Grant Agreement number: 248972

Project acronym: **NaNoC**

Project title: "Nanoscale Silicon-Aware Network-on-Chip Design Platform"

Seventh Framework Programme

Funding Scheme: Collaborative project

Theme **ICT-2009.3.2 Design of semiconductor components and electronic based miniaturised systems**

Start date of project: 01/01/2010 Duration: 36 months

D 2.2 *A Low Overhead Congestion Management Mechanism able to Reduce Head-of-Line Blocking*

Due date of deliverable: Month 24
Actual submission date: Month 24

Organization name of lead beneficiary for this deliverable: SIMULA
Work package contributing to the Deliverable: WP2
Contributing partners: SIMULA, UPV

Dissemination Level		
PU	Public	x
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including theCommission Services)	

APPROVED BY:

Partners	**Date**
All	January 11, 2012

INDEX

ABSTRACT

In this report we describe the efforts made towards identification of congestion and its side effects of Head-of-Line (HoL) blocking. Different congestion management strategies and HoL blocking avoidance techniques are explored and evaluated. Congestion management is still not mature enough in the Network-on-chip field and these efforts should be seen as preliminary and exploratory research.

First congestion situation is identified in a CMP configuration where multiple applications are run in a virtualized environment, where applications run in isolated regions but sharing common resources like memory controllers. Then, we describe three approaches followed within the project to address the congestion issue. First, we propose BAHIA and BAHIA-2 mechanisms to address not the congestion situation but the side effects congestion causes, the HoL-blocking between congested and non-congested packets. In BAHIA, end nodes detect bursty traffic and sources of the bursty traffic are informed. Those nodes separate the traffic in two virtual networks, one used exclusively for bursty flows. In BAHIA-2, congestion is detected within the network and end-nodes are informed (similarly as done in OCRL). Source nodes separate traffic in two virtual networks as done in BAHIA. Finally, the OCRL mechanism detects congestion within the network and source end nodes contributing to the congestion situation are informed. These nodes limit their injection towards the congestion spot, thus relieving the congestion situation With these mechanisms, initial congestion identified can be eliminated (with OCRL) or its negative effects eliminated (with BAHIA and BAHIA-2).

Both research partners (SIMULA and UPV) delved deeply into this issue (congestion), delivering different, but complementary, approaches. By looking at the buffer occupancy switches to detect congestion, OCRL increased network throughput by as much as 45% compared to a state without congestion control, which is 13% below the injected traffic. BAHIA achieves a 66% increase in network throughput compared to not using the mechanism, which is close to the ideal performance. Both mechanisms requires only a few control packets per congestion event, so the task goal of 90% throughput with 2-3 control packets has been reached. This deliverable is associated with task 2.3.

GLOSSARY

- **CMP**: Chip MultiProcessor
- **MPSoC**: Multi Processor System-on-Chip
- **NoC**: Network on Chip
- **SCC**: Single-chip Cloud Computer
- **DRAM**: Dynamic Random-Access Memory
- **DMA**: Direct Memory Access
- **BAHIA**: Burst-Aware Head-of-line Injection Avoidance
- **OCRL**: On-chip Rate-Limiting
- **MOESI**: Cache coherency protocol based on states Modified, Owned, Exclusive, Shared and Invalid
- **LBDR**: Logic-Based Distributed Routing
- **QoS**: Quality of Service
- **HoL**: Head of Line
- **BNN**: Burst notification network
- **CNN**: Congestion notification network
- **DDR**: Decreased Data Rate

1 Introduction

The high-performance computing domain is taking advantage of the inclusion of multicore solutions in the form of Chip Multiprocessor (CMP) and System-on-Chip (MPSoC) systems. As the integration scale goes further, more cores, nodes, or processing units are expected to be included in the same chip. Examples of the many-core integrated CMPs are the products developed by Intel inside its Tera-scale Computing Research Program such as two prototype chips: The Teraflops Research Chip [3] with 80 cores, and the single chip cloud computer (SCC) [2]. Tilera products, like the Tile-GX [4] with up to 100 cores, represents a good example for a high-end MPSoC system. These systems provide support for the specific needs of the different applications to be run including multimedia, wireless networking, and cloud-computing applications.

Both design platforms, CMPs and MPSoCs, rely on an interconnection network infrastructure that provides the communication between all the processing nodes. This must be a high-bandwidth, low-latency network to avoid slowing down processors while waiting for remote data. Networks-on-chip (NoCs) suit well when a large number of processing nodes are present [6], as is the case of the Intel prototypes and Tilera products. NoC design is challenging due to the tight constraints found in on-chip systems. Thus, a NoC must be simple in its mechanisms exhibiting low hardware overhead, low power demanding, and at the same time performance efficient, not affecting the performance of the applications running on the system. These are factors that are not present in the off-chip domain.

One of the trends followed by current chip designers is the use of a tile-based design, as seen in Figure 1. A typical tile structure consists of a processing core/unit with access to different cache levels and a router to access the on-chip network. When one tile has been designed with its local components, then, it is replicated throughout the chip. Tiles are usually homogeneous in CMP systems. Although MPSoCs are usually built from heterogeneous tiles, they also tend to conform to a regular design pattern. The tiled chip is completed with the addition of memory controllers, usually placed at the chip boundaries, to access the off-chip memory (DRAM), with multiple access points to the chip. This tile-based design method reduces the efforts spent when designing, building and testing a product.

The two platforms, CMPs and MPSoCs, present totally different network traffic patterns. Typical application traffic patterns observable in CMPs are described in [14]. CMP traffic is in general classified into four levels. The first level is the traffic between the CPU and L1 cache, and between the L1 and L2 caches in the same core. This traffic is generally not considered for NoCs because it uses a dedicated bus. In typical designs, the L1 cache is private to the core, and the L2 cache is shared between all cores. Thus, the first traffic level which is relevant to the NoC is therefore traffic from the private L1 cache to the L2 cache in a different core, and the cache coherency traffic between the L2 caches of different cores. Some newer designs include more cache levels in order to improve performance, but the basic idea of traffic levels is still valid. The final traffic level is that between the last level of cache and off-chip memory. This traffic is directed through the memory controller ports connected to the chip.

In addition to this traffic breakdown in CMPs, these systems scale to large sizes and become suitable for concurrently running multiple general-purpose computing applications. Virtualizing the chip increases the utilization due to application-level parallelism. A single chip simultaneously running multiple applications leads to highly unpredictable traffic patterns. To support such operations, there are several key properties in the virtualization challenge that must be inherent in the NoC [6]. Each application must run in isolation, ideally on its own contiguous part of the chip,

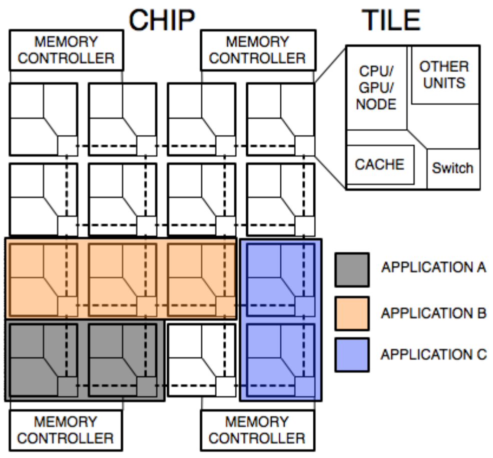


Figure 1: Example of tiled CMP/MPSoC systems

in order to satisfy the requirements of security/privacy and predictability. Additionally, all applications need to access off-chip memory through memory controllers and/or perform maintenance and keep coherency between the different cache levels. Hence, each application must be provided with exclusive access to its private resources, and also with efficient access to shared resources. An efficient chip virtualization strategy including the NoC is critical to achieve these properties.

A typical scenario in a virtualized CMP is depicted in Figure 1. There are three applications sharing the chip, and the memory controllers are accessible through one edge of the chip. The interconnection network at this edge not only has to support traffic from the private application running on the surrounding cores in one virtual network, but also the shared memory controller access from applications located at the other end of the chip. Although the different traffic levels run on different virtual networks, they compete for the same physical channel, thus contributing to create a congestion situation around the access to the memory controller. These congestion situations can degrade the overall chip performance if not properly managed by means of suitable mechanisms.

On the other hand, MPSoCs in general exhibit a different traffic behavior. MPSoC platforms, especially those in the high-performance computing domain, are designed to run specific applications with more dedicated approaches. Accelerator devices, DMA devices, and non-coherent traffic are typical in these systems. In low-end MPSoCs the applications to be run can be known in advance, so that the system can be accordingly tailored to these applications, but this is not the case for high-end MPSoCs where multiple applications (with similar characteristics but different at the end) can be run. In particular, a type of traffic usually present in high-end MPSoCs are bursts that happen when two end nodes intensely communicate during a small fraction of time. This kind of traffic may create temporary hotspots where the traffic is concentrated, thereby leading to the appearance of network congestion that is likely to have a negative impact on the rest of the traffic. As we will see in this document, dealing with the problems derived from congestion can significantly improve the overall chip performance.

Indeed, in this document we describe two complimentary solutions researched within the project to alleviate the negative effects of congestion that may degrade the network performance in both virtualized CMPs and MPSoCs. BAHIA is a mechanism that dynamically detects bursty traffic in the network, then isolating the burst and thus avoiding that non-bursty traffic is affected. The second method, focused on CMPs, is known as OCRL, and deals with congestion by means of a notification mechanism that warns of congestion the sources contributing to create it, in order to throttle packet injection at these sources, thus removing the congestion. Although both methods may work in isolation they can complement each other in order to achieve better performance. Indeed, we extend the BAHIA mechanism to support detection of congestion situations (rather than only bursty traffic). The mechanism is termed BAHIA-2. Preliminary to this, we also analyze the effect of congestion in CMP systems where the number of memory controllers is limited.

To sum up, we have developed the following mechanisms:

- BAHIA (*burst-aware Head-of-line Injection Avoidance*) detects and removes the head-of-line blocking effects of bursty traffic in NoCs. This is achieved at the boundaries of the network thus not increasing NoC complexity.
- OCRL (*On-chip rate-limiting*) monitors buffer occupancy in all switch buffers in the network to detect congestion. Congestion is alleviated through employing rate-limiting at the sources after being notified of congestion by the switches.
- BAHIA-2. Previous BAHIA mechanism is improved to deal also with congestion and Head-of-Line blocking effects within the network.

The rest of the document is organized as follows. First present a study of the impact of congestion in a virtualized CMP scenario in Section 2. In Section 3 the BAHIA mechanism and its extension (BAHIA-2) is described. OCRL is presented in Section 5. Next, performance results are provided in Section 6 and some conclusions in Section 8.

2 CMP congestion study

In this section we present a study on how network congestion affects application performance in a virtualized CMP. A virtualized CMP is a chip multi-processor where several applications are running simultaneously on continuous and disjoint portions of the chip. Traffic local to each application (cache coherency traffic) is isolated within the applications partition on the chip, while access to shared resources such as memory controllers is provided through the global network so that memory controller traffic from one application may pass through partitions belonging to several other applications.

When using virtualization to support multiple concurrent applications on a CMP, the local cache coherency traffic is separated from the memory controller traffic into two different virtual networks by using virtual channels. This separation means that there is limited interaction between the two traffic types. The congestion tree caused by the memory controller traffic cannot directly transfer into the local traffic.

The exact division of link bandwidth between the two virtual channels that share it depends on the virtual channel arbitration mechanism. In general, the arbitration mechanism must be fair, allowing each virtual channel equal access to the link bandwidth [9]. The actual implementation of virtual channels and the arbitration mechanism is subject to numerous variations [6].

A given virtual channel can use any amount of free capacity on the link, but it will always be guaranteed its fair share, which in the case of two virtual channels is 50% of the available bandwidth. Consequently, if the ratio of local traffic to memory controller traffic is significantly skewed in one direction or the other for a given network link, the presence of one type of traffic (the one with the smallest ratio) will have an impact on the other traffic type. The absence of the low ratio traffic will allow the high ratio traffic to consume more of the link bandwidth than if the low ratio traffic is present. In other words, even though the two traffic types are much more separated than would be the case if they shared the same buffers/virtual channels in the network, there is still some degree of interaction between them.

For a CMP where the local traffic is contained within the application partition on the chip, it is only the memory controller traffic that has the potential of affecting other applications. The general understanding when mapping applications to CMPs is that increasing the distance to the memory controller will reduce the performance of the application compared to an application that is located close to the memory controller [5][1]. This seems reasonable because an application located further away will have a longer path, and thus more chances of contention which results in increased queueing time and end-to-end latency. On the other hand, an application located close to a memory controller would only have a few hops and should therefore be guaranteed better throughput and latency. After all, the applications located close to the memory controller have the same probability of accessing each of the channels on the path as the packets coming from applications located further away, giving it an overall higher probability of access.

The above arguments are valid if we consider only the memory controller traffic type. However, for many CMP applications the amount of local traffic will often be more than the memory controller traffic. For the applications located far away from the memory controller this means that the local traffic will get a larger portion of the link bandwidth than the memory controller traffic. As we look at applications located closer and closer to the memory controller, the amount of memory controller traffic will increase within the application partition. Each application partition must support the memory controller traffic from the local application, as well as transit memory controller traffic from the applications located further away. This will reduce the amount of local traffic accepted on the links to a minimum of 50% of the link capacity. This impacts the ability of the application to expediently carry out its assigned tasks. Following this line of arguments, the transit memory controller traffic will have a detrimental effect on the execution of the local application, even though the traffic is separated into different virtual channels. This effect will of course be significantly worse if there is no traffic separation at all.

There is, of course, still merit to the understanding that memory intensive applications should be located close to the memory controller, since the memory controller access also has a significant impact on application performance. The problem we have highlighted in this section is the unfair penalty given these applications by the transit memory controller traffic from other applications. This cannot be solved simply by rearranging the applications on the chip, since moving the memory intensive applications further away from the local memory controllers will reduce their performance. Fair use of the on chip network resources therefore requires a resource management mechanism (like congestion control or changing the traffic priorities) to even out the effect of the transit memory controller traffic and provide predictable CMP performance.

In the next section we present a series of experiments with traffic from real applications to verify our reasoning above and to quantify the effect this interference has on application performance. We also explore changing the relative priorities of the two virtual channels carrying local and memory controller traffic to determine if this simple mechanism can have a positive impact on the variation in application performance.

2.1 Simulation environment

Our simulation framework is a combination of tools chosen to simulate a CMP system as closely as possible. In Figure 2, we present an overview of the simulation environment. Multi2sim [15] is a simulation framework for heterogeneous computing, including models for superscalar, multithreaded, multicore, and graphics processors. It allows one or more applications to run on top of it without booting a guest operating system, and implements emulation of system calls and x86 instructions. Multi2sim implements contexts which define how an application behaves. It is able to model a complete memory hierarchy system integrated into the CMP and its connection to the respective processor cores. Although Multi2sim allows for defining basic interconnection networks (bus and basic point-to-point), we opted to combine Multi2sim with an in-house cycle-accurate flit-level network-on-chip simulator called gNoCsim (developed by *Universidad Politécnica de Valencia*, and being used in the NaNoC project [11] by different partners). gNoCsim is able to simulate the communication and more complex topologies for all the resources in the chip; caches, memory controllers, and processor cores. With this simulator, different configurations of routing strategies, arbitration control, and packet switching policies can be defined, as well as other switch properties like buffering strategies et cetera.

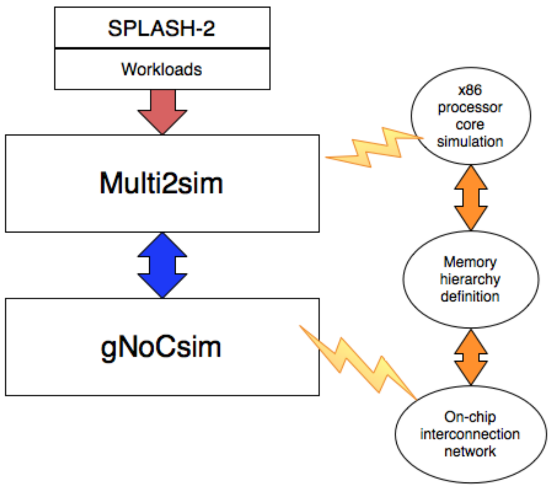

SPLASH-2
Workloads
Multi2sim
gNoCsim
x86 processor core simulation
Memory hierarchy definition
On-chip interconnection network

Figure 2: General overview of the simulation environment

The same simulation environment has been used for the remaining evaluations in this report, with the exception that different traffic generators have been employed for some of the experiments.

2.2 System configuration

For the evaluation process, we modeled a CMP that resembles current chip configurations. This configuration implements a tile-based system, and each tile is composed of a processor core, a private L1 cache, a bank of a L2 shared cache, a memory directory bank to be used with the directory-based MOESI cache coherency protocol, and different configurations of memory controllers. Each memory controller is connected to the main memory with 2 channels (each memory

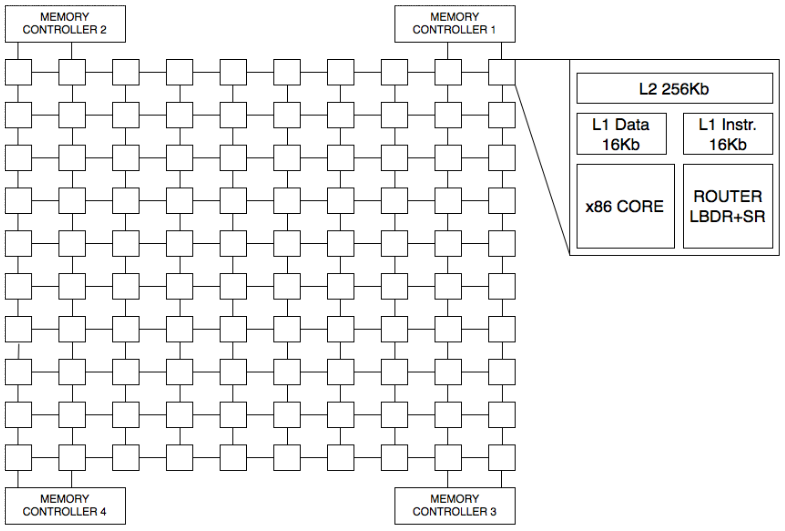


Figure 3: General overview of the CMP model

controller has two access points). The modeled CMP is presented in Figure 3 and a detailed overview of the chip configuration is shown in Table 1.

The parameters for the on-chip interconnection network are shown in Table 2. A 10×10 2-dimensional regular mesh topology was used for the CMP system. The LBDR [13] mechanism was used for the routing purposes as it allows the user to define application domains in an easy way in the mesh, and in combination with the Segment-Based Routing algorithm (SR) [10], LBDR allows close to minimal-paths with deadlock-free routing. Virtual networks are used for different levels of traffic of the memory hierarchy system, implemented as multiple virtual channels (a total of two virtual channels are used).

2.3 Workloads and application-mapping

For the evaluations we use a collection of applications from the SPLASH-2 benchmark [17] with the default parameters defined in [16]. We have evaluated several configurations of multiple application instances mapped to the same CMP for different numbers of memory controllers, in order to detect indications of congestion problems as we scale the number of memory controllers from 1 to 4 (giving 2 to 8 memory controller channels).

The applications are statically mapped to the chip when the experiment is set up. Applications are mapped to completely fill the chip, giving a fair share of cores to each application. Every batch consists of a single application type from the benchmark suite rather than being composed of a collection of mixed applications. This regularity makes it significantly easier to generate relevant statistics and spot trends in the results, such as to get averaged results for the execution time comparison. Running a mix of applications will even out some of the variations of the communication demand over time, but the conclusions will still be the same. See Figure 4 for an example of the mapping of 12 concurrent applications with 2 memory controllers.

Parameter	Configuration
Core	x86
L1 cache	16 KBytes Instructions + 16 KBytes Data Total 32 KBytes per core 2 cycles latency 2-way associativity 64 bytes block size
L2 cache	256 Kbytes per core 20 cycles latency 4-way associativity 64 bytes block size
Main memory	1 Gbyte total 200 cycles latency
Coherence protocol	MOESI CMP, directory-based

Table 1: CMP configuration.

Parameter	Configuration
Topology	10×10 2-dimensional regular mesh
Routing mechanism	LBDR + SR
Packet switching	Virtual cut-through (VCTlite) [12]
Buffer queue size	12 flits
Flit-size	8 bytes

Table 2: Network-on-chip configuration.

2.4 Results

We now present the results from the different evaluation batches. First, we present the scenario with just 1 memory controller with 6, 8 or 12 concurrent applications of the same type. The memory controller was placed like the *Memory Controller 1* in Figure 3. We performed evaluations for all the application types, and similar patterns of congestion problems appear for every application type. We only present the results for the ocean workload.

Figure 5a shows network throughput results for the ocean application with 6 concurrent applications. The point series labeled as *injected* reflect the traffic that the network interfaces try to push into the network, and the other one reflects the traffic that is currently accepted and is being forwarded through the network. The results are averaged over all the application instances. This representation shows that there is no significant gap between the injected and accepted traffic, meaning that congestion is practically non-existent. This is also the case for the other application types in the benchmark.

In Figure 5b the results from running 8 concurrent applications are shown, again for the same application. This figure starts to indicate some minor contention problems. If there is no congestion, the accepted traffic should follow the injected traffic closely as time progresses. In this figure, however, there is a clear gap between the injected and accepted traffic (accepted being lower) which indicates that the network is saturated and congestion occurs.

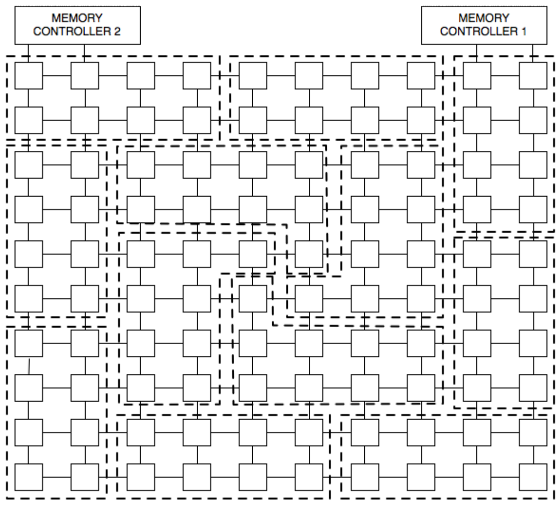


Figure 4: 12 concurrent applications mapped on the system

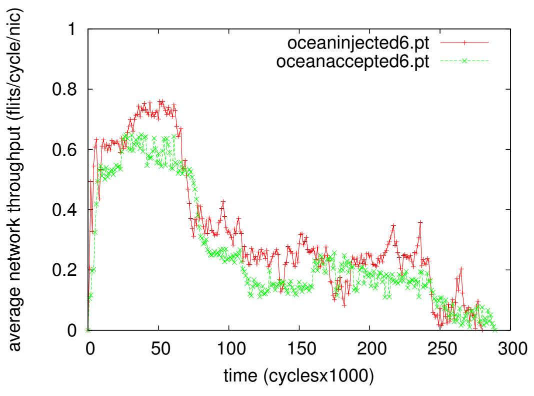
This is even clearer in Figure 5c for the ocean application with 12 concurrent instances. As we can see, there are some significant drops in the throughput for different parts of the execution. In fact, as shown in Figure 5d, we can see the progression of the different averaged execution times of all applications with increasing sharing. Please note that the variation between different runs of the same application at the same location is insignificant, so the baseline for this figure is the execution time of just 1 instance closest to the memory controller. An increase of more than 8% in execution time for 12 concurrent applications for several applications is visible.

To further evaluate how the problem scales we evaluated a batch of 12 concurrent applications and present the results for the ocean workload (which behaves similar to the rest of applications) in Figure 6a. We have increased the number of memory controllers to 2, located like *Memory Controller 1* and *Memory Controller 2* in Figure 3.. As expected, adding more memory controllers alleviates the congestion problem and balances the traffic in the network. This is the most obvious solution to the congestion problem, add more memory controllers if it is feasible in the design process.

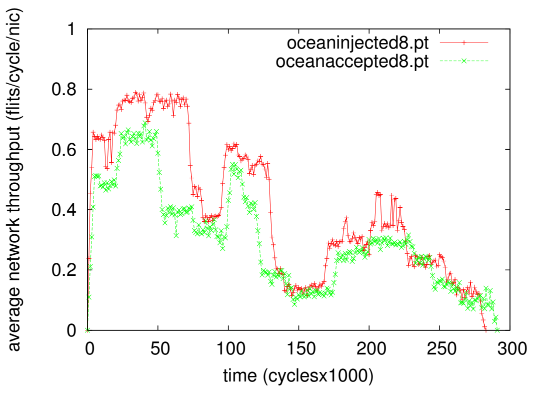
However, by increasing the number of concurrent applications in the system (more application instances, each with fewer cores), we can recreate the previous congestion scenario. Figure 6b shows the results for 16 concurrent applications and 2 memory controllers.

Although not as significant as the case with 12 concurrent applications and 1 memory controller, the network suffers from congestion problems. This is even clearer in Figure 6c where execution time results are displayed.

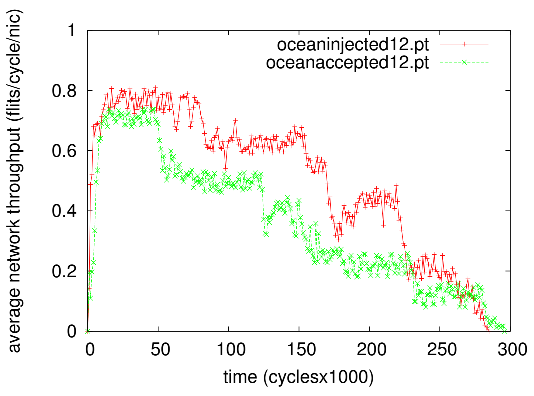
Finally, we place 4 memory controllers in the system like the configuration visible in Figure 3. Again, adding more controllers reduces the congestion problem, and when increasing the number of applications to 32, the congestion problem re-asserts itself. In Figure 7a we show the results for 32 instances of the ocean application. The results show that congestion problems can appear again in the event of enough concurrent applications even with 4 memory controllers configured in the system, and the average execution time is also penalized, as seen in Figure 7b.



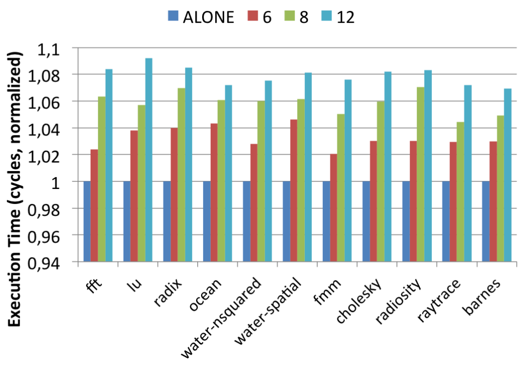
(a) Throughput, 6 applications, ocean workload



(b) Throughput, 8 applications, ocean workload



(c) Throughput, 12 applications, ocean workload

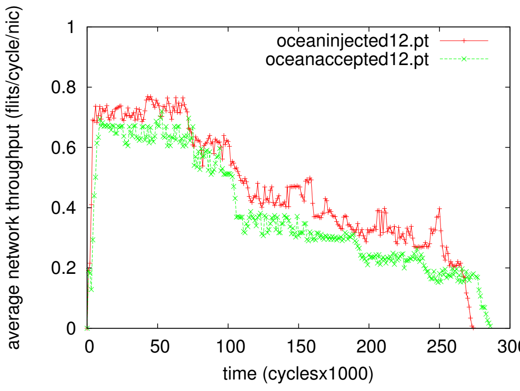


(d) Execution time

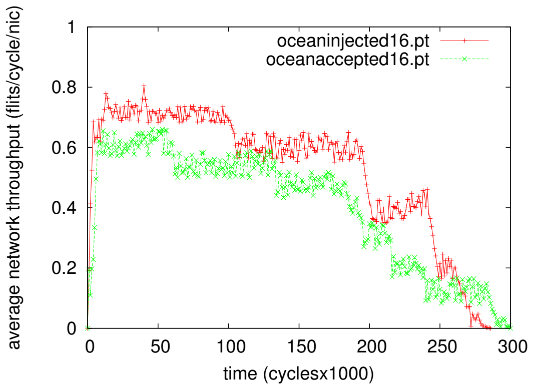
Figure 5: Results for 1 memory controller and different amounts of concurrent applications

Previously, we postulated that the transit traffic has a negative impact on the execution time of the instances. Let us discuss the distribution of the execution time per application instance in the scenario with 12 concurrent applications and 1 memory controller located at coordinates (9, 9) and (8, 9). As seen in Figure 8a which displays the increase in execution time for a fully utilized CMP for the ocean workload, the instances close to the memory controller are penalized more than the ones that are further away. Each bar represents one core in the system (although the evaluation scenario runs 12 concurrent applications, we get the detail to each core) and we can see a variance of 15% between the fastest instance and the slowest one. As the different types of traffic must share a physical channel, and the arbitration policy tries to balance between the different virtual channels, every different type of traffic gets a 50% of the channel share when both traffic types are fully saturated, and thus, the execution time which is linked to the success of delivering local traffic, is penalized. To further evaluate this impact, we evaluated this particular batch together with a simple QoS (quality-of-service) priority system.

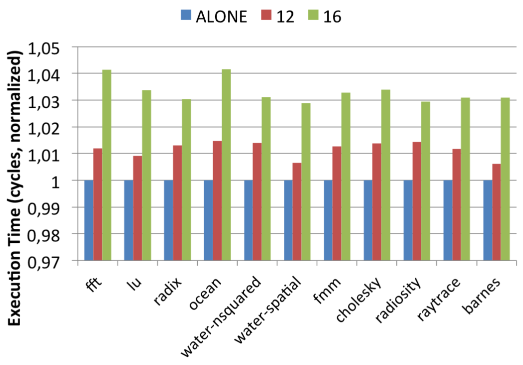
In Figure 8b, local traffic was given higher priority than memory traffic, a 60/40 share. Although



(a) Throughput, 12 applications, ocean workload



(b) Throughput, 16 applications, ocean workload

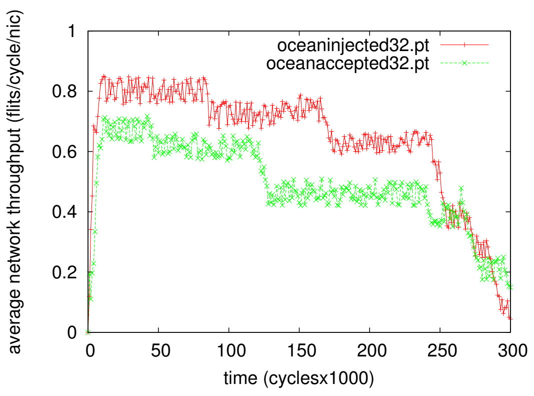


(c) Execution time

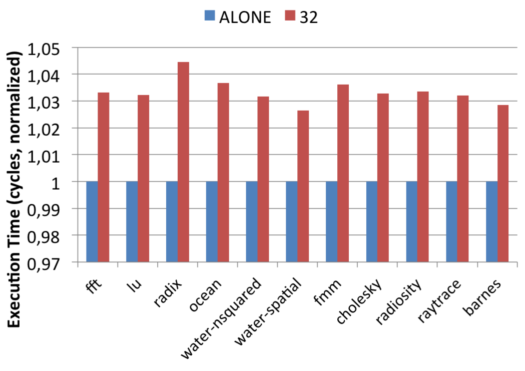
Figure 6: Results for 2 memory controllers and different amounts of concurrent applications

the overall performance has dropped down a bit, compared to the previous case, the gap between the fastest instance and the slowest instance now corresponds to 11%. Still, we need more fine-grained balancing to equalize the performance over all the chip. Compared to the averaged runtime of the ocean case displayed in Figure 5d, which is a 7% increase over the normal execution time, with this priority configuration, now is displayed with a 7.4% increase. The cost of the increase in fairness is slightly worse performance.

A more extreme share can be seen in Figure 8c, which displays a 80/20 share between local traffic and memory controller traffic. Now the application instance farthest away is penalized almost as much as the one located near the memory controller and the gap between execution times is lower than the previous scenario. The worst case is 6.8% difference. The averaged execution time has a 7.55% increase over the normal execution time. Implementing different priorities is not enough to alleviate the impact of congestion in the chip performance. With just one solution, although simple, there are still undesirable results under the premise of certain conditions, in this case, many concurrent applications trying to access a shared chip resource. These findings verify our assertions.



(a) Throughput, 32 applications, ocean workload



(b) Execution time

Figure 7: Results for 4 memory controllers and different amounts of concurrent applications

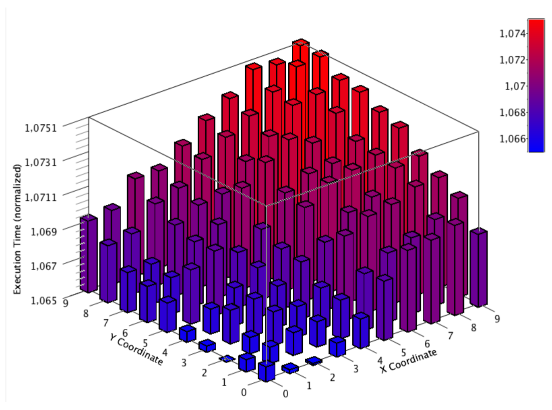
2.4.1 Evaluating the effect of 3-D stacking

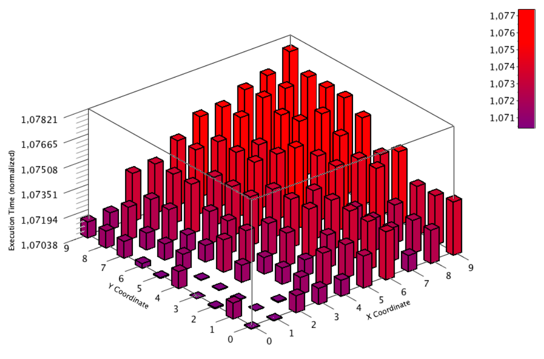
The congestion patterns we have observed in the previous discussion relate closely to the placement of the memory controller. In future networks on chip we expect 3-D stacking to become prevalent. Using this technology that previously-chip memory and can be placed on the chip at a separate chip layer (on top of the regular chip) with vertical links connecting the two layers. This traffic to changes the location of the memory controllers, and depending on the number of vertical connections (TSV) it will have significant impact on system performance. With only a single or a few TSVs the congestion problems will be similar to what we have already observed since there are a few memory controller hotspots that receive large amounts of the traffic. However, TSVs are efficient and relatively cheap, so it is conceivable to have as much as one per core on the chip. In this case the congestion situation alters significantly as it is displayed in Figure 9. This figure is generated on the same premises as the previous figures, except that each core is now connected directly to the memory controller through a TSV. The figure shows that the increase in execution time for the various threads is now much smaller because of the improved memory access.

Summarising, the objective of these evaluation cases was to reproduce scenarios that try to reflect current chip configurations, and to see in the event of full utilization of the chip, if the performance of applications would be affected by congestion problems. The trade-off depends on how much resources are available (in our case, the amount of memory controllers) and there clearly is a need for congestion management strategies that can alleviate the problem with minimal impact on the design of the chip. This is the topic for the remaining sections in this report.

3 BAHIA description

BAHIA (Burst Aware HoL blocking Injection Avoidance) provides a method to dynamically isolate detected bursty traffic in a network. Detection of bursty traffic is performed at the receiving node of the burstiness. End nodes are then notified and the bursty traffic is separated from normal traffic. By doing this, we avoid harmful effects between traffic flows thus increasing performance. BAHIA makes use of virtual networks to separate traffic. This implies that BAHIA needs at least


(a) Normal priority


(b) 60/40 share

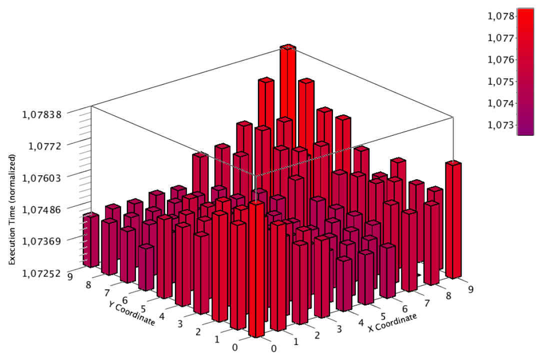

(c) 80/20 share

Figure 8: Execution time distribution for 1 memory controller, ocean workload

two virtual networks: the default virtual network, destined to accommodate normal traffic, and an extra virtual network to accommodate bursty traffic. In a normal operation (no bursty traffic detected) the traffic is injected through the default virtual network. Nevertheless, when a burst is detected, the flow will be mapped into the extra virtual network, thus avoiding HoL blocking between both traffic classes. BAHIA can be divided in three steps as seen in Figure 10: Burst detection, burst notification, and traffic separation.

3.1 Burst detection

As previously said, traffic burst detection is performed at the receiving node. Each node will calculate its receiving traffic rate periodically. If the traffic rate exceeds a threshold value, this node will notify the other nodes it is receiving bursty traffic. Similarly, the end node will detect the end of the bursty traffic. This is detected by comparing the reception rate with a low threshold value. In this case nodes will be informed about this.

Notice that detection of bursty traffic could be done at the sources. However, in that case, combination of bursty traffic at the receiving end node (made of two small traffic rates from two

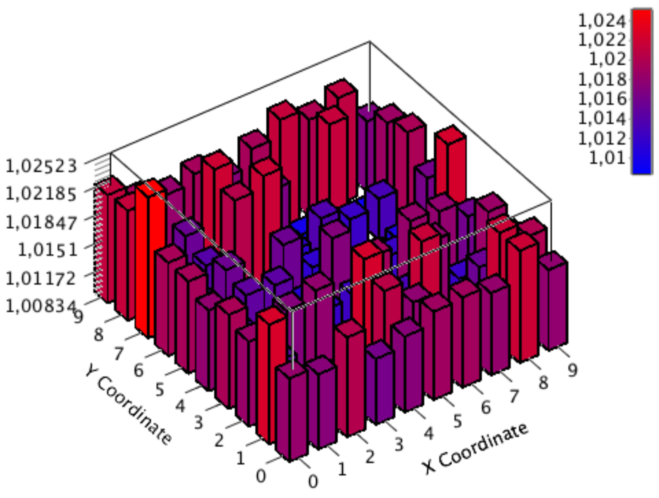


Figure 9: Execution time per thread with one TSV per core.

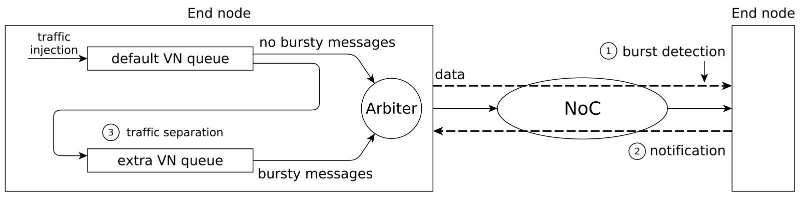


Figure 10: BAHIA behavioral description

different sources) would not be detected. Thus, we opted for a detection at the destination end node.

3.2 Burst detection notification

In order to notify nodes contributing to a traffic burst BAHIA implements a dedicated simple signaling network (burstiness notification network or BNN). This network consists of a one-bit-wide-tree mesh network connecting all nodes as seen in the Figure 11. Each node has a dedicated control network. Therefore, for a 16-node system 16 control networks are used. However notice that a control network is made only of a wire with no logic. The complete BNN can be viewed as a N-bit-link. Every wire in the link corresponds to a node in the network, thus this is equivalent to having a one-bit-wide dedicated wire for every node in the network. Any node will notify the rest of nodes by setting to high value its BNN wire while the burst is present at its receiving channel. With the BNN notification network a node will notify the rest of nodes in few cycles. The processing of this signal and the hardware required for its implementation is negligible. For the area overhead of the BNN, we have synthesized a 4-stage pipelined wormhole switch and added the wiring for the 16 1-wire BNNs, obtaining negligible area overheads. Notice that, alternatively, the BNN used in the OCRL mechanism (described later) can be used for the purposes of notifying bursty traffic events. Also, we analyzed the effect on performance of a slower notification network, achieving similar results.

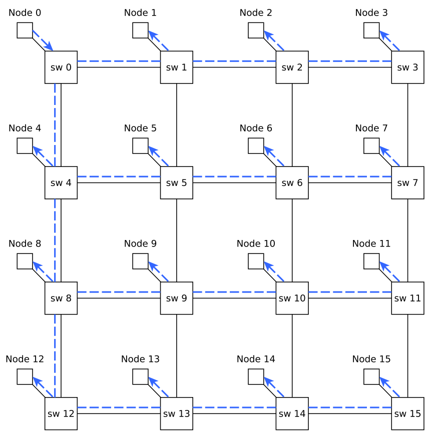


Figure 11: Node 0 communicates burst events through this 1-bit network

3.3 Traffic separation

As a first step every node implements a bitmap of as many bits as nodes exist in the network (burstiness bitmap). Each bit corresponds to a node in the network as seen in Figure 12. When a node receives a traffic burst this node will notify it to all nodes through its BNN wire. The rest of nodes will detect a high value in the BNN wire that corresponds to this node. When this occurs, every node will set to one the corresponding bit of the burstiness bitmap.

At allocation stage, all messages are queued into the default virtual network. Nevertheless, at injection time, every message is checked for its destination. In case the packet to be injected is destined to an affected node according to the burstiness bitmap, this message will not be injected. Instead, this message will be transferred to the extra virtual network. Obviously, packets destined to a node with its burstiness bit reset, will be injected from the default VN and will be forwarded through the network using the default VN.

In Figure 12 we can see an example where the sender node has messages queued for node 1, 5 and 6 in the default virtual network. Currently the sender node is about to inject a message destined to end node 5. As can be seen in the burstiness bitmap, node 5 previously notified was receiving bursty traffic, so messages destined to node 5 must be reallocated to the extra virtual network. Just before injecting, the arbiter of the sender node checks the burstiness bitmap and transfers the messages to the extra virtual network for a later injection.

Once a node notifies that bursty traffic has been dissipated (reseting its BNN wire), the remaining nodes will reset the corresponding bit in their bitmaps and new messages allocated for this node will be injected through the default virtual network. However, out of order issues could arise which is treated in the next section.

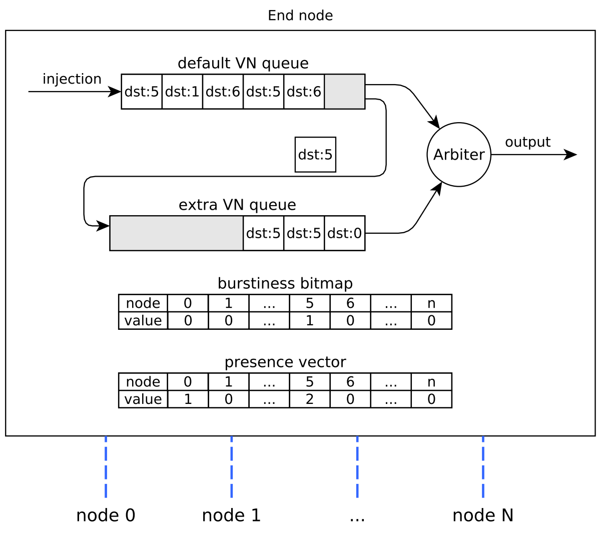


Figure 12: Flow followed by messages in an end node

3.4 Out of order avoidance

In a normal situation (no bursty traffic detected) all traffic is sent through the default virtual network. When a node sends a burst notification, from that moment all traffic destined to this node will be allocated in the extra virtual network. Nevertheless, at this point it is usual that previous messages destined to this node had been already queued in the default virtual network. If the source node just maps new allocated messages to the node receiving bursty traffic on the extra virtual network there will be messages destined to that node queued in both virtual networks. As the arbiter is implemented as a simple round-robin, this may introduce out of order injection of messages destined to the same end node.

In order to avoid this harmful effect a post-processing mechanism has been adopted. The post-processing mechanism essentially consists in providing the arbiter with some additional intelligence in order to evaluate whether a message should be injected through the default virtual network or, if necessary, moving flits (changing its pointers) destined to nodes receiving bursty traffic from the default virtual network to the extra virtual network. To carry this out, before injecting the message into the network the extra virtual network queue is inspected. If there are flits destined to this node the message is moved to the extra virtual network for preserving the order of injection. This behavior in the arbiter would be represented with the next pseudocode (when dealing with the packet at the head of the default VN):

```
if(isNodeReceivingBurst(msg.destination) || numFlitsInExtraVN(msg.destination) >
    0) {
  moveMessageToExtraVN(msg);
  injectFromExtraVN();
} else {
  injectFromDefaultVN();
```

}

However, is necessary to know whether a destination node has pending flits queued in the extra virtual network. To do this, every node in BAHIA implements a presence vector. This vector contains an element per node in the network and every element represents a counter of how many flits destined to this node are queued in the extra virtual network. Obviously, every counter is incremented when a new flit is post processed and moved to the extra VN. In the same sense, the counter is decremented when a flit is injected from the extra VN. Notice that once a burst is no longer detected, an end node may still inject messages through the extra VN as there are remaining flits in the extra queue.

4 BAHIA-2

As will be seen later, BAHIA works well in an environment where bursty traffic is common. However, bursty traffic is not the only source for HoL blocking effects. Congested situations do not form only from bursty traffic. In order to deal with HoL-blocking produced by in-network congestion situations, we extend the previous mechanism and adapt it to a real congestion-aware HoL-blocking removal mechanism. BAHIA-2 will detect in-network congestion ports and will notify all the sources. The end nodes will react similarly as how they do in BAHIA, that is, separating the traffic in two VNs one for normal traffic and one for congested traffic. By doing this, HoL-blocking is removed.

4.1 Congestion detection

Since detection of congestion is now carried at switches, we move the detection logic to switches. Also, as now in BAHIA-2 we detect congestion rather than bursty traffic, we need a different detection mechanism. The new mechanism basically consists in measuring how long messages are queued at the switches waiting for winning an output port. If more than two messages wait longer than a determined threshold, then we assume the output port they request is congested. In that situation, a notification is triggered to the end nodes (possibly with a modified BNN or using the control network assumed in the OCRL mechanism).

The mechanism to detect congestion relies on two counters per output port. One counter keeps track of the number of active requests to that output port (one request every time a message comes and is routed to that output port) and the second counter keeps track the time elapsed in cycles from the time two or more requests are pending. When the first counter reaches two for an output port (at least two messages are competing for the same output port) the second counter is reset and starts counting. Congestion is detected when the second counter reaches a threshold. When the first counter decreases to 1 or zero, the second counter is reset (congestion vanished) and a end-of-congestion notification is triggered to the end nodes.

4.2 Congestion detection notification

In BAHIA-2 congestion notification must inform of the output port congested in a switch. For this purpose, every output port in the network has its own congestion notification 1-wire network (CNN). Similarly as BAHIA, when a switch detects congestion in one of its output ports, it will set the CNN wire associated to the output port (we will have as many CNN wires as nodes times the number of output ports). Alternatively, the control network used in OCRL can be used. As we will see later, the delay in notification does not severely affect the benefits of the congestion management strategy.

4.3 Traffic separation

For traffic separation we must take into account that a message must be moved to the extra virtual network only if this message will go through a congested output port. In order to know which destination nodes will pass through a congested port we implement a small logic block. The block has as input the row and column of the switch (assuming a 2D mesh) and the output port (N, E, W, S) that is congested. The block assumes the use of the XY routing algorithm. From this, it is straightforward to deduce the end nodes that are reached through the congested port. The resulting vector bit is ORed with the vector bit used at the end nodes for identifying congested end nodes (one bit per node). This vector bit is the one used in BAHIA named *burstiness bitmap*. A destination is assumed to be congested if the path to reach the destination passes through one (or more) congested points. For instance, for a E (east) port congested in a switch, only the end nodes in the same row but at the left of the congested switch will set the bit to destinations located at the right side of the congested port. Figure 13 shows two examples of different output ports congested and the end nodes that detect some end nodes as congested.

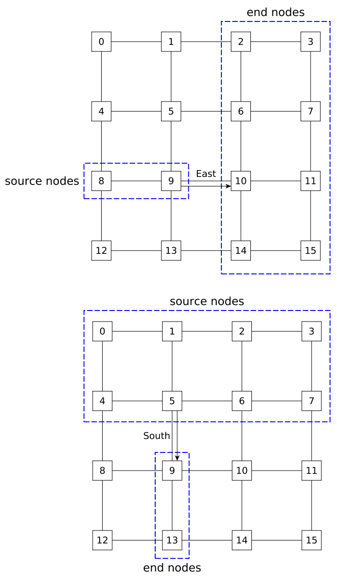
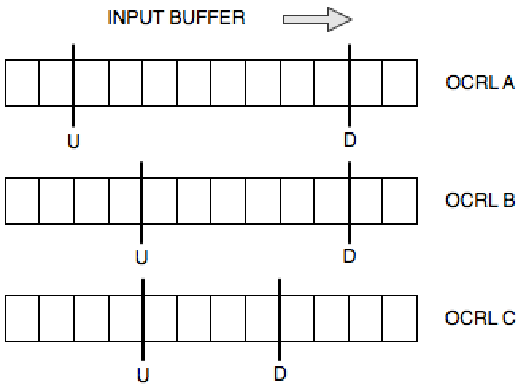


Figure 13: BAHIA-2 examples of congested destinations (XY routing assumed).

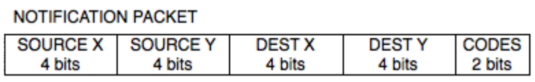
5 OCRL

The OCRL mechanism is an injection throttling mechanism where the switches notify the sources to adjust the traffic rate in order to remove the congestion present in the network. It is compatible with both wormhole and virtual cut through switching, and is composed of three basic elements (although congestion detection is easiest with the large buffers of virtual cut through switching):

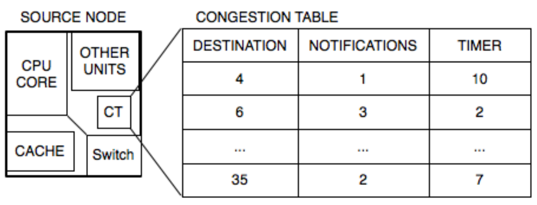
- Every input buffer at each switch has two control thresholds in order to detect the presence and the absence of congestion.
- Notification packets are delivered through a control network that matches the current data network topology, which is a 2-D mesh.
- Every processing node implements a congestion table to keep track of the possible destinations affected by the congested spots in the network.



(a) Thresholds placement



(b) Notification packet



DESTINATION	NOTIFICATIONS	TIMER
4	1	10
6	3	2
...	...	...
35	2	7

(c) Congestion table

Figure 14: OCRL mechanism

The first element is implemented with two thresholds at the input buffer of a switch. Figure 14a shows different placements used in the evaluations. An upper threshold to detect network congestion (U label in the figure), and a lower threshold (D label in the figure) to detect the removal of congestion. A control logic monitors these two thresholds. When the buffer occupancy exceeds the upper threshold, the input port of the switch enters the congested state. In this state it will notify through the control network the source(s) of the packet(s) currently allocated in the congested buffer. Upon notification reception, the source(s) will trigger the injection throttling mechanism. The input port switch remains in this state until the buffer occupancy decreases below the D threshold and, then, the input port switch returns to its normal state.

The notification process is the second element of the mechanism. The switches notify the sources by sending small congestion notification packets on a parallel control network. In modern CMP systems, such a control network will often be in place to support fault detection, configuration, and other management tasks. The format of the packet for this control network is shown in Figure 14b. The first 8 bits define the coordinates of the source where the notification packet is headed. The next 8 bits define the coordinates of the destination to be notified as congested. The last 2 bits define operation codes. In this method we are defining just 1 code, the congested status. Notification packets will be sent through an ad-hoc control network designed with the same philosophy as the 18-bits wide dual bus proposed in [7], conforming the average latency as $number\_of\_nodes/t_s$ being t_s the average latency for a notification packet to cross a switch.

The third element of the mechanism is the congestion tables at the source nodes, shown in Figure 14c. Each entry is composed of a destination field, a field containing the number of notifications received, and a timer value which is incremented every cycle. When a node receives a congestion notification, it creates an entry in the table setting the number of notifications to 1 and setting the timer for that destination to zero. If a node receives more notification packets for the same destination before the timer reaches a maximum timer value, the notification counter is increased and the timer value is reset. If an entry reaches the maximum timer value, then the congestion has vanished and the entry is removed. The maximum timer value is set in our evaluations to the average latency displayed by the control network, in order to emulate a notification packet crossing the network to inform of a non-congested status.

Based on the number of notifications, the node injects messages to the congested node D using the following formula:

$Traffic\_rate(D) = max\_traffic\_rate - (number\_of\_notifications(D) \times DDR)$

where DDR is a constant value representing the amount of traffic decreased for each received notification. DDR constant value is set differently for each concurrent application scenario. It was obtained from several tests to remove unpredictability, and it was computed as the inverse value of the average number of notifications received for each congested situation in a certain destination.

If the actual traffic rate reaches a zero value, it will stay with that value until the timeout occurs. When the timer expires, the table entry is removed and the traffic rate for the destination is increased at the rate of DDR/cycle in a similar manner up to max_traffic_rate (typically 1flit/cycle).

6 Results

In this section we present the evaluations performed in real traffic and bursty traffic scenarios and an assessment of the hardware overhead for the different mechanisms proposed in the project for congestion and HoL-blocking removal (OCRL, BAHIA, and BAHIA-2).

6.1 Results for OCRL mechanism

To evaluate OCRL we use real application traffic. The system configuration implements a tile-based system like in Figure 1, but with a 10×10 2-D mesh, and each tile is composed of a processor core, a private L1 cache, a bank of a L2 shared cache, a memory directory bank to be used with the directory-based MOESI cache coherency protocol, and 4 memory controllers. The simulation parameters are identical to the ones used for the CMP congestion study presented in Section 2.

In order to evaluate the real traffic scenario we launched a set of evaluations with 32 concurrent applications with 3 different placements of the congestion thresholds at the input buffers as shown in Figure 14a, which stands for OCRLA, OCRLB, and OCRLC labels at the figures.

In Figure 15 we can see the effect of sharing NoC resources with other applications on the same chip. Please note that the variation between different runs of the same application at the same location is insignificant, so the baseline for this figure is the execution time of just 1 instance closest to one of the memory controllers. As it is seen, there is an average of 1.5% reduction in the execution time for the 3 variants when using the congestion time. Notice that this reduction can make a difference in the total execution time of the system while improving chip utilization if we aggregate all the averaged time reductions. Real-time constraints that influence certain traffic types can benefit also for every reduction.

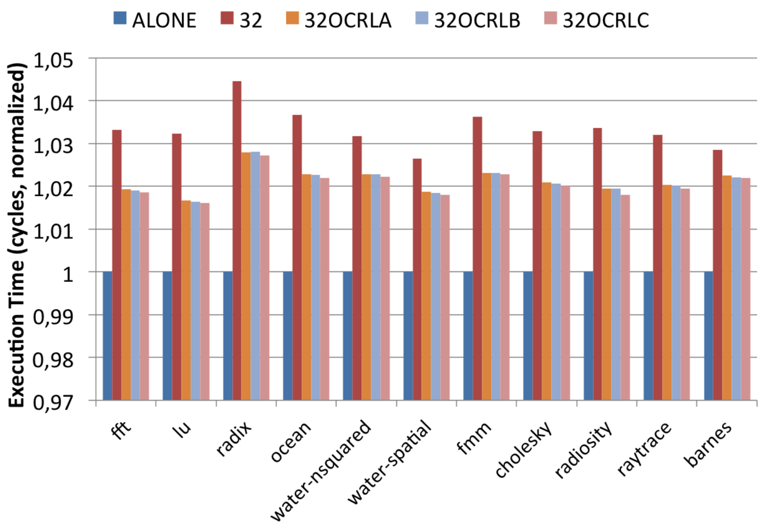


Figure 15: Execution time for 32 concurrent different applications

We performed evaluations for all the application types, and similar patterns of congestion problems appear for every application type. We only present the results for the ocean workload. In Figure 16 we show the results for network throughput. The point series labeled as *injected* reflect the traffic that the network interfaces try to push into the network, and the one labeled as *accepted* reflects the traffic that is currently accepted and is being forwarded through the network with no congestion control. As shown in the figure, there is an important drop due to congestion between the cycle 130k and cycle 250k. All OCRL variants are able to alleviate the congestion drop by 15% in average, although OCRLC seems to perform better. The variant OCRLD is OCRLC in which we modeled the latency of a faster control network than the one that was set up for previous evaluations. As it can be seen, OCRLD has almost zero congestion and is a demonstration on how to tuning the parameters of OCRL in the CMP scenario will lead us to better results.

6.1.1 Area overhead analysis

In order to assess the control network and the logic implementation cost in terms of area, OCRL was integrated in the virtual channel-based switch presented in [8] and it was supported by a control

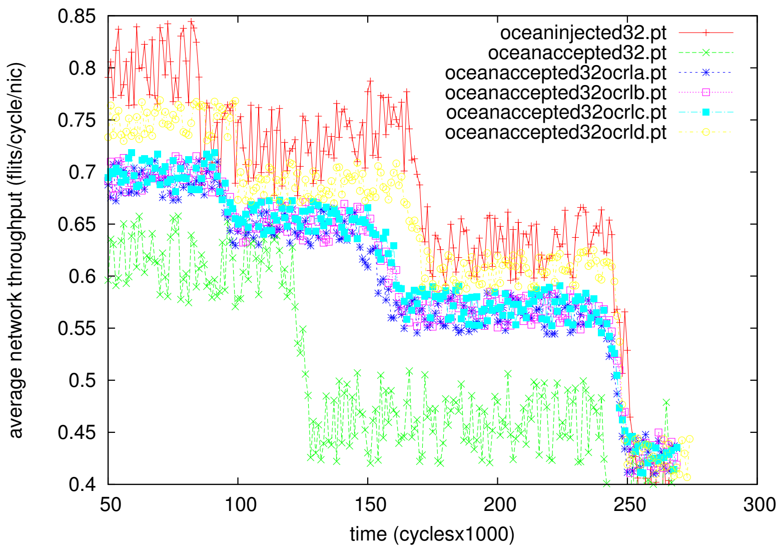


Figure 16: Averaged network throughput for ocean workload

network featuring 18 control bits like the one shown in [7]. To notice that the adopted control network also features fault-tolerance properties. The synthesis was performed with a low-power 65 nm technology library. As a result, the OCRL mechanism and the control network requires a 9% of area overhead with respect to the baseline virtual channel-based switch.

6.2 Results for BAHIA

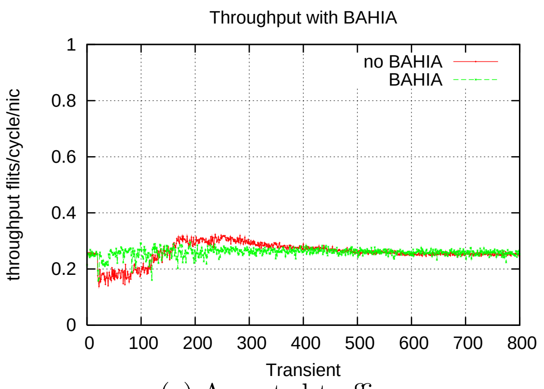
For the bursty traffic scenario, the gNoCsim network simulator was used for the evaluations. In the case of BAHIA, the configuration consists of a 2D mesh with 64 switches arranged in a 8×8 distribution where each node is attached to a switch. Regarding to the traffic pattern generated, each node generates a 0.2 flit/cycle baseline/background traffic following an uniform pattern and, when simulation reaches cycle 10000, traffic bursts are sent to 4 different nodes following a 4-to-1 strategy (each hotspot node will receive traffic from 4 nodes) at 1 flit/cycle until cycle 20000 is reached. Meanwhile, the rest of nodes will continue sending at a 0.2 flits/cycle. See Table I for all configuration parameters. The table shows the upper and lower thresholds used to detect the bursty traffic.

Parameter	Value	
Topology	8x8 2D regular mesh	
Virtual networks	no BAHIA	2VN
	BAHIA	1 default VN + 1 extra VN
Packet switching	VCT	
Flit size	5 bytes	
Packet size	10 flits	
Message size	40 flits	
BAHIA		
Upper Limit	0.7 flits/cycle	
Lower Limit	0.2 flits/cycle	
Polling interval	500 cycles	
Notification delay	1 cycle	

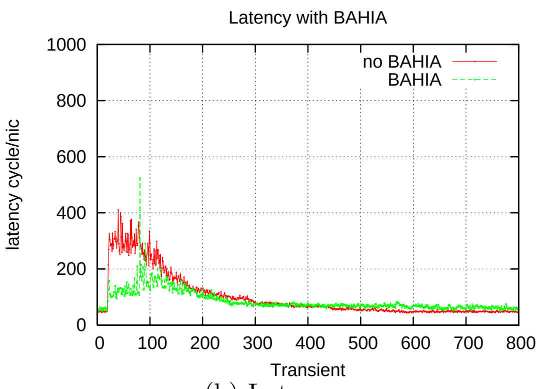
Table 3: Simulation configuration for BAHIA and BAHIA-2 mechanisms.

In Figure 17.(a) we see the accepted network traffic. As can be seen, the simulation begins with a background traffic of 0.2 flits/cycle approximately. However, starting at transient 20 (each transient is made of 500 cycles, thus being cycle 10000), four nodes receive bursty traffic, each from four different nodes. Nodes receiving the burst are not able to dispatch bursty traffic thus contention appears. In no-BAHIA NoC both virtual networks are affected equally since there is no traffic separation, either bursty traffic and normal traffic is allocated without distinction into any virtual network so performance degradation appears in both virtual networks due to HoL blocking effects. Nevertheless, in the case of BAHIA NoC, when bursty traffic is detected at the end nodes, sender nodes are notified so this bursty traffic is isolated into the extra virtual network so there is no HoL blocking in the default virtual network, improving performance in the overall network. Average network latency, shown in Figure 17.(b), behaves similarly, achieving best performance for the implementation with BAHIA. In this case, non-burst traffic latency keeps roughly unaltered.

In Figure 18 a study of the robustness of BAHIA is shown. For these simulations the delay of the notification has been varied between 1 (value in previous simulations) and 16 cycles. We can see in the results that increasing the notification delay has a negligible effect in the effectiveness of the mechanism.

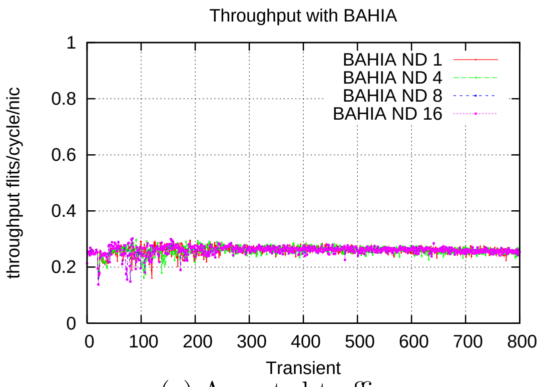


(a) Accepted traffic

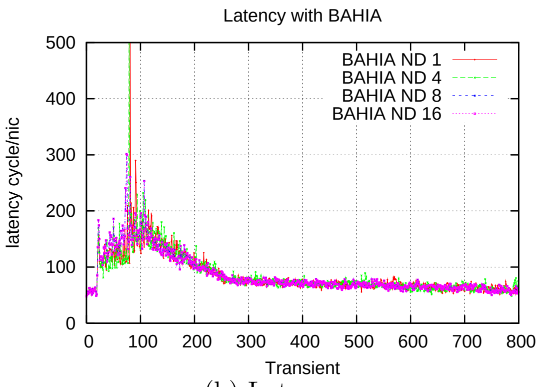


(b) Latency

Figure 17: Accepted traffic and packet latency for BAHIA mechanism.



(a) Accepted traffic



(b) Latency

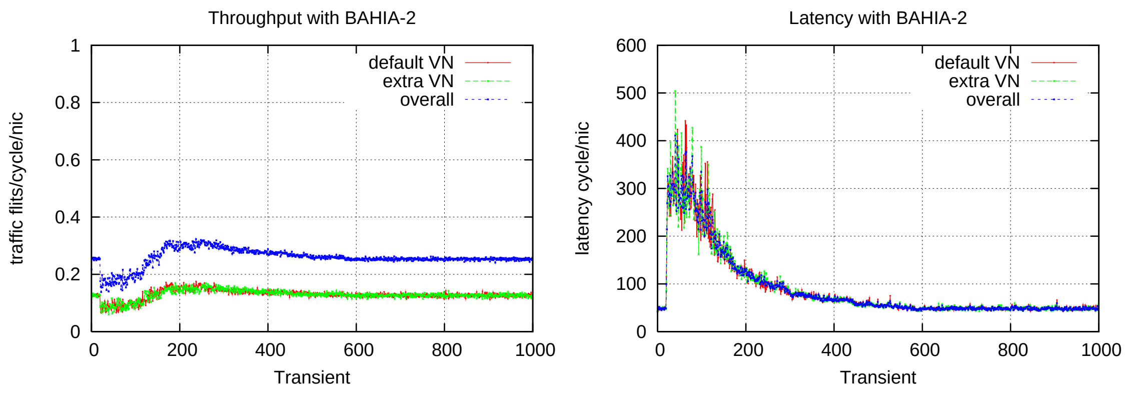
Figure 18: Accepted traffic and packet latency for BAHIA mechanism. Different notification delays.

6.3 Results with BAHIA-2

BAHIA-2 can be configured according to the threshold used to trigger the congestion notification (time elapsed more than two messages block for an output port). This parameter and the delay of the notification are variables that potentially affect the behavior of the mechanism. To test these variables, several simulations have been carried out with different values.

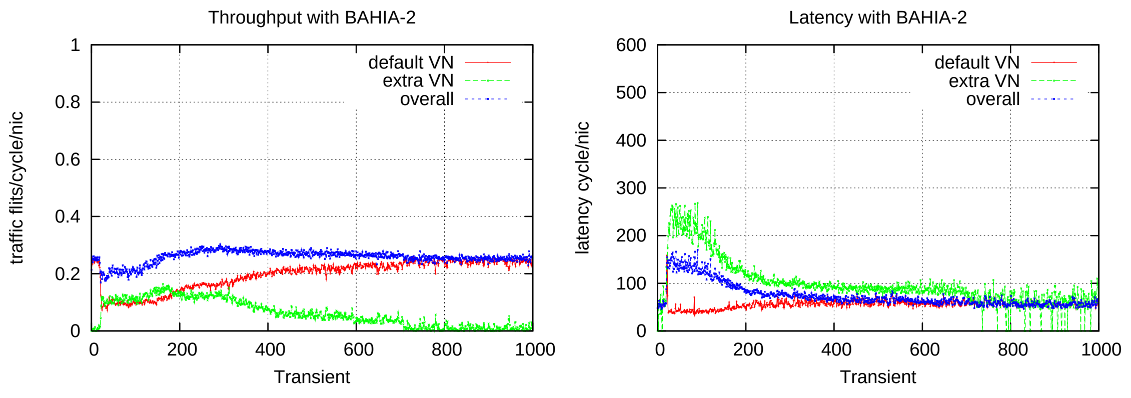
In Figure 19 we see the scenario without BAHIA-2. As can be appreciated, both virtual networks experiment the same behavior as traffic is randomly distributed between both virtual networks.

In Figure 20 we can see latency and throughput for the same network but, in this case, with BAHIA-2. In this case the notification is triggered when more than two messages are competing for an output port more than 50 cycles. A conservative notification delay of 10 cycles is assumed. Similarly, Figure 21 shows results for a delay in notification of 8 cycles, and Figure 22 shows an aggressive scenario where detection threshold is set to only 10 cycles and propagation is set to 1



(a) Throughput (b) Latency

Figure 19: Throughput and latency for an 8x8 network without BAHIA-2



(a) Throughput (b) Latency

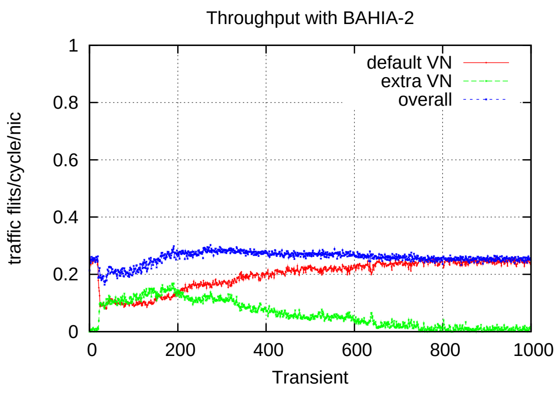
Figure 20: BAHIA-2 throughput and latency with a congestion detection threshold of 50 cycles and a notification delay of 10 cycles.

cycle. As can be deduced, the BAHIA-2 mechanism is largely insensitive to the threshold values.

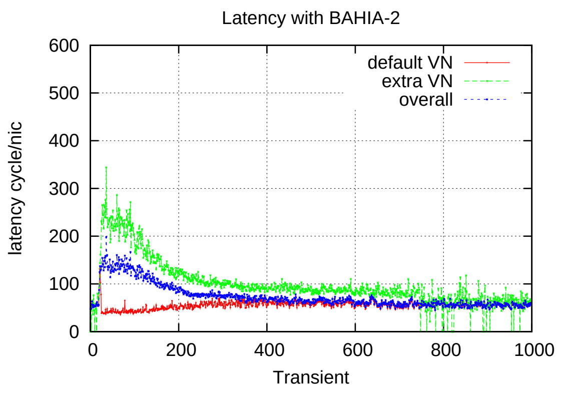
7 Exporting to CEF

The results from network simulations may reveal traffic hotspots in the network on chip caused by specific traffic patterns or burst patterns. This information can be useful for the designer to tune the network on chip implementation in order to minimise the probability and impact of network congestion.

Currently such information is not supported in the CEF standard, but by utilising the option

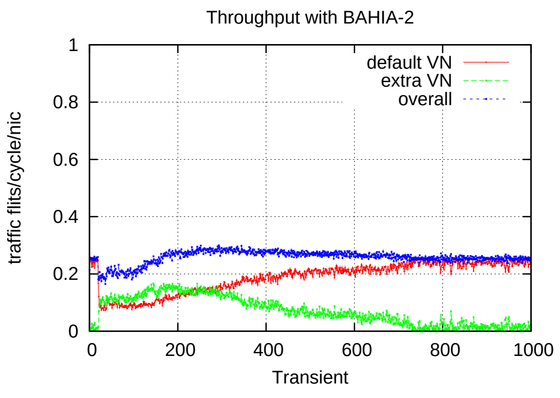


(a) Throughput

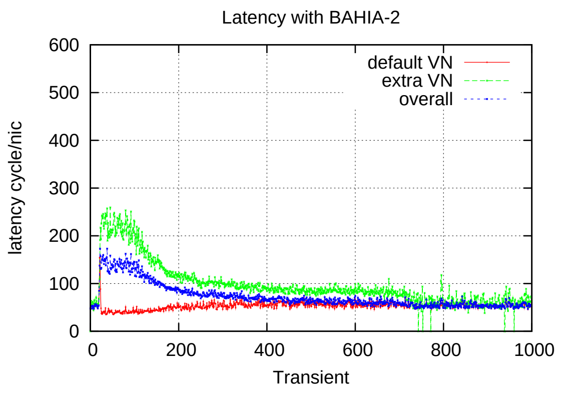


(b) Latency

Figure 21: BAHIA-2 throughput and latency with a congestion detection threshold of 50 cycles and a notification delay of 8 cycles.



(a) Throughput



(b) Latency

Figure 22: BAHIA-2 throughput and latency with a congestion detection threshold of 10 cycles and a notification delay of 1 cycles.

of vendor specific tags, it is possible to append this information to the switch specifications in the CEF file format.

The data entry requires only two tags, $< nodeid >$ and $< congestion\_level >$. If desired, and if this information is available from the simulator results, a third tag can be added to indicate whether a specific note is the root of a congestion tree, $< congestion\_root >$. The congestion route is the point where all the buffers for the port are full, while at the same time the link is fully utilised. This is the bottleneck that causes the congestion tree to build up throughout the network, and dealing with this is the first step to reducing network congestion from the design point.

8 Conclusions

It is well-known that unpredictable or bursty traffic patterns may lead to network congestion which can impact on the execution time of the running applications. This document has presented the research effort within the project to address congestion and HoL blocking effects. To handle the dynamic traffic variation in CMPs, an injection limiting approach has been developed. To handle the bursty traffic in MPSoCs, BAHIA was employed to detect and move bursty traffic to a separate virtual network. In both cases the proposed solutions were able to greatly alleviate network congestion. For the CMP experiments with real traffic traces, the proposed solution was able to increase network utilization with 45% which resulted in a 1.5% reduction in application execution time compared to a fully utilized chip without congestion control. For synthetic traffic, BAHIA solution was able to completely remove the negative effects of the traffic bursts. Also, BAHIA-2 mechanism can handle in-network congestion by addressing the HoL-blocking effect it causes to non-congested traffic. All the mechanisms are within 90% of optimal throughput without congestion (OCRL is slightly below, but additional tuning will increase this), and require a low number of control packets. This is within the requirements for deliverable 2.2, so network-on-chip congestion control has been successfully addressed in the NaNoC project.

References

[1] Guangyu Chen, Feihui Li, S. W. Son, and M. Kandemir. Application mapping for chip multiprocessors. *Proceedings of the 45th annual conference on Design automation - DAC '08*, page 620, 2008.

[2] Intel Corp. The single-chip cloud computer. Available at `http://techresearch.intel.com/ResearchAreaDetails.aspx?Id=27`.

[3] Intel Corp. Teraflops research chip. Available at `http://techresearch.intel.com/ProjectDetails.aspx?Id=151`.

[4] Tilera Corp. Tilera tile multicore processors. Available at `http://www.tilera.com/products/processors/TILE-Gx_Family`.

[5] Reetuparna Das, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. Application-to-core mapping policies to reduce interference in on-chip networks. Technical report, SAFARI Technical Report No. 2011, 2011.

[6] José Flich and Davide Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010.

[7] Alberto Ghiribaldi, Daniele Ludovici, Michele Favalli, and Davide Bertozzi. System-level infrastructure for boot-time testing and configuration of networks-on-chip with programmable routing logic. In *VLSI-SoC*, pages 308–313. IEEE, 2011.

[8] F. Gilabert, M. E. Gómez, S. Medardoni, and D. Bertozzi. Improved utilization of noc channel bandwidth by switch replication for cost-effective multi-processor systems-on-chip. In *Proceedings of the 2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '10, pages 165–172, Washington, DC, USA, 2010. IEEE Computer Society.

[9] N. Kavaldjiev, G.J.M. Smit, and P.G. Jansen. A virtual channel router for on-chip networks. In *SOC Conference, 2004. Proceedings. IEEE International*, pages 289–293. IEEE, 2004.

[10] A. Mejía, J. Flich, J. Duato, S. A. Reinemo, and T. Skeie. Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. *International Parallel and Distributed Processing Symposium*, 0:84, 2006.

[11] NaNoC project. Nanoc design platform. Available at `http://www.nanoc-project.eu`.

[12] S. Roca, J. Flich, F. Silla, and J. Duato. Vctlite: Towards an efficient implementation of virtual cut-through switching in on-chip networks. In *International Conference on High Performance Computing (HiPC)*, pages 1–12, 2010.

[13] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In *NOCS '10: Proceedings of the 4th ACM/IEEE International Symposium on Networks-on-Chip*, pages 25–32, 2010.

[14] V. Soteriou and L. Peh. A Statistical Traffic Model for On-Chip Interconnection Networks. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 104–116. IEEE, 2006.

[15] R Ubal, J Sahuquillo, S Petit, and P López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing*, 2007.

[16] Multi2sim Wiki. Splash2 execution commands. Available at `http://www.multi2sim.org/wiki/index.php5/SPLASH2_Execution_Commands`.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, New York, NY, USA, 1995. ACM.