Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 115 (2017) 626–634

Procedia Computer Science

www.elsevier.com/locate/procedia

7th International Conference on Advances in Computing & Communications, ICACC 2017, 22-24 August 2017, Cochin, India

Implementation and Analysis of Adaptive Packet Throttling in Mesh NoCs

Aswathy N. S.[a], Reshma Raj R. S.[a], John Jose[b,*], Josna V. R.[a]

[a]*Government Engineering College Bartonhill, Trivandrum, Kerala, India*
[b]*Indian Institute of Technology Guwahati, Assam, India*

Abstract

Network-on-Chip (NoC) has emerged as the preferred communication framework for multi-core systems. Packet throttling is a cost effective technique which simply delays packet injection into the network. However, deciding on when to throttle and how much to throttle are key design challenges of any throttling technique. Existing techniques use local throttling decisions by a central controller. This paper overcome the issues related with the existing works by partitioning the network into number of sub-networks, each with a zonal controller. The experimental results with real traffic workloads show substantial reduction in average packet latency when compared with state of the art packet throttling techniques.

Peer-review under responsibility of the scientific committee of the 7th International Conference on Advances in Computing & Communications.

Keywords: Network congestion; source throttling; congestion control

1. Introduction

Design and scalability issues associated with increasing core counts on Chip Multi-Processors (CMPs) is a top research focus area in computer architecture over the last decade. Communication among cores in these CMPs housing processors, caches and memory controllers is an important task that requires focused attention for better performance and throughput. Thus designing an efficient and scalable interconnect is critical for future energy efficient CMP designs.

Interconnects like bus and crossbars are no longer scalable with these ever growing trend in CMPs. Hence, researchers have moved towards Network-on-Chip (NoC), a scalable, packet switched and distributed interconnect framework that offer much lower latency and higher bandwidth. Most modern CMPs are arranged in 2D mesh topology due to its simple layout and short wires.

A 9-core tiled CMP implemented in 2D 3×3 mesh topology is depicted in Fig. 1. Each processing core encloses a superscalar processor, a private L1 cache and a slice of shared L2 cache as shown in the zoomed view. Each of

* Corresponding author. Tel.: +919048665842.
E-mail address: johnjose@iitg.ernet.in

1877-0509
Peer-review under responsibility of the scientific committee of the 7th International Conference on Advances in Computing & Communications
10.1016/j.procs.2017.09.149

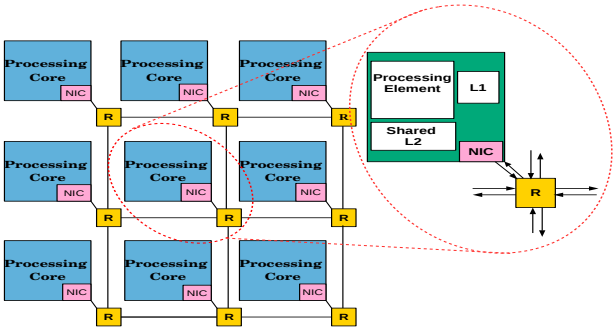

Fig. 1. Core-router interaction in a 2D Mesh Topology

this processing core is connected to a router. Inter-core communication is needed in the event of an L1 cache miss because the L2 look up happen at a core different from the source core due to the SNUCA [1] based L2 cache block mapping. The mode of communication in such systems is packet based, where the packet contains control information like source address, destination address, L2 bank address etc. Generally, a source core creates a packet when an L1 miss occurs and it is injected into the local router. Input buffers and handshaking signals between routers facilitate flow control for packet movement between source and destination routers. Wormhole switching [2] is used in most of the NoCs, where each of these packets are divided into smaller flow control units called flits. Division of packets to flits helps in reduction of input buffer size. Packets and flits have been used interchangeably throughout the paper, thus should not be confused with. Based on the employed routing algorithm, the packets traverse through the network to the destination core. Similarly reply packets also traverse through the NoC back to the source core.

Along with typical networking related challenges, NoC designers also encounter hardware constraints which include constraints in router area, power consumption and implementation complexities. With these constraints and resource crunch in modern CMPs, it is neither feasible nor practical to propose new NoC designs with sufficiently large storage buffers and wider inter-router links. As more and more packets compete for shared resources like routers and links, the overall system throughput degrades drastically. This network congestion, if not dealt properly can eventually bring the entire system down. Congestion in interconnects is now a popular problem with few works done in the recent past.

Source throttling [2]-[3] is an efficient congestion-control approach for improving system performance. It is orthogonal to hybrid designs and scalable to arbitrary workloads. Here, cores injecting large traffic and crowding the network are throttled or temporarily prevented from packet injection. Convincing source throttling proposals are available in literature.

Modern CMPs host heterogeneous applications across various cores with unpredictable network traffic patterns. This paper implements an adaptive heterogeneous optimal application aware and network load aware source throttling technique to counter network congestion. Our contribution in this paper can be summarized as follows:

- A novel cost effective throttling of network intensive cores so as to maintain maximum network utilization and throughput.
- A zonal controller is implemented (to be explained later) to coordinate throttling rate and point of throttling.

2. Related works

Source throttling in multi-processor and wormhole networks are studied even before NoC became a popular alternative to traditional bus and crossbar interconnects [2] [4]. Global knowledge-based and self tuned source throttling technique in multiprocessor networks [4] gracefully adapts to the dynamic congestion pattern. This technique upgrades system performance during heavy loads without downgrading it during light loads. A mechanism to avoid network saturation in wormhole networks by local traffic estimation using free virtual output channel count is also studied [2]. Here based on a decided threshold, next packet injection will be allowed or forbidden.

By the beginning of this decade NoC has established itself to be the numero uno scalable interconnect for ever growing CMPs. Unpredictable massive traffics from cores running heterogeneous applications lead to network congestion thus requiring the network intensive applications to be throttled or controlled. Manifestation of network congestion in bufferless NoCs is very different than in a traditional network [5]. Application-level awareness in NoCs are studied and throttling decisions are taken that boosts overall system performance to a great extent. Fairness via source throttling (FST) [6], proposes to measure the unfairness in shared memory system. FST is also capable of enforcing system software level fairness objectives including fairness-performance tradeoffs. ACT (Adaptive Cluster Throttling) [7], a source throttling mechanism explore the possibility of making application clusters based on traffic traits and then throttling these clusters alternatively. Nychis et. al. [8] discuss key differences between bufferless NoCs with traditional networks. They identify few key issues of scalability and congestion in NoCs and then propose a low complexity and high performance source throttling technique with application-level awareness for reducing network congestion.

Heterogeneous Adaptive Throttling (HAT) [9] which is both application aware and network load aware allows network-sensitive applications to make fast progress by throttling network-intensive applications, with maximum network utilization. This is the first throttling technique combining both application-aware and network-load-aware metrics to tackle congestion in NoC. Yan et. al. propose Cbufferless, a distributed source throttling mechanism for bufferless NoCs to detect congestion and control them [10] [11]. SCEPTER bufferless NoC [3] uses source throttling among other mechanisms to traverse non-minimal paths without latency penalty. They programmed a distributed self-learning throttling technique where each core independently learns and tunes its throttle rate with respect to global starvation indicators.

3. Motivation

Source throttling is always considered a congestion control technique in traditional packet switched networks. It is introduced in NoC for tackling with heavy traffics from data intensive cores. The effort is to mitigate congestion by identifying network intensive cores and then selectively throttling packet injections from those cores to reduce network load. Once the system reaches a stable state, throttling is disabled.

Since heterogeneous applications inject diverse traffic into the network, a source throttling technique must be application aware for deciding on whom to throttle. Blindly throttling applications only based on their traffic pattern might lead to under or over utilization of network resources, hampering performance in both the cases. Thus a source throttling technique must also be network aware for knowing on how much to throttle. And most importantly, the hardware that implements throttling should be simple. Available techniques in literature are either application oblivious [2] [4], network load unaware [5] [8] [10] [11] or sub-optimal [6] [7] [9] [3].

In this work we are exploring the possibility of improving the performance of one of the sub-optimal implementation in HAT [9] as it is considered best in the available literature. HAT works on local throttling decisions taken by the core itself. In HAT each core is classified by a central controller either as network intensive or network non-intensive core based on the number of packets it inject into the network for a given time period. Cores which inject packets greater than a threshold are classified as network intensive and others fall under the group of network non-intensive. All the network intensive cores are throttled in the subsequent time period. The problem with this method is that it may lead to either over-throttling or under-throttling. Over-throttling happens when every core injects packets which is higher than a threshold value set by the central controller leading to throttling of all the cores. Under-throttling occurs when most of the cores inject very less packets while few inject packets just higher than the threshold. Even though there is not much congestion in the network, the cores which generate misses above threshold are throttled. Both over-throttling and under-throttling is because each core is unaware about what is the injection pattern in other cores. We observed 7 over-throttling cases and 8 under-throttling cases on an average upon implementing HAT using five SPEC CPU 2006 benchmark mixes (refer Table 2 for workloads) on an 8×8 2D mesh NoC.

Another problem with HAT is the single central controller. After receiving packet count updates from each core, the central controller finds out the rate of throttling. But for large networks, having a single central controller is a big bottleneck. The single central controller can cause high round trip delay. Let t be the transmission time for the request to the central controller and d be the delay at the central controller. The core need to wait $2t + d$ time to receive the response (round trip time). Since there is a single central controller, d value is high. As the controller is situated at the center, t also can be reasonably high. Because of the slow response from the central controller, the system stabilization

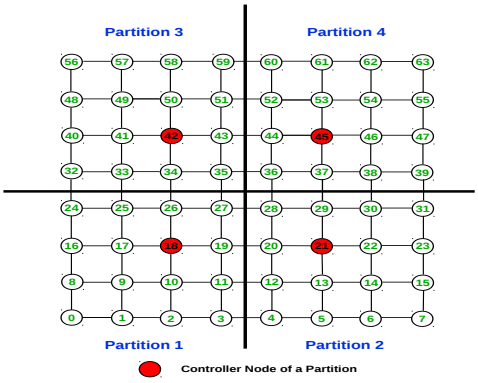

Fig. 2. Network with 4 zonal controllers

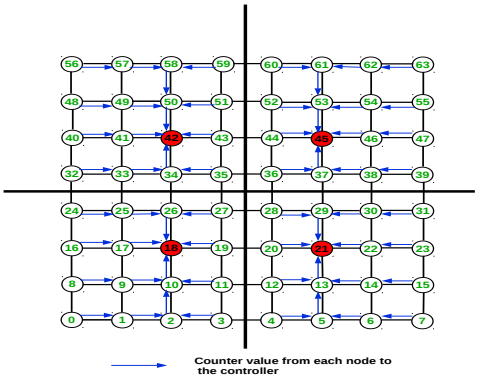

Fig. 3. Counter statistics to zonal controllers

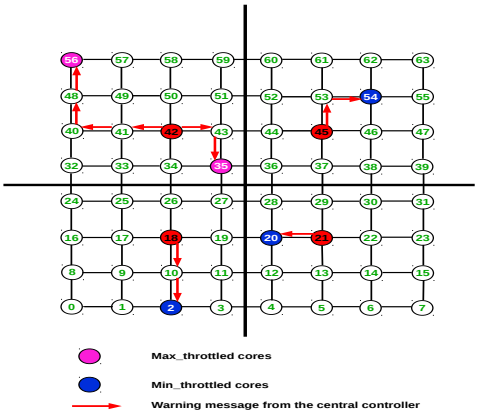

Fig. 4. Warning message from zonal controllers

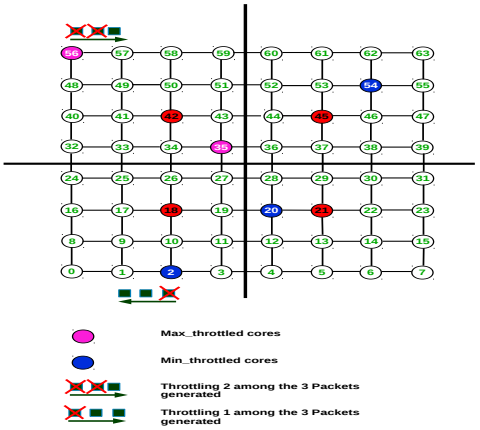

Fig. 5. Differential throttling

time increases. The experimental implementation of HAT shows that in an 8×8 mesh, the round trip delay of control packets that carry crucial throttling parameters from a core to the central controller can be around 40 - 45 cycles even at low network traffic. Moreover we find that the central controller can become a hotspot at regular intervals due to flooding of control packets from various other cores.

We are therefore motivated to propose an improved application and network load aware, adaptive source throttling technique with optimal implementation and minimum hardware overhead. Evaluation and comparison studies of the proposed approach with the existing proposals are found in the expected manner with improved system performance.

4. Proposed Method

In this approach, a 2D mesh with 8×8 organization is considered. The whole network is logically partitioned into four 4×4 sub-networks as shown in Fig. 2. Instead of using a single central controller like in HAT [9], we use four zonal controllers, one for each of the four partitions. The zonal controllers (shown in dark colours) are chosen in such a way that they should be at least two hops away from the edge nodes in all four directions. This is to ensure that the zonal controller is approximately in the center of the respective partition, so that the controller can respond to various control packets from respective cores within a reasonable time gap. We use a 5-bit *miss-counter* per core to record the cache misses generated by that core.

The whole time period is sequentially divided into a series of three phases: (1) Measurement Phase (M), (2) Processing Phase (P) and (3) Throttling Phase (T). During the measurement phase, *miss-counter* is incremented for each

of the miss generated by the respective core. At the beginning of processing phase, the miss statistics from each of the cores in the partition is send to the zonal controller as shown in Fig. 3. The zonal controller receives information from each of the core in its partition. For example, all cores in partition 1 send control message containing the miss statistics (value of *miss-counter*) during the processing phase to core 18. Core 18 will process these information received and determine the throttling parameters.

A threshold is set by the zonal controller and warning messages are sent back to the respective cores which has a *miss-counter* value greater than the threshold as shown in Fig. 4. For example in partition 3 (top left partition), the zonal controller 42 identifies 35 and 56 as the cores whose cache *miss-counter* value after the measurement phase is greater than the threshold. So warning messages are send to 35 and 56 during the processing phase to initiate throttling. Unlike in HAT, here the zonal controller determines which core to be throttled instead of taking local throttling decisions. The cores which receives the warning message learn that they have to be throttled in the next throttling phase. The zonal controller classifies cores in its partition to *max_throttled cores*, *min_throttled cores* and *nil_throttled cores*. The throttling rates are different for *max* and *min_throttled cores*; we call this differential throttling. No warning messages are generated to *nil_throttled cores*. During the throttling phase, packets generated from the *max_throttled* and *min_throttled cores* will be throttled at a pre-determined rate as shown in Fig. 5. For a *max_throttled core*, two packets will be throttled out of every three packets generated. Likewise, for a *min_throttled core*, one packet will be throttled out of every three packets generated by the core. The *miss-counter* is reset and then updated for each measurement phase (M_i) with the number of misses generated by the core during the time window. This ensures that the same core is not throttled every time.

In the proposed design, we use a time window of 128 cycles for the measurement phase, i.e. for every 128 cycles the *miss-counter* value is sent to the zonal controller. Since the time window is very less more precise will be the measurement. Moreover, the *miss-counter* size can be varied at design time based on the performance objectives. Processing phase uses 32 cycles, i.e. within this 32 cycles, the *miss-counter* values from all the cores are sent to respective zonal controllers and cores receive warning messages from zonal controllers if they are either *max_throttled* or *min_throttled* cores. We use the threshold as 10 for *min_throttled* cores and 15 for *max_throttled* cores. For the next 128 cycles (throttling phase), the *max* and *min_throttled* cores receiving the warning messages are throttled as per the throttling rate mentioned before.

Fig. 6 illustrates how the proposed system behaves in different phases of execution. Let M_1, M_2, M_3, ... be different measurement phases, P_1, P_2, P_3, ... be the corresponding processing phases and T_1, T_2, T_3, ... be the corresponding throttling phases in the entire time frame of the application's execution. Now consider an i with phases M_i, P_i and T_i. During M_i, the *miss-counter* value for each of the core is incremented for every cache miss occurred in that core. These statistics are sent to the respective zonal controllers at the beginning of P_i. The zonal controllers will then send warning messages to the *max* and *min_throttled* cores during P_i. The *max* and *min_throttled* cores are then throttled during T_i. After the completion of the first measurement phase M_1, the next phase of measurement M_2 starts its execution in parallel with the processing phase P_1. Similarly, a third measurement phase M_3 is initiated at the beginning of processing phase P_2. This series continues throughout the execution of applications in a pipelined manner.

Throttling is not blocking packets, but temporarily delaying packet injections into the network. Throttled packets try to get injected into the network in subsequent cycles. When a packet is throttled the core will try to inject the throttled packet after a delay of 2 cycles. If a new packet is generated in the core during that same cycle it will be queued in the core just after the throttled packets. Preference will be given to already throttled packets than newly generated packets for injection into the router. This makes sure that none of the throttled packets will be delayed for a longer time duration.

5. Experimental Analysis

5.1. Simulation Setup

We use BookSim 2.0 [12], a cycle accurate NoC simulator for modelling 8×8 CMP with 2D mesh topology. BookSim supports various kinds of routing algorithms, traffic patterns and network topologies. It can generate NoC traffic from real traffic traces in addition to the synthetic traffic patterns. We use the network traces generated by a 64

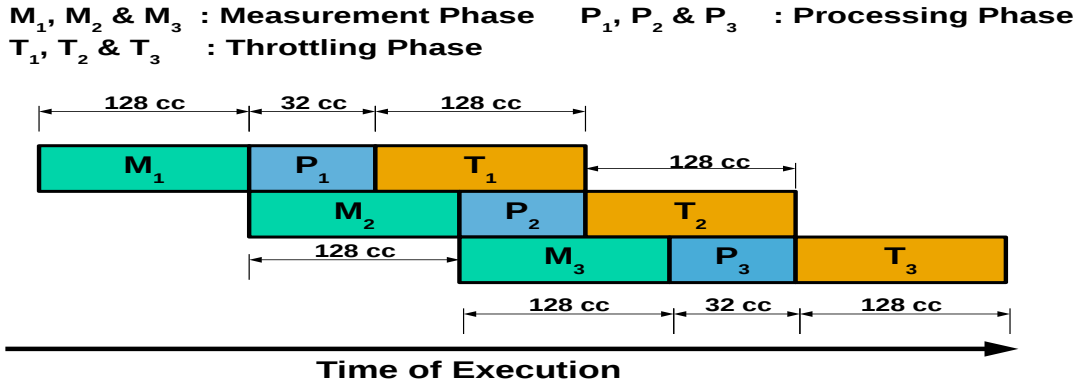

Fig. 6. Various phases in the application execution

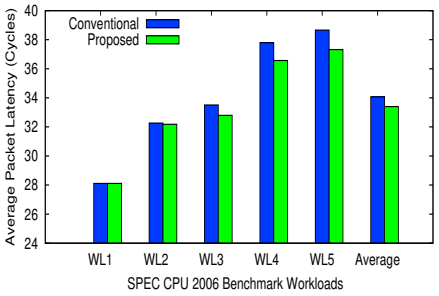

Fig. 7. Overall packet latency

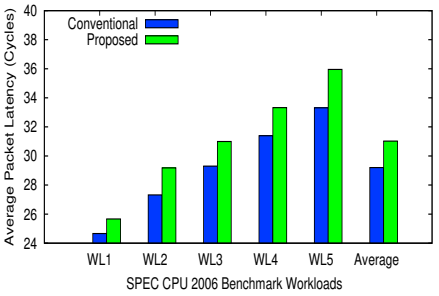

Fig. 8. Throttled packet latency

core CMP (modelled via gem5 architectural simulator [13]) upon running 64 instances of various SPEC CPU 2006 benchmark applications. We run one SPEC CPU 2006 benchmark application per core.

Based on the Misses Per Kilo Instructions (MPKI), each SPEC CPU 2006 application is grouped into Low MPKI (less than 5), Medium MPKI (between 5 and 25) and High MPKI (greater than 25). Details are given in Table 1. Low MPKI group comprises of *calculix*, *gobmk*, *gromacs*, and *h264ref*, Medium MPKI consists of *bwaves*, *bzip2*, *gamess*, and *gcc* and High MPKI includes *hmmer*, *lbm*, *mcf*, and *leslie3d* benchmark applications. We construct 5 workload mixes each with 64 application instances based on the proportion of network injection intensity of these applications as given in Table 2. To understand the distribution of benchmarks in workloads, consider workload 3 (*WL3*). Out of the 64 cores in the CMP configuration in gem5, 16 cores run *bwaves* benchmark, 16 cores run *bzip2* benchmark, 16 cores run *gamess* benchmark and the remaining 16 cores run *gcc* benchmark. Similarly applications that were used to create other workloads can also be trivially understood.

The network trace generated by the above multicore workload is given to BookSim for modelling the NoC events and statistics are collected. Each NoC router port is associated with 8 VCs with 3 flit buffers/VC. We use the Dimension Order Routing (DOR) algorithm. All cache miss requests are single flit packets and cache miss replies are 4-flit packets.

5.2. Impact of Packet Throttling

If a core is identified as to be throttled for a single throttling phase, then it is called one instance of throttling. Similarly if a core is identified as to be throttled for 2 consecutive throttling phases and another core is to be throttled for 3 consecutive throttling phases then altogether it is considered as 5 instances of throttling.

Table 3 shows the number of instances of throttling we have encountered upon using this approach on different workload mixes. We can deduce from the table that higher MPKI workloads (*WL4* and *WL5*) have large number of

Table 1. Classification of applications based on cache MPKIs

Percentage miss rate	Benchmarks (t)
Low MPKI (less than 5)	calculix, gobmk,gromacs,h264ref
Medium MPKI (between 5 and 25)	bwaves, bzip2, gamess, gcc
High MPKI (greater than 25)	hmmer.nph3,lbm, mcf,leslie3d

Table 2. Workload Constitution

Workload#	SPEC 2006 Benchmarks (t)
WL1	calculix(16) gobmk(16) gromacs(16) h264ref(16)
WL2	calculix(16) gobmk(16) gamess(16) gcc(16)
WL3	bwaves(16) bzip2(16) gamess(16) gcc(16)
WL4	bwaves(16) bzip2(16) hmmer.nph3(16) lbm(16)
WL5	hmmer.nph3(16) lbm(16) mcf(16) leslie3d(16)

Table 3. Number of throttled instances

Workload#	Min throttled cores	Max throttled cores	Total throttling instances
WL1	22	0	22
WL2	89	24	113
WL3	423	72	495
WL4	700	298	998
WL5	782	489	1271

throttled instances while low MPKI workloads generate relatively smaller number of throttled instances and also the number of *max_throttled instances* is very less compared to *min_throttled instances* for low MPKI workloads.

5.3. Effect on Overall Packet Latency

Fig. 7 shows the overall packet latency obtained from both conventional (HAT [9]) and proposed method. We can see from the figure that using the proposed method, overall latency of the system is reduced considerably across all workloads. The relative latency reduction are more predominant with *WL4* (5.89%) and *WL5* (10.45%). This also shows that the control overhead induced by throttling is not affecting the overall packet latency of the network.

5.4. Effect on Throttled Packet Latency

Fig. 8 plots the packet latency of the throttled packets. Since by packet throttling we are delaying the packet injection, the overall packet latency of throttled packets should be high. This is confirmed by our observation of an increase in average throttled packet latency on all workloads (8). Because of this throttling from congestion-causing cores it helps the packets injected by *nil_throttled cores* to reach the destination with minimal latency. Thus the average packet latency of the entire network can be reduced (7) in spite of increase in throttled packet latency (8).

5.5. Sensitivity Analysis on M, P, T and Threshold

For all the results discussed so far we considered time frames for measurement, processing and throttling phases as 128, 32 and 128 clock cycles respectively. In this section we have conducted a sensitivity analysis, with an objective to identify optimal time frames for all the phases. We have varied the time frames for M, P and T with preset and dynamic thresholds and recorded the percentage reductions in average packet latency.

Table 4 presents the effect on average packet latency with varying time frames for different phases of execution (M, P and T). For the first six cases (*Case 1 - 6*), the applied threshold is static and set as 10 for *min_throttled cores* and 15 for *max_throttled cores*.

Here we can see that Case 4 (M = 128, P = 32 and T = 128) yields the highest reduction of 6.12% in average packet latency than conventional HAT [9] approach. Now keeping these values of M, P and T fixed, we are

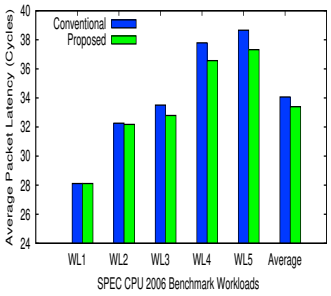

Fig. 9. Overall packet latency - Case 1

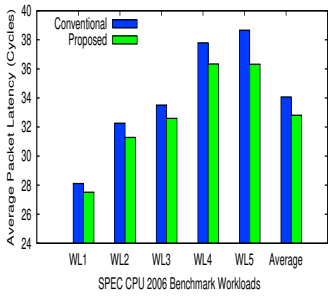

Fig. 10. Overall packet latency - Case 2

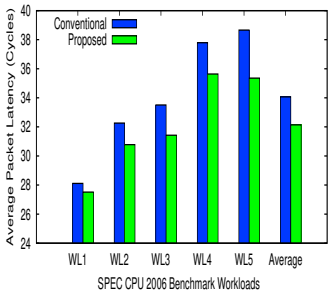

Fig. 11. Overall packet latency - Case 3

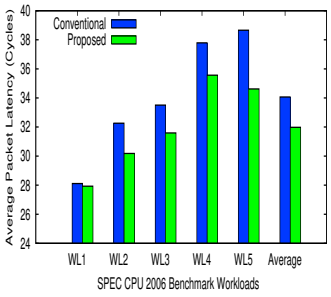

Fig. 12. Overall packet latency - Case 4

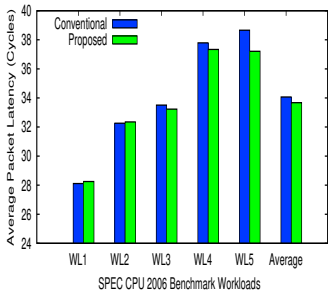

Fig. 13. Overall packet latency - Case 5

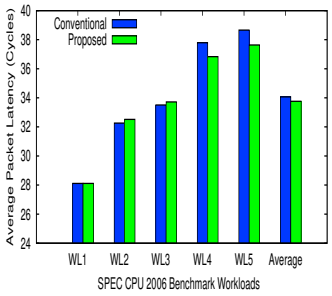

Fig. 14. Overall packet latency - Case 6

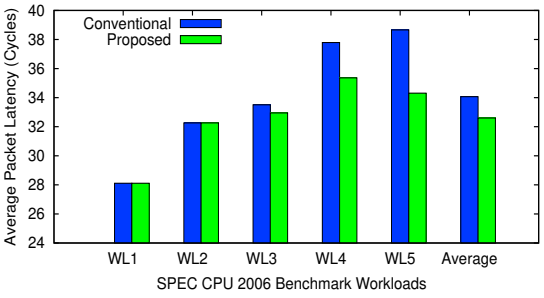

Fig. 15. Overall packet latency - Case 7

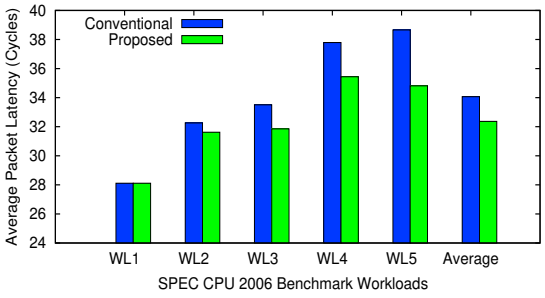

Fig. 16. Overall packet latency - Case 8

Table 4. Sensitivity analysis on M, P, T and Threshold

Case #	M	P	T	Avg. Packet Latency Reduction
Case 1	256	100	128	1.97%
Case 2	256	50	128	3.69%
Case 3	128	50	128	5.65%
Case 4	128	32	128	6.12%
Case 5	256	100	256	1.08%
Case 6	256	50	256	0.89%
Case 7	128	32	128	4.31%
Case 8	128	32	128	4.99%

dynamically changing the value of threshold in Cases 7 and 8. In both cases, the threshold is set as the average number of injected packets in the partition in a given time frame (M_i). But the calculation in finding the average is different.

In Case 7, for a given time frame the threshold is

$$\frac{\text{Total number of packets injected in a partition}}{\text{Cores with minimum 3 packets injected in the partition}}$$

Whereas in Case 8, the threshold is calculated as

$$\frac{\text{Total number of packets injected in a partition}}{\text{Cores with minimum 1 packet injected in the partition}}$$

The reduction in average packet latency for Cases 7 and 8 with dynamic threshold and preset M, P and T values are 4.31% and 4.99% respectively. These obtained reductions are at par with static threshold.

Thus, we can conclude that the ideal time frames for M, P and T with both static as well as dynamic threshold should be 128, 32 and 128 clock cycles respectively. Depending on flexibility in cost / area overhead of *miss-counter* size and phase management circuits, other combinations of M, P, T and threshold listed in Table 4 can also be adopted.

6. Conclusion

Congestion in NoC is a challenging issue to be solved with cost effective techniques. Packet throttling is one kind of such technique, which suppress packet injection into the network from the core causing congestion. We proposed a cost effective packet throttling technique which properly manages the point of throttling and the rate of throttling. Multiple zonal controllers in this technique help to overcome over-throttling and under-throttling issues of the existing throttling techniques. Unthrottled packets get more benefit by throttling of heavy injection cores. Results showed that the number of throttling instances increases with the increase in number of misses. Also, the overall packet latency of the system is decreased by throttling the congestion causing cores.

References

[1] C. Kim, D. Burger, and S. W. Keckler (2002), "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *10th International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, 211-222.

[2] E. Baydal, P. Lopez, and J. Duato (2001), "A congestion control mechanism for wormhole networks," in *Ninth Euromicro Workshop on Parallel and Distributed Processing*, 19-26.

[3] B. K. Daya, L.-S. Peh, and A. P. Chandrakasan (2016), "Quest for high-performance bufferless nocs with single-cycle express paths and self-learning throttling," in *in 53rd Annual Design Automation Conference*, 36.

[4] M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee (2001), "Self-tuned congestion control for multiprocessor networks," in *The Seventh International Symposium on High-Performance Computer Architecture, (HPCA)*, 107-118.

[5] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu (2010), "Next generation on-chip networks: What kind of congestion control do we need?" in *The 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 12.

[6] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt (2010), "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ACM SIGPLAN Notices*, **45**(3):335-346.

[7] R. Ausavarungnirun, K. K.-W. Chang, C. Fallin, and O. Mutlu (2011), "Adaptive cluster throttling: improving high-load performance in bufferless on-chip networks," *Computer Architecture Lab (CALCM) Carnegie Mellon University, SAFARI Technical Report*.

[8] G. P. Nychis, C. Fallin, T. Moscibroda, O. Mutlu, and S. Seshan (2012), "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," *ACM SIGCOMM computer communication review*, **42**(4):407-418.

[9] K. K.-W. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu (2012), "HAT: Heterogeneous adaptive throttling for on-chip networks," in *IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 9-18.

[10] J. Yan, X. Lin, and G. Lai (2013), "Cbufferless: a novel congestion control for bufferless networks on-chip," in *2nd international conference on Advances in Computer Science and Engineering*, 153-156.

[11] J. Yan, G. Lai, and X. Lin (2014), "A novel distributed congestion control for bufferless network-on-chip," *The Journal of Supercomputing*, **68**(2):849-866.

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

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and others (2011), "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, **39**(2):1-7.