Adaptive Packet Throttling Technique for Congestion Management in Mesh NoCs

Aswathy N.S[1], Reshma Raj R.S[1], Abhijit Das[2], John Jose[2], and Josna V.R[1]

[1] Goverment Engineering College Bartonhill, Trivandrum, Kerala, India
nsaswathy1993@gmail.com, reshmaraj26@gmail.com, josna.chandu@gmail.com
[2] Indian Institute of Technology Guwahati, Assam, India
abhijit.das@iitg.ernet.in, johnjose@iitg.ernet.in

Abstract. Network on Chip is an emerging communication framework for multi-core systems. Due to increasing number of cores and complex workloads, congestion management techniques in NoC are gaining more research focus. Packet throttling is one of a cost effective technique for congestion management. It delays the packet injection into the network, thereby regulating traffic in network and hence provide ease of packets generated by other critical applications. Finding point of throttling and rate of throttling are two major design issues that can impact the performance and stability of any throttling algorithm in a multi-core framework. Existing state of the art throttling techniques use local throttling decision coordinated by a single central controller. We overcome the issues related with this by partitioning the network into number of subnetworks, each with a zonal controller. Our experiment results in 8×8 2D mesh with real traffic workloads consisting of SPEC 2006 CPU benchmarks shows an average packet latency reduction of 6.2% than the state of the art packet throttling techniques.

Keywords: network congestion, packet throttling

1 Introduction

Design and scalability issues associated with increasing core counts on Chip Multi-Processors (CMPs) is a prominent research domain 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 deeper exploration for better performance and throughput. Thus designing a 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 than their traditional bus based counter parts. 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 Figure 1. Each processing core encloses a superscalar processor, a private L1

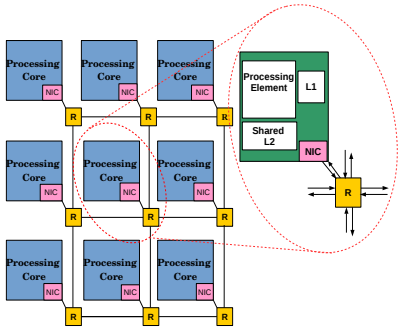

Fig. 1. Core-router interaction in a 2D Mesh Topology

cache and slice of shared L2 cache distributed all over the CMP as shown in the zoomed view. Each of these processing core is connected to a switching device called 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 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 [1] is used in NoCs, where each of these packets are divided into smaller flow control units called flits. Based on the employed routing algorithm, the required packet 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. 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. Source throttling [1]-[3],[4]-[6],[8]-[12] is an efficient congestion-control approach for improving system performance. Cores injecting large traffic and crowding the network are throttled or temporarily prevented from packet injection.

Modern CMPs run heterogeneous applications that generates unpredictable network traffic patterns across various cores. Unpredictable massive traffics from cores running heteregenous applications lead to network congestion thus requiring the network intensive applications to be throttled or controlled. We have contributed to this paper in the following ways:

– We dynamically analyze and keep track of all packet injections across the network from all processing cores without hampering the usual network flow.

- We implement a zonal controller logic (to be explained later) each in different network zones that cordinate throttling of network intensive cores so as to maintain maximum network utilization.
- We classify all the cores into either network intensive or network non-intensive based on their packet injection behaviour at regular intervals.
- Finally, we perform evaluation and comparison studies with existing techniques to understand the performance improvement obtained.

2 Related Work

Source throttling in multi-processor and wormhole networks are studied even before NoC became a popular alternative to traditional bus and crossbar interconnects[1][2]. Global-knowledge-based and self tuned source throttling technique in multiprocessor networks [2] gracefully adapts to the dynamic congestion pattern. This technique upgrades system performance during heavy loads. A mechanism to avoid network saturation in wormhole networks by local traffic estimation using free virtual output channel count is also studied [1].

Manifestation of network congestion in bufferless NoCs is very different than their buffered couterpart. Application-level awareness in NoCs are studied and throttling decisions are taken that boosts overall system performance to a great extent [3]. Fairness via source throttling (FST) [4], proposes to measure the unfairness in shared memory system. Then based on a threshold, traffic from cores that cause unfairness in the system are throttled down. FST is also capable of enforcing system software level fairness objectives including fairness-performance tradeoffs.

ACT (Adaptive Cluster Throttling) [5], explore the possibility of making application clusters based on traffic traits and then throttling these clusters alternatively. Nychis et al [6] find the key issues of scalability and congestion in NoCs and then proposes a low complexity and high performance source throttling technique with application-level awareness for reducing network congestion.

Heterogeneous Adaptive Throttling (HAT) [7] which is both application aware and network load aware allows network-sensitive applications to make fast progress by throttling network-intensive applications. This is the first throttling technique combining both application-aware and network-load-aware metrics to tackle congestion in NoC. Yan et al. analyzes deflection ratio of routing messages to understand the level of network congestion caused by deflection routing [8][9]. They propose Cbufferless, a distributed source throttling mechanism for bufferless NoCs to detect and control congestion brings extra hardware and bandwidth overhead. SCEPTER bufferless NoC [10] uses source throttling to traverse non-minimal paths without latency penalty. They programmed a distributed self-learning throttling technique for controlling new packet injection into the network, where each core independently learns and tunes its throttle rate with respect to global starvation indicators.

3 Motivation

Source throttling is introduced in NoC for tackling with heavy traffics from data intensive applications. The effort is to mitigate congestion by identifying network intensive cores and then selectively throttling packet injections from those cores to reduce congestion in the system. As the congestion goes down, the system performance goes up. Once the system reaches a stable state, throttling is disabled.

Since heterogenous 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. Thus a source throttling technique must also be network aware for knowing the throttling rate. Moreover the hardware that implements throttling should be simple. Available techniques in literature are either application oblivious [1][2], network load unaware [3][6][8][9] or sub-optimal [4][5][7][10].

In this paper we identify the limitations of HAT [7], the best application aware source throttling technique proposed so far and suggest few modifications so as to improve its performance. HAT uses local throttling decisions taken by the respective core. In HAT each application is classified by a central controller either network intensive or network non-intensive applications based on the number of packets it is inject into the network at regular 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 applications 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 injects packets just higher than the threshold. Even though there is no much congestion in the network, the cores which generate misses above threshold are throttled. Both over throttling and under throttling happen because each core is unaware about what is the injection pattern in other cores. We identify around 7 number of over throttling cases and 8 number of under throttling cases on an average upon implementing HAT using the five SPEC 2006 CPU benchmark mix (Refer Table 2 for workloads)

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 as it is not a scalable proposal. 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 processing 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 situated at the center of the mesh, both d and t also can be high. Because of the slow response from the central controller, the system stabilization time also increases. Our experimental implemetation of HAT shows

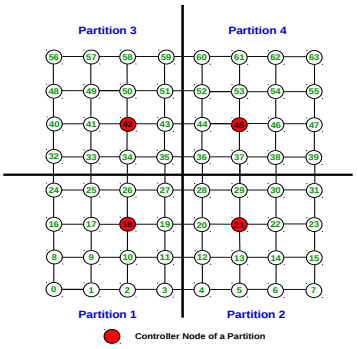

Fig. 2. Network with 4 zonal controllers

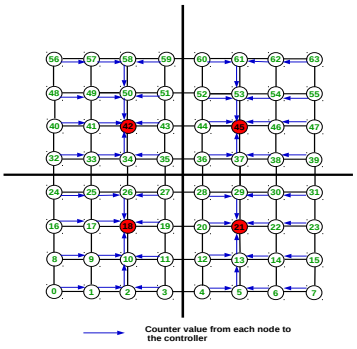

Fig. 3. Sending counter values from all nodes to the zonal controllers

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. We also find that the central controller can become a hotspot at regular intervals due to flooding of control packets from various other cores.

Exploring further on the above mentioned limitations of HAT, we propose an improved application and network load aware, adaptive source throttling technique with a distributed zonal controller logic that implements differential throttling. Evaluation and comparison studies of our approach with the existing proposals are found in our favour with improved system performance.

4 Proposed Method

In our approach, a 2D mesh with an 8×8 organization is considered. The whole network is logically partitioned into four 4×4 subnetworks. Instead of using a single central controller like in HAT [7], we use four zonal controllers, one for each of the four partitions as shown in Figure 2. The four zonal controllers (shown in dark colours) eliminate the single central controller bottleneck. The zonal controllers are selected in such a way that it should have at least two-hop neighbour in each of the four directions. This is to ensure that the zonal controller is approximately in the center of the respective partition, so that, the controller can legitimately control all the cores within that partition. We use a 5-bit counter per core to record the cache misses generated by the core.

The whole time period is sequentially divided into a series of three phases: (a) measurement phase-M, (b) processing phase-P and (c) throttling phase-T. During the measurement phase, the counter is incremented for each of the miss generated by the respective core. At the beginning of the processing phase, the miss statistics from each of the cores in the partition is send to the zonal controller as shown in Figure 3. The zonal controller receives information from each of the core in its partition. For example, all nodes in partition 1 send control

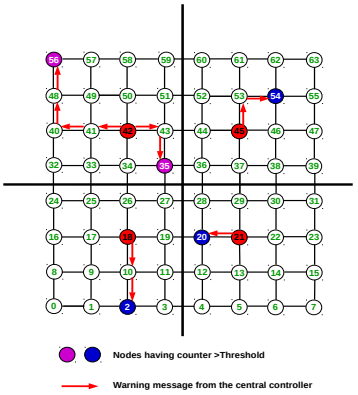

Fig. 4. Zonal controller sending warning messages

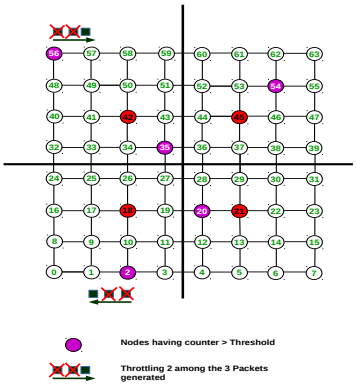

Fig. 5. Throttle cores whose counter greater than threshold

packets at the end of measurement phase to node 18. Node 18 will process these information received and determines the throttling parameters.

A threshold is set by the zonal controller and warning messages are send back to the respective cores which hold a counter value greater than the threshold as shown in Figure 4. For example in partition 3 (top left partition) the zonal controller 42 identifies 35 and 56 as the nodes whose cache miss count value during 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 the local core. Hence this approach avoids the problems associated with local throttling decision. The cores which receives the warning message learns that they have to be throttled in the next throttling phase. During the throttling phase, packets generated from the cores having counter value greater than threshold will be throttled at a pre-determined rate as shown in Figure 5. For example if throttling rate is 2/3, two packets will be throttled out of the three packets generated. Likewise, if throttling rate is 1/3, one packet will be throttled out of the three packets generated by the core.

The counter is updated for each measurement phase based on the number of misses generated by the core during the time window. This ensures that the same core is not throttled every time. The throttling of the core depends on the number of misses generated by the core during previous measurement phase. Here we use a time window of 128 cycles for the measurement phase, ie, for every 128 cycles the counter is updated.

Since the time window is very less more precise will be the measurement. Moreover, from the design perspective the size of counter can be reduced. For the processing phase we use 32 cycles, ie, with in this 32 cycles the counter

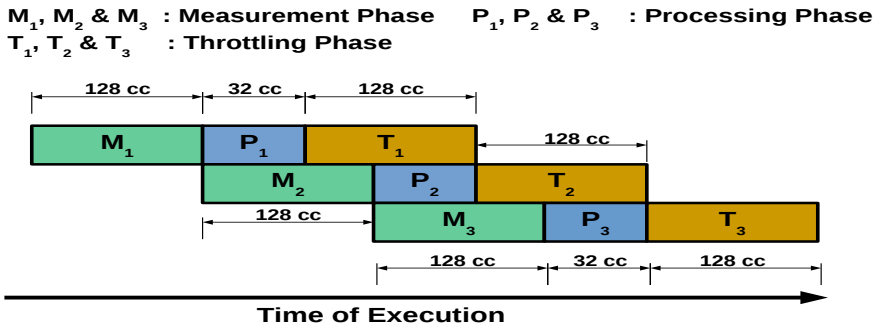

Fig. 6. Various phases in the application execution

statistics is send to the respective zonal controllers from the cores and the zonal controllers will send the warning message to the cores having counter statistics greater than the threshold of 15. After that for a 128 cycle, the cores which receive the warning message are throttled as per throttling rate mentioned.

Throttling is not blocking packets, it is temporarily delaying packets injected into the network. The throttled packets tries to inject into the network during subsequent cycles. Here we provide 2 cycle delay for each of the throttled packets ie, after the packet is throttled the core will try to inject the throttled packet after 2 cycles. If a new packet is generated in the core during the 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 waiting for injection into the router. This makes sure that none of the throttled packets will be delayed for a longer time duration.

Figure 6 illustrates how the proposed system behaves in the different phases of execution. Let $M_1, M_2, M_3, ...$ be the different measurement phases, $P_1, P_2, P_3, ...$ be the different processing phases and $T_1, T_2, T_3, ...$ be the different throttling phases of the entire time frame in the application's execution. Consider M_i, P_i and T_i. During M_i the counter value for each of the core is incremented for every cache miss request from that core. These statistics is send to the respective zonal controllers at the beginning of P_i. The zonal controllers will send the warning message to the cores with number of packets greater than the threshold during P_i. The cores which receive warning messages are throttled during T_i. After the completion of the first measurement phase M_1, the next phase of measurement M_2 starts the 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 program in a pipelined manner.

5 Experimental Analysis

5.1 Simulation Setup

We use Booksim2.0 [11], the cycle accurate NoC simulator for modelling 8×8 CMP with 2D 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 core CMP (modelled via GEM5 architectural simulator) upon running 64 instances of different SPEC 2006 CPU benchmark applications.

In GEM5 [12], we run one instance of a SPEC 2006 CPU benchmark application on each of the core. Based on the misses per kilo instructions (MPKI) each SPEC 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. We construct 5 workload mixes 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), in which we use 64 application instances for the simulation. Out of these 64 cores, 16 cores run *bwaves* benchmark, 16 cores run *bzip2* benchmark, 16 cores run *gamess* benchmark and the remaining 16 cores run *gcc* benchmark. Similarly other workloads can also be described.

The network trace generated by the above multicore workload is given to Booksim for modelling the NoC events and statistics are collected. Each of the NoC router port is associated with 8 VCs. We use the dimension order routing algorithm. All cache miss requests are single flit packets and cache miss replies are 4-flit packets.

Table 1. Classification of applications based on MPKI

Percentage miss rate	Benchmarks
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

5.2 Results & Discussions

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 3 consecutive throttling phases and another core is to be throttled for 2 consecutive throttling phases then altogether it is considered as 5 instances of throttling.

Table 2. Workload Constitution

Workload#	SPEC 2006 Benchmarks			
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)

Here, different workload mixes results in different number of throttling instances. From the result analysis, we have identified that a higher MPKI workload leads into a larger number of throttled instances while a lower MPKI workload results in a smaller number of throttled instances. For low MPKI workload(WL1) we have identified 22 throttling instances and for workload WL2 113 instances are identified. The medium MPKI workload WL3 results 495 throttling instances and for the workload WL4 998 throttling instances are identified. The largest number of instances are identified for higher MPKI wokload WL5 which is around 1271.

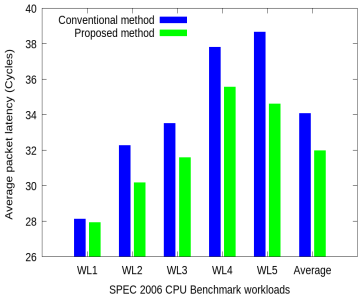

Fig. 7. Overall packet latency

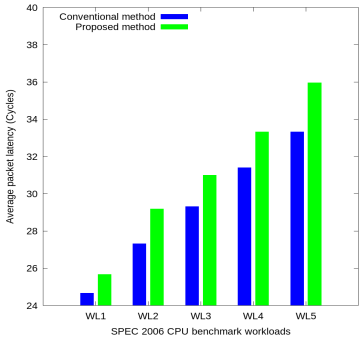

Fig. 8. Throttled packet latency

Figure 7 shows the overall packet latency obtained from both conventional method and proposed technique. We can see from the figure that using the proposed method the overall latency of the system is reduced considerably. The control overhead induced by throttling is not affecting the overall packet latency of the network. Figure 8 plots the packet latency of the throttled packets. By packet throttling we are delaying the packet injection. Hence the overall packet latency of throttled packets will be high. Delaying the packets from the congestion causing cores helps the unthrottling cores to inject packets into a least congested network and hence can reach the destination with minimal latency. Thus the average packet latency of the entire network can be reduced.

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 our 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. Baydal *et al.*, "A congestion control mechanism for wormhole networks," in *Ninth Euromicro Workshop on Parallel and Distributed Processing.* IEEE, pp. 19–26, 2001.
2. Thottethodi *et al.*, "Self-tuned congestion control for multiprocessor networks," in *The Seventh International Symposium on High-Performance Computer Architecture, HPCA.* IEEE, pp. 107–118, 2001.
3. Nychis *et al.*, "Next generation on-chip networks: What kind of congestion control do we need?" in *The 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* ACM, pp. 12, 2010.
4. Ebrahimi *et al.*, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *ACM SIGPLAN Notices*, vol. 45, no. 3. ACM, pp. 335–346, 2010.
5. Ausavarungnirun *et al.*, "Adaptive cluster throttling: improving high-load performance in bufferless on-chip networks," *Computer Architecture Lab (CALCM) Carnegie Mellon University, SAFARI Technical Report TR-2011-006, 2011.*
6. Nychis *et al.*, "On-chip networks from a networking perspective: Congestion and scalability in many-core interconnects," *ACM SIGCOMM computer communication review*, vol. 42, no. 4, pp. 407–418, 2012.
7. Chang *et al.*, "HAT: Heterogeneous adaptive throttling for on-chip networks," in *IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).* IEEE, pp. 9–18, 2012.
8. J. Yan *et al.*, "Cbufferless: a novel congestion control for bufferless networks on-chip," in *2nd international conference on Advances in Computer Science and Engineering*, pp. 153–156, 2013.
9. Yan *et al.*, "A novel distributed congestion control for bufferless network-on-chip," *The Journal of Supercomputing*, vol. 68, no. 2, pp. 849–866, 2014.
10. Daya *et al.*, "Quest for high-performance bufferless nocs with single-cycle express paths and self-learning throttling," in *in 53rd Annual Design Automation Conference.* ACM, pp. 36, 2016.
11. Jiang *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on.* IEEE, pp. 86–96, 2013.
12. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.