic injection rate is 25%

Figure 8 plots the average latency (clock cycles) as a function of network size. The traffic injection rate is 25%. As shown, the proposed architecture outperforms the wormhole switching for various network sizes, particularly for the large network size. When the network size is small, the performances of compared approaches are close because the latency is limited by the network size.

To fairly compare two different architectures, the performance results are obtained using the same number of data (flit) buffer for different techniques. As shown above, the performance can be enhanced by the proposed architecture. In other words, to

achieve the same performance, the occupied resources of the proposed design can be lower than that of conventional switching techniques.

4 Conclusions

This work points out the problem of integrating many heterogeneous functions into a complex and embedded IoT object. With a moderate engineering effort for the dynamic link list operation, we designed and implemented a shared-memory output-queued NoC router, AnyNoC, with a high utilization and performance gain, and verified that the proposed design can minimize the design gap of communication infrastructure in modern IoT SoC designs. The proposed architecture features easy integration of miscellaneous IoT devices. The proposed flow control mechanism can simply achieve the statistical multiplexing gain and suit well for the proposed architecture. We also demonstrated that the proposed architecture can improve the performance of the on-chip network significantly. Despite an increase in wiring complexity, the proposed AnyNoC design is promising for future complex SoC integration. The wiring complexity can be solved by increasing metal layers via modern semiconductor process technology. Therefore, compared to the popular wormhole switching, the proposed NoC architecture is promising for the IoT SoC and features lower cost and higher performance, which can even approach the theoretical upper bound.

Acknowledgements This work is supported in part by the grant MOST 105-2221-E-006-019-MY2, Taiwan.

References

1. Xu Q, Aung KMM, Zhu Y, Yong KL (2016) Building a large-scale object-based active storage platform for data analytics in the internet of things. J Supercomput 72(7):2796–2814
2. Chen S, Xu H, Liu D, Hu B, Wang H (2014) A vision of IoT: applications, challenges, and opportunities with China perspective. IEEE Internet of Things J 1(4):349–359
3. ITRS Edition Reports (2011) http://public.itrs.net/reports.html
4. Benini L, De Micheli G (2002) Network on chips: a new SoC paradigm. Computer 35(1):70–78. doi:10.1109/2.976921
5. AMBA Specification Rev. 2.0 (1999) ARM Axis Sunnyvale, CA
6. Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. Revision: B.3, Released: 7 Sept 2002. https://opencores.org/cdn/downloads/wbspec_b3.pdf. Accessed 5 Apr 2017
7. IBM Microelectronics (1999) CoreConnect bus architecture. IBM White Paper
8. Kyeong KR, Eung S, Mooney VJ (2001) A comparison of five different multiprocessor SoC bus architectures. In: Proceedings of EUROMICRO Symposium Digital System Design, pp 202–209
9. Bindal A, Mann S, Ahmed B, Raimundo L (2005) An undergraduate system-on-chip (SoC) course for computer engineering students. IEEE Trans Educ 48(2):279–289
10. Tayan O (2009) Networks-on-chip: challenges, trends and mechanisms for enhancements. In: Proceedings of ICICT'09, pp 57–62
11. Yu Z, Xiang D, Wang X (2015) Balancing virtual channel utilization for deadlock-free routing in torus networks. J. Supercomput. 71(1):3094–3115
12. Diguet JP (2014) Self-adaptive network on chips. In: Proceedings of SBCCI'14, pp 1–6
13. Nachiondo T, Flich J, Duato J (2006) Destination-based HoL blocking elimination. In: Proceedings of ICPADS'06, pp 1–10