Adaptive Packet Throttling Technique for Congestion Management in Mesh NoCs

N. S. Aswathy[1(✉)], R. S. Reshma Raj[1], Abhijit Das[2], John Jose[2(✉)], and V. R. Josna[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, Guwahati, Assam, India
{abhijit.das,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 packet movement 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. 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 and Related Work

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.

Network-on-Chip (NoC), is a scalable, packet switched and distributed interconnect framework that offer much lower latency and higher bandwidth than their traditional bus based counter parts. In a tiled CMP each processing core encloses superscalar processor, a private L1 cache and slice of shared L2 cache distributed all over the CMP. 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. They use packet based communication where the packet contains control information like source address, destination address, L2 bank address etc. 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. These packets traverse through the network to the destination core by following the routing algorithm implemented in the router.

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 is an efficient congestion-control approach for improving system performance. Cores injecting large traffic and crowding the network are throttled temporarily from packet injection.

Source throttling in 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. Fairness via source throttling (FST) [3], 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. ACT (Adaptive Cluster Throttling) [4], explore the possibility of making application clusters based on traffic traits and then throttling these clusters alternatively. Nychis et al. [5] propose a low complexity and high performance source throttling technique with application-level awareness for reducing network congestion. Heterogeneous Adaptive Throttling (HAT) [6] which is the first throttling technique combining both application aware and network load aware allows network-sensitive applications to make fast progress by throttling network-intensive applications.

2 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 improves and 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. 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 [2], network load unaware [5] or sub-optimal [3,4,6].

In this paper we identify the limitations of HAT [6] 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 1 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 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.

3 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

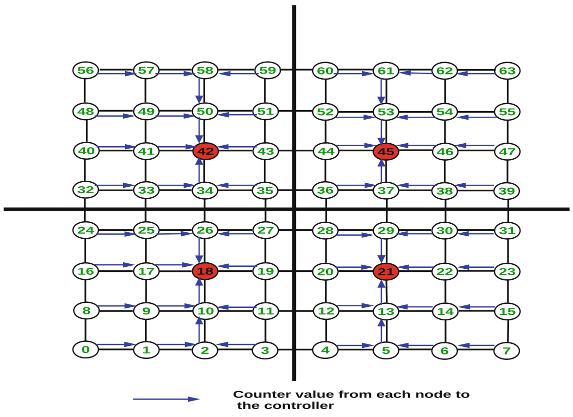

Fig. 1. Sending counter values from all nodes to the zonal controllers

a single central controller like in HAT [6], we use four zonal controllers, one for each of the four partitions as shown in Fig. 1. 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.

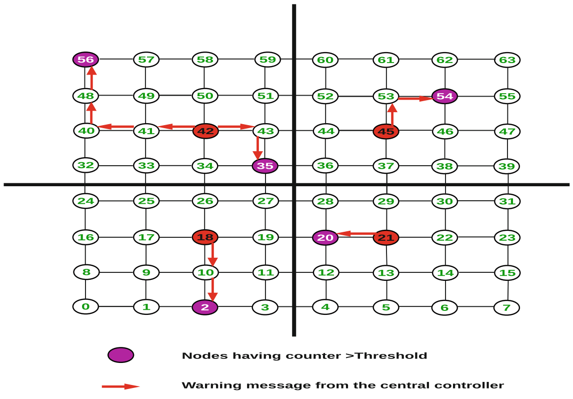
Nodes having counter >Threshold
Warning message from the central controller

Fig. 2. Zonal controller sending warning messages

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 Fig. 1. The zonal controller receives information from each of the core in its partition. For example, all nodes in partition 1 send control 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 Fig. 2. 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. During the throttling phase, packets generated from the cores having counter value greater than threshold will be throttled at a pre-determined rate. 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.

Here we use a time window of 128 cycles for the measurement phase, i.e., for every 128 cycles the counter is updated. For the processing phase we use 32 cycles, i.e., with in this 32 cycles the counter 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 i.e., 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.

4 Experimental Analysis

4.1 Simulation Setup

We use Booksim2.0 [7], the cycle accurate NoC simulator for modelling 8×8 CMP with 2D topology. Booksim supports 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 [8], 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). In our experiment we consider *calculix, gobmk, gromacs, h264ref* under Low MPKI, *bwaves, bzip2, gamess, gcc* under Medium MPKI and *hmmer.nph3, lbm, mcf, leslie3d* under High MPKI. We construct 5 workload mixes based on the proportion of network injection intensity of these applications as given in Table 1. To understand the distribution of benchmarks in workloads, consider workload 3 (WL3). Out of 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.

4.2 Results and 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 m consecutive throttling phases and another core is to be throttled for n consecutive throttling phases then altogether it is considered as $(m+n)$ instances of throttling.

Table 1. 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

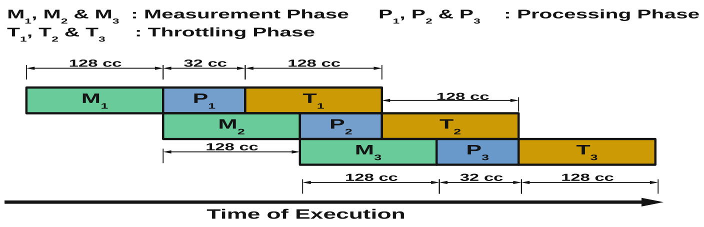
M1, M2 & M3 : Measurement Phase
P1, P2 & P3 : Processing Phase
T1, T2 & T3 : Throttling Phase
128 cc
32 cc
128 cc
M1
P1
T1
M2
P2
T2
M3
P3
T3
Time of Execution

Fig. 3. Various phases in the application execution

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.

Figure 3 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.

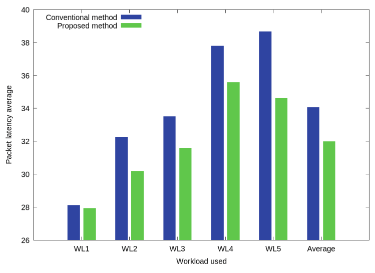

Fig. 4. Overall packet latency

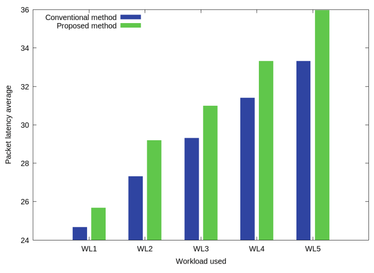

Fig. 5. Throttled packet latency

Figure 4 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 5 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.

5 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, E., 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, M., et al.: Self-tuned congestion control for multiprocessor networks. In: The Seventh International Symposium on High-Performance Computer Architecture, HPCA, pp. 107–118. IEEE (2001)
3. Ebrahimi, E., et al.: Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. ACM SIGPLAN Not. **45**(3), 335–346 (2010). ACM
4. Ausavarungnirun, R., 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)
5. Nychis, G.P., et al.: On-chip networks from a networking perspective: congestion and scalability in many-core interconnects. ACM SIGCOMM Comput. Commun. Rev. **42**(4), 407–418 (2012)
6. Chang, K.K.-W., et al.: HAT: heterogeneous adaptive throttling for on-chip networks. In: IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 9–18. IEEE (2012)
7. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 86–96. IEEE (2013)
8. Binkert, N., et al.: The gem5 simulator. ACM SIGARCH Comput. Architect. News **39**(2), 1–7 (2011)