Floodgate: Application-Driven Flow Control in Network-on-Chip for Many-Core Architectures

Yoshi Shih-Chieh Huang[†], Huan-Yu Liu[†], Yuan-Ying Chang[†], Chung-Ta King[†], and Shau-Yin Tseng[‡]

[†]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[‡]SoC Technology Center, Industrial Technology Research Institute, Hsinchu, Taiwan
[†]{yoshi, elmo, king}@cs.nthu.edu.tw, [‡]tseng@itri.org.tw

ABSTRACT

With the prevalence of multi- and many-core architecture, network-on-chip (NoC) is becoming the main paradigm for on-chip interconnection. However, the performance of NoCs can be degraded significantly if the network flow is not controlled properly. Most previous solutions have tried to detect network congestion by monitoring the hardware status of the network switches or links. Unfortunately, such strategies rely on the backpressure of the traffic flows for congestion detection and may be too slow to respond. This paper proposes a proactive strategy which predicts the global, end-to-end traffic patterns of the running application and takes preventive flow control actions to avoid congestions. The proposed system entails an application-level prediction table for accurate traffic prediction and a packet injection scheduler for congestion avoidance. The proposed scheme is evaluated by a trace-driven simulator with synthetic traffic traces as well as a real application trace of an instance in the SPLASH-2 benchmark. The results show the superior performance of the proposed scheme with negligible execution overhead.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network architecture and design

Keywords

network-on-chip, end-to-end traffic prediction, flow control, congestion control

1. INTRODUCTION

With the introduction of many-core architecture for advanced MPSoC, network-on-chip (NoC) is becoming the main paradigm for on-chip interconnection of the cores. NoCs not only offer significant bandwidth but also provide outstanding flexibility and scalability. There have already had many-core processors on the market that adopt NoC as the communication fabric. For example, Tilera's TILE64 [3] uses a 2-D mesh network to interconnect sixty-four tiles and four memory controllers.

Since the cores on the chip are connected by a network, congestion and flow control are critical for the NoC performance. If one or more routers in the network receive more packets than they can handle, the queued packets may delay or even block the flows in other routers, causing more queuing delays and packet blockages. As a result, the amount of packets that can be accepted to the network drops sharply and the packets already in the network will experience long delays [4].

To the best of our knowledge, there are few research works on congestion control in NoCs. In [10], the switches exchange their load information with neighboring switches to avoid hot spots where most packets will pass through. In [12, 13], a predictive closed-loop flow control mechanism is proposed, which predicts how many flits the router can accept in the next k time steps. However, it ignores the flits injected by neighbor routers in the prediction period. In [2, 11], a centralized, end-to-end flow control mechanism is proposed. Nevertheless, it needs a special network, called control NoC, to transfer OS control messages; it also relies on blocked messages in the local buffer for packet injection decisions. Flit-reservation flow control is also a switch-layer flow control mechanism. Similarly, it requires a special look-ahead network for booking the resources in the network [14]. In [15], the weights of virtual channels are adjusted based on the estimated number of overlapped transmissions. The prediction is based on the utilization of the ports of each router.

These works detect network congestions by monitoring the status of local hardware, such as buffer fillings, link utilization, and the number of blocked messages. Unfortunately, if congestion occurs somewhere else in the network, it will take some time to detect it, often through the backpressure of the network flow. Such an approach is too passive and slow, and cannot respond to the congestion in time. Similarly, when a core determines that the network is out of congestion according to local hardware status and decides to inject packets onto the network, other cores might still experience congestion, causing more severe congestion.

In this paper, we consider NoC-based distributed-memory many-core systems, i.e., the cores communicate with each other via explicit *send/receive* instructions rather than implicit communications through shared variables. To address the congestion control problem in such systems, we propose *Floodgate*, a proactive congestion detection and flow control mechanism. The core idea is to predict the global, end-to-end traffic in the NoC according to the data transmission behavior of the running application. With the prediction, we can control network injection to prevent congestions. Note that our prediction scheme is based on the premise that the running application exhibits certain repetitive communication patterns. Such a *self-similarity* traffic property can be found in many applications [17], e.g., when executing a loop. These patterns greatly affect the network states because applications are the sources of network traffic.

The main contributions of this paper are as follows. First, we show that congestion control in NoC can be made more effective by controlling the traffic at the source ends considering application behavior. Second, we demonstrate that it is possible to proactively control the traffic at the source ends for congestion avoidance if the end-to-end traffic can be predicted. Third, we introduce the design of a table-driven predictor/controller that not only captures and predicts the data transmission behaviors in the application at runtime, but also decides how to control the traffic at source ends. Note that, since we predict and control traffic at the source ends, the traffic is recorded before being injected to and mixed in NoC. The proposed traffic predictor is not affected even when multiple applications are running in the systems, as long as they do not occupy the same cores.

This paper is organized as follows. In Section 2, a motivating example is given to show the repetitive data transmission behavior in applications. In Section 3, related works are discussed. Next, we give a formal definition of the flow control problem in Section 4. In Section 5, we present the details of *Floodgate*, its overheads for implementation, and the various ways to implement. Evaluations are shown in Section 6. Finally, conclusions are given in Section 7.

2. MOTIVATING EXAMPLE

In this section, we show that data transmission behavior in parallel programs appears to have repetitive patterns. We use the LU matrix decomposition in the SPLASH-2 benchmark as an example. LU decomposition tries to factorize a matrix to two matrices, one is a lower triangular matrix and the other is a upper triangular matrix. The LU decomposition kernel is ported to the TILE64 platform and run on 4×4 tile, as the first diagram in Figure 1 shows. Detailed experimental setup is given in Section 6. We use 16 tiles to execute the application, and the routing algorithm is X-Y dimensional routing. This program has been modified to use massage passing only. In the following discussion, we use the notation ($source \rightarrow destination$) to describe the transmission pairs.

Figure 1 shows the transmission trace of router 4. In the second diagram, the traffic is mixed from the viewpoint of the East. The mixed traffic is somewhat messy and hard to predict. In previous works, the traffic prediction is made mainly by checking the hardware status, such as the fullness of buffers, the utilization of links, and so on. The hardware status is affected by the mixed traffic as the diagram shows. Irregular traffic makes it difficult to predict the network workload based on the hardware status only.

However, when we extract the traffic between the pairs ($5 \rightarrow 4$), ($6 \rightarrow 4$) and ($7 \rightarrow 4$), they are more regular and predictable, as the third to the fifth diagram show, with

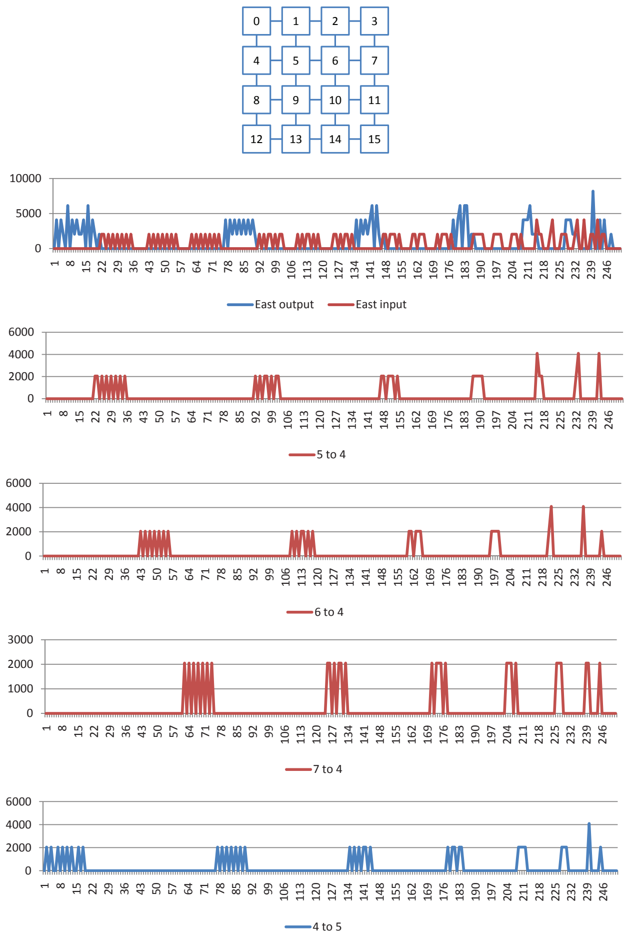


Figure 1: The network topology and the traffic of router 4 is tracked. The first diagram is all the traffic input/output from router 4. The second to the fourth diagrams show the decomposed traffic. Note that the traffic relayed by router 4 is omitted. The last one is the output traffic from router 4 to 5

the last diagram showing the output traffic ($4 \rightarrow 5$). Each transmission corresponds to an end-to-end data transmission issued by the running application. From the figure, we can see that the end-to-end data transmissions exhibit certain repetitive patterns, since the application is executing similar operations during this interval. We can also see that the communication patterns are quite similar but time-shifted. This creates an opportunity to predict other end-to-end traffic pairs by using an existing one. This part is left for further study. Finally, we can see that applications usually transfer data for a period and keep silent for another period.

By utilizing the repetitive traffic patterns based on application behavior, we can predict the end-to-end data transmissions accurately by recording the history. The traffic of a given link in the network can be predicted by summing all the predicted end-to-end data transmissions that pass through this link. As we can predict the NoC traffic in the next time interval, we can control packet injection at the sources to regulate the traffic and avoid congestion in the NoC

3. RELATED WORKS

In [10], information of the routers is exchanged with each other for deciding the routing path to avoid congestion. The control information is sent locally and cannot reflect the status of the whole network. In [12, 13], network congestions are predicted based on their proposed traffic source and router model. By using this model, each router predicts the *availability* of its buffer ahead of time, i.e., how many packet flits a router can accept currently. The traffic source cannot inject packets until the availability is greater than zero. They predict traffic from the router perspective. In [8, 19, 20, 21], a congestion control scenario is proposed that models flow control as a utility optimization problem. These works propose an iterative algorithm as the solution to the maximization problem. In [11], they use operating system (OS) and let system software control the resource usage. In [2], they proposed a NoC communication management scheme based on a centralized, end-to-end flow control mechanism by monitoring the hardware status. All the works above need a dedicated control NoC to transfer OS-control messages and a data NoC for delivering data packets. In [9], the authors add some extra hardware to support a distributed HW/SW congestion control technique. In [15], they use a dedicated core as the *master core* to control the weights of the virtual channels. Each destination node feedbacks its status to the master core periodically via the control network. This work also introduces the global and end-to-end communication concepts. However, it still relies on each port to estimate the number of overlapped links.

In contrast to the works mentioned above, our work makes predictions from the application layer rather than the link layer. Intuitively, our approach is more effective because application generates traffic and congestion-control must be based on application behavior. Note that there is no universally accepted definition of network congestion [18]. In [22], link utilization of a router is used as the indicator. In [23], the queuing delay is used. We take *link utilization* as a congestion measure in this paper. Similarly, latency is a commonly used performance metric that can be interpreted in different ways [4]. We use the definition that the time elapsed from the message header is injected into the network at the source node to the tail of the packet is received at the destination node.

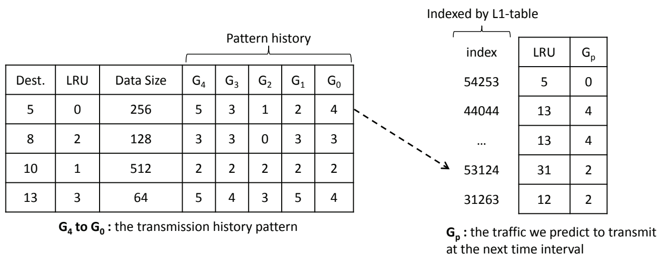
4. PROBLEM FORMULATION

The utilization of a link e_i at the t-th time interval is defined as:

$$Util_i(t) = \frac{D_i(t)}{T \cdot W} \qquad 0 \le Util_i \le 1$$

where $D_i(t)$ denotes the total data size transmitted by e_i at the t-th time interval. The period of a time interval is defined as T seconds and W is the maximum bandwidth of a communication link. Thus $T \cdot W$ denotes the maximum possible data size transmitted in one time interval. Besides, we assume that one core run one task at most.

To prevent network from being congested, we need to predict possible traffic at the t-th time interval in advance. With the prediction, we can schedule the packet injection effectively and avoid network congestion to improve the average packet latency. Assume that λ is the average packet latency, and t_{exec} is the total execution time without any flow control. Assume also that λ' is the average packet latency and t'_{exec} is the total execution time with our proposed flow control. Our goal is to maximize $\lambda - \lambda'$ and $t_{exec} - t'_{exec}$.


Pattern history
Indexed by L1-table
G_4 to G_0 : the transmission history pattern
G_p : the traffic we predict to transmit at the next time interval

Dest.	LRU	Data Size	G_4	G_3	G_2	G_1	G_0
5	0	256	5	3	1	2	4
8	2	128	3	3	0	3	3
10	1	512	2	2	2	2	2
13	3	64	5	4	3	5	4

index	LRU	G_p
54253	5	0
44044	13	4
...	13	4
53124	31	2
31263	12	2

Figure 2: An example of a L1- and L2-table. The columns G_4 to G_0 record the quantized transmitted data size of the last 5 time intervals. L2-table is indexed by the transmission history pattern G_4 to G_0. The corresponding data size level G_p is the value predicted to transmit in the next time interval

5. FLOODGATE DESIGN

In this section, we present the system design of *Floodgate*. The design consists of two parts: (1) predicting global, end-to-end network traffic by using a table-driven predictor, and (2) making traffic control decisions with an extra table, which records the delayed transmissions. The traffic prediction method in part (1) is originally proposed in [7]. However, that work only discusses how to monitor and predict network traffic, without considering how to react and control. In this paper, we focus more on using the information obtained from part (1) to perform traffic control in part (2).

5.1 Background

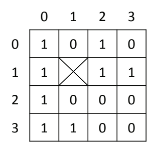
To simplify the discussion, we assume a 2D mesh topology in the NoC with a size of $N = M \times M$. Note that our approach is independent of the topology and the size of the network. Each tile consists of a processor core, a memory module and a router. We assume that the router has 5 input and 5 output ports and a 5×5 crossbar. Each crossbar contains five connections: east, north, west, south and the local processor. Each connection consists of two unidirectional communication links for sending and receiving data, respectively. Deterministic routing is assumed so that the path between a source and a destination is determined in advance. This is the most common type of routing in the current NoC.

5.2 Network Interface Design

To support the prediction and feed the data back to the centralized controller, the network interface of a core needs the following supports.

5.2.1 L1-Table: Pattern recorder

A table-driven predictor is employed to record the traffic of the past history. The history is then used to predict the data size and the destination of the outgoing traffic from each router in the next time interval. Each router maintains two hierarchical tables for tracking and predicting the data transmission. The first-level table (L1-table) as shown in Figure 2 tracks all output data transmissions. Each router



	0	1	2	3
0	1	0	1	0
1	1	╳	1	1
2	1	0	0	0
3	1	1	0	0

Figure 3: An example of the gates of core $(1,1)$

here uses only four entries to record transmission destinations. This is because a core may only communicate with a subset of all the cores [7], and the destination entries can be replaced by the Least Recently used (LRU) replacement policy for reducing the size of the table.

5.2.2 L2-Table: Pattern matcher

The history recorded above is then matched against a second-level table (L2-table). The matching entry will contain a traffic value, which can be used to predict the traffic for the next time interval. Specifically, at the beginning of the t-th time interval, the transmission history recorded in the L1-table is used to index the L2-table to get the predicted transmission data size at the t-th time interval. At the end of the t-th time interval, when an output transmission is issued by the processor core, the destination and data size are recorded in L1-table. The data size is quantized and recorded in G_0. The columns from G_0 to G_n records the quantized transmitted data size of the last $n+1$ time intervals. The two tables are updated at the end of each predefined time interval. After checking the prediction, the value of the data size counter in the L1-table is quantized and shift into G_0. One important optimization is based on our key observation as shown in the motivating example. That is, different cores tend to have repetitive behaviors but at different times. This leads to the use of only one L2-Table as the shared-pattern matcher among all cores to reduce the overhead.

5.2.3 Updating operations

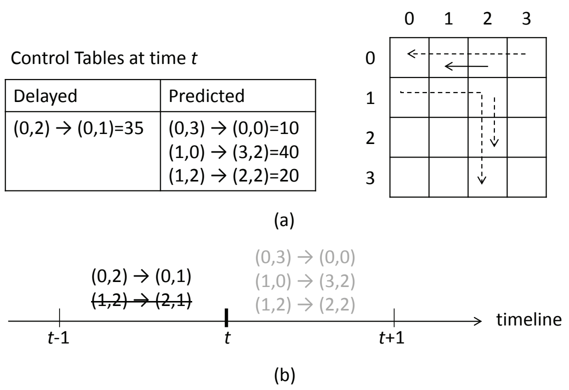
The updated transmission history in the L1-table is used to index the L2-table and retrieve the predicted data size that will be transmitted in the next time interval. If the transmission history cannot be found in the L2-table, the system will either create a new entry or replace the existing entry by LRU in the L2-table. It then uses the last value (G_0) as the predicted transmission data size. The recorded transmitted data sizes in the L1-table are used to check the accuracy of the prediction made at the last time interval. If the prediction is wrong, the value of G_p at the L2-table for the corresponding transmission history pattern will be modified to the data size level recorded in L1-table.

5.2.4 Gates ON/OFF

Each network interface maintains an array for representing whether the gate to a destination is **ON** or **OFF**. Figure 3 shows an example of the gates of core (1,1). An entry of **1** stands for **Gate ON**, and **0** stands for **Gate OFF**. If $\mathbf{Gate}_{x,y}$ is set as **0** at the t-th time interval, it means that this core cannot inject packet flits to $core_{x,y}$.

5.3 Floodgate Administrator

5.3.1 Design concepts



Delayed	Predicted
(0,2) → (0,1)=35	(0,3) → (0,0)=10 (1,0) → (3,2)=40 (1,2) → (2,2)=20

Figure 4: An example of the tables of *Floodgate* at time *t*. (a) shows the delayed and predicted transmissions, and the corresponding visualized figure. The solid line stands for delayed transmission, and the dash line stands for the predicted transmissions. (b) shows the perspective of timeline

Given the predicted end-to-end traffic for the next time interval, it is easy to calculate the total traffic running through each link in the NoC during that period. If the predicted total traffic exceeds a threshold, which is usually a percentage of the physical capacity of the link, a network congestion is likely to occur. A decision thus has to be made as to which end-to-end transmissions should be throttled to avoid congestion. A plausible solution is to apply the *Shortest Job First* (SJF) strategy, i.e., those transmissions with the shortest transmission times are allowed to proceed first. Of course, this strategy will create starvation, but it can be solved using *priority aging*. The priority of a transmission becomes higher if it is delayed longer. These issues and solutions have been discussed extensively [16]. It follows that *Floodgate* needs to maintain a table for recording the delayed transmissions and the predicted transmissions.

1. A transmission can only be allowed if *all* the links on its routing path do not have their predicted total traffic exceeding a capacity threshold.

2. Every link is examined and the high priority transmissions are selected first.

3. The priority of a transmission is higher if its transmission time is shorter or if it is delayed longer.

Next, we use an example to illustrate the rules.

5.3.2 An illustrative example

Take Figure 4 as an example. At the (t-1)-th time interval, there are two transmissions $(0,2) \to (0,1)$ and $(1,2) \to (2,1)$. The former is delayed and the latter is allowed to transmit. Next, at the t-th time interval, by the aggregated prediction data, transmissions of $(1,0) \to (3,2)$, $(1,2) \to (2,2)$, and $(0,3) \to (0,0)$ are predicted to happen between time interval t and t+1. For the former two transmissions, if the overlapped links do not exceed the link capacity, both are allowed to transmit according to *rule 1*. Otherwise, the one with a heavier workload is delayed, and the other is preferred according to *rule 2*. However, the lighter transmission still needs to satisfy *rule 1*. The two rules implement the shortest-job-first strategy and minimize the waiting time. Similarly, the transmission $(0,3) \rightarrow (0,0)$ is preferred over $(0,2) \rightarrow (0,1)$. However, according to *rule 3*, the latter has been delayed for one time interval, so it has a higher priority. This rule avoids starvation.

5.4 Implementation of Floodgate

Floodgate administrator is responsible for making the decisions and sending the control signals to all nodes. The prediction data can be aggregated to a place via the control network periodically. This architecture is similar to [15], and the details of designing the hardware is deeply discussed. In this paper, a dedicated core is used as the master core to control the gates in all the nodes. The concept of dedicating a core to do the controlling, monitoring, and debugging are also discussed in [5]. By dedicating one core to do the flow control, no extra control logics are required and thus reduce the costs. With different target platform, different implementation of one bit requires different number of transistors. For example, 6 transistors are required with SRAM implementation. However, with flip-flop implementation, which is mostly used in ASIC design, each bit may requires 60 transistors. These implementation details are left to our technical report due to the page limitation, and we discuss the architectural design in this paper. For the details and the hardware overhead estimation, please refer to the technical report [6].

5.4.1 Prediction errors

Another practical problem is that what if error prediction occurs. Certainly, since *Floodgate* forbids a source to inject data into the network once the routing path is predicted as congested. However, if the prediction is error the links may result in under-utilized or over-utilized.

In practice, there is an additional 2-bit saturating counter. If the prediction is accurate, the pattern-oriented predictor will be used in *Floodgate*. However, if *Floodgate* has a predefined number of continuous error prediction, it means that currently running tasks tend not to have repetitive behaviors as the history records, so the prediction mechanism will be switched to *last value prediction*, i.e., always guess that the next time interval has almost the same traffic volume as previous time interval, and temporarily prevents the pattern table from being modified. This is based on the observation that applications tend to transfer data for a period. *Floodgate* will change its built-in predictor according to the current situations to avoids some applications which have intermittent repetitive behaviors.

6. EXPERIMENTAL RESULTS

6.1 Simulation Setup

The PoPNet network simulator [1] is used for our simulations and the traces are generated from TILE64. The data transmission traces record the packet injection time, the address of the source router, the address of the destination router and the packet size. The detailed configuration of simulation is provided in Table 1. A link e_i is considered as congested in t-th time interval if $Util_i(t) \geq 0.8$ [22].

The original data transmission traces are altered by *Floodgate*, and this results in that some transmissions are delayed for some periods so as to avoid congestion. The experimental results presented in the following show that *Floodgate* exhibits huge performance improvement.

Table 1: Simulation Configurations.

Parameters	Setting
Network Topology	4×4 mesh
Flit size	8 bytes
Virtual Channel	3
Buffer Size	12 flits
Routing Algorithm	X-Y routing
Bandwidth	4 flits/cycle
Prediction Interval	1000 cycles
Threshold	0.8

Table 2: A summary for the latency improvement of a real application trace, in terms of cycle.

	Original	Floodgate	Reduction
Avg. latency	2410.79	771.858	3.12
Max. latency	5332	3242	1.64

6.2 Real Application Traffic Trace

The Tilera's TILE64 platform is used to run the benchmark programs and collect the data transmission traces. We use SPLASH-2 blocked LU decomposition as our benchmark program. The packet size ranges from 6 flits to 30 flits. We sample the traffic from TILE64 every 1000 cycles, and so is the prediction interval.

As shown in Table 2, the average packet latency drops from 2410.79 *cycles* to 771.858 *cycles* and the maximum packet latency drops from 5332 *cycles* to 3242 *cycles*. The significant performance improvement origins from that we predict traffic workload in the next interval and delay some packet injection to avoid congestion. As depicted in Figure 5 (a), the packet latencies without flow control range between 0 *cycles* and 5500 *cycles*. However, with *Floodgate*, the packet latencies range between 0 *cycles* and 3300 *cycles*. These packet latencies have decreased violently so that the histogram shifts to the left side.

To bear up our conviction, Figure 6 demonstrates more details about the network congestion. We set the congestion threshold as 40 flits/link/cycle. In Figure 6 (a) without flow control, the maximum workload is far apart from the threshold, and consequently causes severe network congestion. The result shows that the average injected number of flits per cycle is 35.6 with standard deviation 3.99, which is well controlled under our target, 40 flits.

6.3 Synthetic Traffic Trace

We not only evaluate *Floodgate* with the real application traffic, but also with the synthetic traffic. In [17], the authors state that injected network traffic possesses self-similar temporal properties. They use a single parameter, the Hurst exponent H, to capture temporal burstiness characteristic of NoC traffic. Based on this traffic model, we synthesize our traffic traces as shown in Table 3. These parameters are chosen based on [17]. Table 1 in [17] shows a part of benchmark values of Hurst exponent H. In this paper, we choose five values among them for evaluations (*art, jpeg, mpeg2, mgrid, 8b_encode*). Similarly, we sample the trace every 1000 cycles and so is the prediction interval. The average packet latency and the maximum latency both drop down significantly.

To conclude, relative large H values indicate highly self-similar traffic and thus predictable, so *Floodgate* performs better. Note that the average packet size also increases with

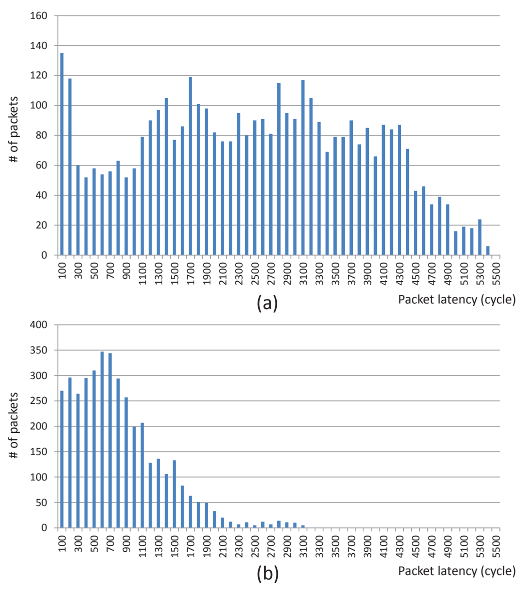


Figure 5: Histograms of the packet latencies without (a) and with (b) the proposed flow control in terms of cycles. In (b), the latencies slow down drastically

Table 3: A summary of the latency improvement for the synthetic traffic traces. Latencies are in terms of cycles. It shows that *Floodgate* leads to the huge reduction in the average latency and the maximum latency.

Hurst value	0.576	0.661	0.768	0.855	0.978
Original avg latency	3553	3597	3649	3666	3615
Improved avg latency	482.5	467.8	387.7	413	417.6
Reduction	7.36	7.69	9.41	8.88	8.66
Original max latency	7623	7623	7710	7658	7714
Improved max latency	1591	1532	1016	1054	1037
Reduction	4.79	4.98	7.59	7.27	7.44
Original sim time	8580	8510	8550	8480	8450
Improved sim time	8280	8260	7690	7781	7731

H because of the burstiness, so the reduction does not arise linearly with H.

7. CONCLUSION

In this paper, we propose an application-driven flow control *Floodgate* for packet-switched networks-on-chip. By tracking and predicting the end-to-end transmission behavior of the running applications, we can limit the traffic injection when the network is heavily loaded. By delaying some transmissions efficiently, the average packet latency can be decreased significantly so that the performance can be significantly improved. In our experiments, we adopt real application traffic traces as well as synthetic traffic traces. The experimental result shows that our proposed flow control decreases the average packet latency and the maximum latency effectively.

8. REFERENCES

[1] N. Agarwal, T. Krishna, L. Peh, and N. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2009.

[2] P. Avasare, J.-Y. Nollet, D. Verkest, and H. Corporaal. Centralized end-to-end flow control in a best-effort network-on-chip. In *Proc. 5th ACM internatinoal conference on Embedded software*, 2005.

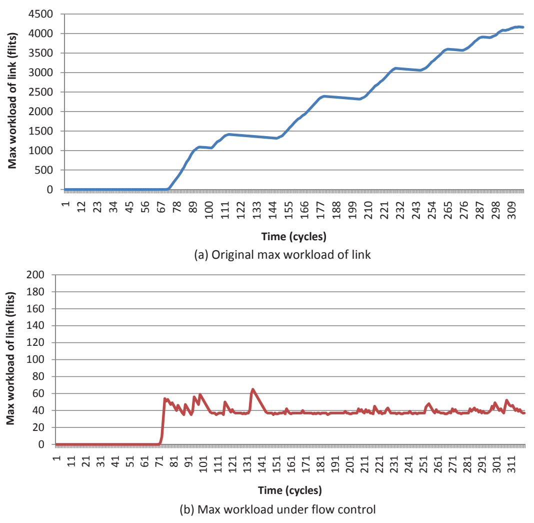


Figure 6: The maximum workload of links in the network without (a) and with (b) the proposed flow control in terms of cycles

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Proc. Digest of Technical Papers. IEEE International Solid-State Circuits Conference ISSCC 2008*, pages 88–598, Feb. 3–7, 2008.

[4] J. Duato, S. Yalmanchili, and L. Ni. Interconnection networks. pages 428–431, 2002.

[5] G. He, A. Zhai, and P. Yew. Ex-mon: An architectural framework for dynamic program monitoring on multicore processors. In *The Twelfth Workshop on Interaction between Compilers and Computer Architectures, Interact-12*, 2008.

[6] Y. S.-C. Huang, C.-K. Chou, C.-T. King, and S.-Y. Tseng. Area overhead estimation for table lookup implementation in chip design. Technical report, 2010.

[7] Y. S.-C. Huang, C.-K. Chou, C.-T. King, and S.-Y. Tseng. Ntpt: On the end-to-end traffic prediction in the on-chip networks. In *Proc. 47th ACM IEEE Design Automation Conference*, 2010.

[8] F. Jafari, M. S. Talebi, M. H. Yaghmaee, A. Khonsari, and M. Ould-Khaoua. Throughput-fairness tradeoff in best effort flow control for on-chip architectures. In *Proc. 2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009.

[9] T. Marescaux, A. Rångevall, V. Nollet, A. Bartic, and H. Corporaal. Distributed congestion control for packet switched networks on chip. In *ParCo*, 2005.

[10] E. Nillson, M. Millberg, J. Öberg, and A. Jantsch. Load distribution with the proximity congestion awareness in a network on chip. In *Proc. Design, Automation, and Test in Europe*, page 11126, 2003.

[11] V. Nollet, T. Marescaux, and D. Verkest. *Operating-system controlled network on chip.* 2004.

[12] U. Ogras and R. Marculescu. Analysis and optimization of prediction-based flow control in networks-on-chip. *ACM Transactions on Design Automation of Electronic Systems*, 2008.

[13] U. Y. Ogras and R. Marculescu. Prediction-based flow control for network-on-chip traffic. In *Proc. 43rd ACM IEEE Design Automation Conference*, pages 839–844, 2006.

[14] L. Peh and W. Dally. Flit-reservation flow control. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 73–84. IEEE, 2002.

[15] A. Sharifi, H. Zhao, and M. Kandemir. Feedback control for providing qos in noc based multicores. In *Proc. Design, Automation, and Test in Europe*, 2010.

[16] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts.* Wiley Publishing, 8th edition, 2008.

[17] V. Soteriou, H. Wang, and L.-S. Peh. A statistical traffic model for on-chip interconnection networks. In *Proc. 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 2006.

[18] K. Srinivasan. Congestion control in computer networks. 1991.

[19] M. S. Talebi, F. Jafari, and A. Khonsari. A novel flow control scheme for best effort traffic in noc based on source rate utility maximization. In *MASCOTs*, 2007.

[20] M. S. Talebi, F. Jafari, A. Khonsari, and M. H. Yaghmae. A novel congestion control scheme for elastic flows in network-on-chip based on sum-rate optimization. In *ICCSA'07: Proceedings of the 2007 international conference on Computational science and its applications*, pages 398–409, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] M. S. Talebi, F. Jafari, A. Khonsari, and M. H. Yaghmaeem. Best effort flow control in network-on-chip. In *CSICC*, 2008.

[22] J. van den Brand, C. Ciordas, K. Goossens, and T. Basten. Congestion-controlled best-effort communication for networks-on-chip. In *Proc. Design, Automation, and Test in Europe*, 2007.

[23] J. Yuho, Y. Ki Hwan, and K. Eun Jung. Adaptive data compression for high-performance low-power on-chip networks. In *Proc. 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.