UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Universitat Politècnica de València

Departamento de Informática de Sistemas y Computadores

Improving Networks-on-Chip Performance in Multi-Core Systems

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Author

Miguel Gorgues Alonso

Advisor

Prof. José Flich Cardo

July 2018

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

ACK	**ACK**nowledgement
CDG	**C**hannel **D**ependency **G**raph
CMP	**C**hip **M**ulti **P**rocessor
CP	**C**ongested **P**oint
CS	**C**ircuit **S**witching
CSP	**C**ircuit **S**etup **P**eriod
DI	**D**omain **I**dentifier
EPC	**E**nd **P**oint **C**ongestion Filter
EVC	**E**xpress **V**irtual **C**hannel
FA	**F**ully **A**daptive
GHz	**G**iga **H**ert**z**
HoL	**H**ead-**o**f-**L**ine
HPC	**H**igh **P**erformance **C**omputing
ICARO	**I**nternal-**C**ongestion-**A**ware HoL-blocking **R**em**O**val
LLC	**L**ast **L**evel **C**ache
LT	**L**ast **T**oken
MPSoC	**M**ulti **P**rocessor **S**ystem **o**n **C**hip
MC	**M**emory **C**ontroller
MHz	**M**ega **H**ert**z**
MLCU	**M**emory **L**atency **C**ontrol **U**nit
NACK	**N**on-**ACK**nowledgement
NI	**N**etwork **I**nterface
NoC	**N**etwork-**o**n-**C**hip
PC	**P**ROSA **C**ontroller
PR	**P**ROSA **R**outer
PROSA	**PRO**otocol-oriented circuit **S**Withching **A**rchitecture
PROSA DD	**PROSA D**istance **D**riven
PS	**P**acket **S**witching
RECN	**R**egnional **E**xplicit **C**ongestion **N**otification
ResArb	**Res**ource **Arb**iter
RTT	**R**ound-**T**rip-**T**ime

SA	**S**witch **A**llocation
SCL	**S**hortest **C**ycle **L**atency
SoC	**S**ystem-**on**-**C**hip
SUR	**S**afe **U**nsafe **R**outing
TBFC	**T**ype **B**ased **F**low **C**ontrol
TDM	**T**ime **D**ivision **M**ultiplexing
VA	**V**irtual **A**llocation
VC	**V**irtual **C**hannel
VCT	**V**irtual **C**ut-**T**hrough
VN	**V**irtual **N**etwork
WH	**W**orm**H**ole

Abstract

The Network on Chip (NoC) has become the key element for an efficient communication between cores within the multiprocessor chip (CMP). The use of parallel applications in CMPs and the increase in the amount of memory needed by applications have pushed the network communication to gain importance. The NoC is in charge of transporting all the data needed by the processors cores. Moreover, the increase in the number of cores pushes the NoCs to be designed in a scalable way, but at the same time, without affecting network performance (latency and productivity). Thus, network-on-chip design becomes critical.

This thesis presents different proposals that attack the problem of improving the network performance in three different scenarios. The three scenarios in which our proposals are focused are: 1) NoCs with an adaptive routing algorithm, 2) scenarios with low memory access time needs, and 3) high-assurance NoCs. The first proposals focus on increasing network throughput with adaptive routing algorithms via the improvement of the network resources utilization, the first proposal SUR, and avoiding congestion spreading when an intense traffic to a single destination occurs, second proposal ECP. The third one and main contribution of this thesis focuses on the problem of reducing memory access latency. PROSA, through a hybrid circuit-packet switching architecture design, reduces the network latency by getting benefit of the memory access latency slack and to establishing circuits during that delay. In this way the information when arrives to the NoC is served without any delay. Finally, the proposal Token-Based TDM focuses on the scenario with high assurance networks on chips. In this type of NoCs the applications are divided into domains and the network must guarantee that there are no interferences between the different domains avoiding this way intrusion of possible malicious applications. Token-based TDM allows domain isolation with no design impact on NoC routers.

The results show how these proposals improve the performance of the network in each different scenario. The implementation and simulations of the proposals show the efficient use of network resources in CMPs with adaptive routing algorithms which leads to an increasement of the injected traffic supported by the network. In addition, using a filter to limit the adaptivity of the routing algorithm under congested situations prevents messages from spreading the congestion along the network. On the other hand, the results show that the combined use of circuit and packet switching reduces the memory access latency significantly, contributing to a significant reduction in application execution time. Finally, Token-Based TDM increases network performance of TDM networks due to its high flexibility and efficient arbitration. Moreover, Token-Based TDM does not require any modification in the network to support a different number of domains while improving latency and keeping a strong traffic isolation from different domains.

Resumen

La red en el chip (NoC) se han convertido en el elemento clave para la comunicación eficiente entre los núcleos dentro de los chip multiprocesador (CMP). Tanto el uso de aplicaciones paralelas en los CMPs como el incremento de la cantidad de memoria necesitada por las aplicaciones, ha impulsado que la red de comunicación gane una mayor importancia. La NoC es la encargada de transportar toda la información requerida por los núcleos. Además, el incremento en el número de núcleos en los CMPs impulsa las NoC a ser diseñadas de forma escalable, pero al mismo tiempo sin que esto afecte a las prestaciones de la red (latencia y productividad). Por tanto, el diseño de la red en el chip se convierte en crítico.

Esta tesis presenta diferentes propuestas que atacan el problema de la mejora de las prestaciones de la red en tres escenarios distintos. Los tres escenarios en los que se centran nuestras propuestas son: 1) NoCs que implementan un algoritmo de encaminamiento adaptativo, 2) escenarios con necesidad de tiempos de acceso a memoria bajos y 3) sistemas con previsión de seguridad a nivel de aplicación. Las primeras propuestas se centran en el aumento de la productividad en la red utilizando algoritmos de encaminamiento adaptativos mediante un mejor uso de los recursos de la red, primera propuesta SUR, y evitando que se ramifique la congestión cuando existe tráfico intenso hacia un único destinatario, segunda propuesta EPC. La tercera y principal contribución de esta tesis se centra la problemática de reducir el tiempo de acceso a memoria. PROSA, mediante un diseño híbrido de conmutación de paquete y conmuntación de circuito, permite reducir la latencia de la red aprovechando la latencia de acceso a memoria para establecer circuitos. De esta forma cuando la información llega a la NoC, esta es servida sin retardos. Por último, la propuesta Token Based TDM se centra en el escenario con redes de interconexión seguras. En este tipo de NoC las aplicaciones esta divididas en dominios y la red debe garantizar que no existen interferencias entre los diferentes dominios para evitar de este modo la intrusión de posibles aplicaciones maliciosas. Token-based TDM permite el aislamiento de los dominios sin tener impacto en el diseño de los conmutados de la NoC.

Los resultados obtenidos demuestran como estas propuestas han servido para mejorar las prestaciones de la red en los diferentes escenarios. La implementación y la simulación de las propuestas muestra como mediante el balanceado de la utilización de los recursos de la red, los CMPs con algoritmos de encaminamiento adaptativos son capaces de aumentar el tráfico soportado por la red. Además, el uso de un filtro para limitar el encaminamiento adaptativo en situaciones de congestión previene a los mensajes de la ramificación de la congestión a lo largo de la red. Por otra parte, los resultados demuestran que el uso combinado de la conmutación de paquete y conmutación de circuito reduce muy significativa de la latencia de red acceso a memoria, contribuyendo a una reducción significativa del tiempo de ejecución de la aplicación. Por último, Token-Based TDM incrementa las prestaciones de las redes TDM debido a su alta flexibilidad dado que no requiere ninguna modificación en la red para soportar una cantidad diferente de dominios mientras mejora la latencia de la red y mantiene un aislamiento perfecto entre los tráficos de las aplicaciones.

Resum

La xarxa en el xip (NoC) s'ha convertit en un element clau per a una comunicació eficient entre els diferents nuclis dins d'un xip multiprocessador (CMP). Tant la utilització d'aplicacions paral·leles en el CMP com l'increment de la quantitat de memòria necessitada per les aplicacions, hi ha produït que la xarxa de comunicació tinga una major importància. La NoC és l'encarregada de transportar tota la informació necessària pels nuclis. A més, l'increment del nombre de nuclis dins del CMP fa que la NoC haja de ser dissenyada d'una forma escalable, sense que afecte les prestacions de la xarxa (latència i productivitat). Per tant, el disseny de la xarxa en el xip es converteix crític.

Aquesta tesi presenta diferents propostes que ataquen el problema de la millora de les prestacions de la xarxa en tres escenaris distints. Els tres escenaris en els quals se centren les nostres propostes són: 1) NoCs que implementen un algoritme d'encaminament adaptatiu, 2) escenaris amb necessitat de temps baix d'accés a memòria i 3) sistemes amb previsió de seguretat en l'àmbit d'aplicació. Les primeres propostes se centren en l'augment de la productivitat en la xarxa utilitzant algoritmes d'encaminament adaptatiu mitjançant una millor utilització dels recursos de la xarxa, primera proposta SUR, i evitant que es ramifique la congestió quan existeix un trànsit intens cap a un únic destinatari, segona proposta EPC. La tercera i principal contribució d'aquesta tesi es basa en la problemàtica de reduir el temps d'accés a memòria. PROSA, mitjançant un disseny híbrid de commutació de paquet i commutació de circuit, redueix la latència de la xarxa aprofitant la latència d'accés a memòria i establint els circuits durant aquesta latència. D'aquesta forma la informació quan arriba a la NoC pot ser enviada sense cap retràs. Per últim, la proposta Token-based TDM se centra en l'escenari amb xarxes d'interconnexió d'alta seguretat. En aquest tipus de NoC les aplicacions estan dividides en dominis i la xarxa deu garantir que no existeixen interferències entre els diferents dominis per a evitar d'aquesta forma la intrusió de possibles aplicacions malicioses. Token-based TDM permet l'aïllament dels dominis sense tindre impacte en el disseny dels encaminadors de la NoC.

Els resultats demostren com aquestes propostes han servit per a millorar les prestacions de la xarxa en els diferents escenaris. La seua implementació i simulació demostra com mitjançant el balancejat de la utilització dels recursos de la xarxa, els CMP amb algoritmes d'encaminament adaptatiu són capaços d'augmentar el trànsit suportat per la xarxa. A més, l'ús d'un filtre per a limitar l'adaptabilitat de l'encaminament adaptatiu en situacions de congestió permet prevenir els missatges de la congestió al llarg de la xarxa. Per altra banda, els resultats demostren que l'ús combinat de la commutació de paquet i commutació de circuit redueix molt significativament de la latència d'accés a memòria, contribuint en una reducció significativa del temps d'execució de l'aplicació. Per últim, Token-based TDM incrementa les prestacions de les xarxes TDM debut a la seua alta flexibilitat donat que no requereix cap modificació en la xarxa per a suportar una quantitat diferent de dominis mentre millora la latència de la xarxa i mantén un aïllament perfecte entre els trànsits de les aplicacions.

Acknowledgements

Recorrí un largo camino, durante mucho tiempo ... Y durante este camino he estado acompañado de personas maravillosas que me han ayudado y apoyado para que este trabajo pueda ser hoy una realidad. Por eso quiero aprovechar este espacio para daros las gracias...

Quiero empezar dando las gracias a mis compañeros del GAP bueno.. Y también del malo. Gracias a todos por los momentos compartidos en el laboratorio, en comidas, viajes, experiencias, fiestas... Porque el trabajo con gente como vosotros es menos trabajo. En especial a Vicent, compañero infatigable durante estos cuatro años.

Gracias Davide, quien me acogió en mi estancia en Italia y con el que aprendí que no solo vale con quedarse en la superficie.

Quiero agradecer especialmente a Pepe, director de mi tesis y guía en este trabajo. Gracias Pepe por haberme ofrecido esta oportunidad en mi vida, por tu esfuerzo, por tu paciencia y tu ayuda en todo momento.

Gracias a Xema, Carles, Jorge, Sergi, Ocho, Davinia, Reme, María, Reme, Mari, Fabra, Daniela, Tirants, Juniors... Podría poner muchos nombres y nunca acabaría... Gracias por ser mis amigos y formar parte de mi vida... Sin vosotros nada sería lo mismo.

Gracias a mi familia y sobre todo a Miguel y Concha, mis padres, gracias por la educación que me habéis dado, gracias por darme la vida que todo niño querría tener, gracias por todo lo que habéis hecho por mí. Gracias por ser los mejores padres que podía tener.

Y sobre todo, gracias a ti, María, por haberme acompañado en este viaje y además haber decidido que quieres acompañarme en todos los que tenga (y el que nos viene en septiembre), gracias por tus ánimos en mis momentos de debilidad y desespero con la tesis, gracias por confiar en mí, por tu paciencia y todo el amor que me das, por ser mi luz. Sempre!

Y para terminar... GRACIAS a Salva, porque sé que me está cuidando día a día y parte de este trabajo es suyo...

GRACIAS!

Que la fuerza os acompañe

Chapter 1

Introduction

Nowadays, our society is immersed in a constant technological revolution with the adoption of new technologies and new ways of communication and interaction. In order to sustain this revolution, the computational power required increases dramatically every year. Therefore, the performance of new computing systems must increase to sustain the required computational power. As an example, new emerging multimedia applications in the personal computer or embedded systems landscape or the need of High Performance Computing (HPC) for solving challenging and complex problems pushes more computational power needs.

In order to meet these new performance levels, in the last years, computing architecture realm has suffered a radical change in its paradigm. Traditionally, performance improvements in microprocessors have been achieved by improvements over the architecture (multitasking, cache memories, etc...), but, also by taking benefit of system's clock speed, which has a direct effect on system performance. For instance, first 80286-based[1] processors accounted with system's clock frequency in the order of few MHz's. In contrast, in 15 years processors raised the frequency to the order of GHz's. Increasing the frequency is the straight approach to increase processor throughput. However, recently, clock frequency has reached its feasible limits. As seen in Equation 1.1, power roughly depends quadratically with the frequency, therefore, it is evident that above a given threshold dissipated power will reach unfeasible values. Several solutions have been used in order to increase the clock frequency as much as possible. As an example, processor cooling systems have evolved from simple aluminium heat sinks with no fan to modern liquid cooling systems. However, advanced cooling systems are expensive and consume huge amounts of power. Therefore, associated costs become unaffordable in production platforms such as HPC systems.

$$P = CV^2 f \tag{1.1}$$

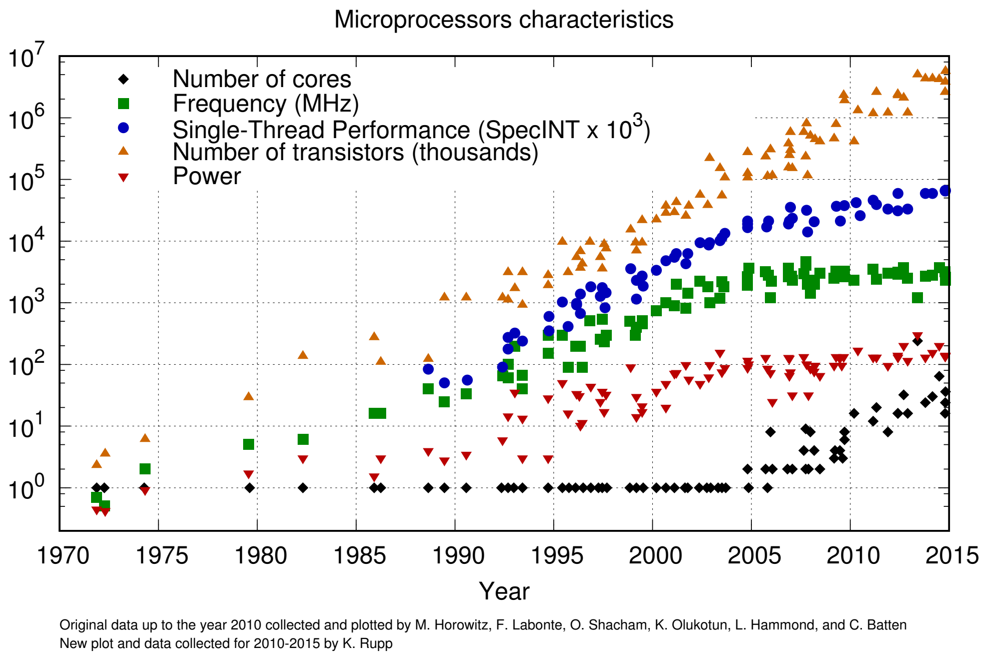


FIGURE 1.1: Microprocessors parameters evolution over time.

Because of the unfeasibility of speeding processors up by increasing the frequency, recently, chip manufacturers set the frequency increase strategy aside and opted to increase performance by means of implementing more cores in the same chip, which are termed chip multi-processors (CMPs). The idea behind this new strategy consists of, instead of relying on a big and complex monolithic processor running at high frequencies, to design simpler processors and physically replicate them several times while running at lower frequency values. In this way, applications can be mapped into different cores, hence, allowing them to run completely in parallel. This clearly implies a significant improvement in performance and power efficiency due to the benefits of parallelism. Due to the use of more power-efficient cores, a set of such cores can lead to same performance levels (or even higher) for the same power budget in monolithic processors.

In Figure 1.1 we can see the evolution of key microprocessor parameters since 1970. As seen, until 2005, manufacturers kept increasing clock frequency for single-core processors. In 2005, the clock frequency reached the top value, after which, the clock frequency has been kept roughly constant while the increased factor has been the number of cores, causing only a slight power consumption increment.

Currently, multi-core architectures range typically from 2 to 24 cores. However, in order to increase parallelism and take even more advantage of the multi-processor paradigm, the trend is to move forward in this approach by adding tens, hundreds or even thousands of cores. These processors are termed many-core processors.

The multi-core and many-core paradigm is not only restricted to CMPs. CMPs are typically composed of several general-purpose microprocessors interconnected in order to run any application in any of their nodes. This could be seen as the evolution of the general-purpose microprocessor. Similarly, Systems-on-Chip (SoC) consist in a complete and specific system integrating on the same chip most of the required components (cores, encoders, specific function modules, memories, . . .). As the integration scale continued its evolution, SoCs, similarly to CMPs, evolved to MPSoCs (Multi-Processor System-on-Chip), in which, regular SoCs implement several processors to take advantage of the multi-core approach.

Both CMPs and MPSoCs need an interconnect fabric in order to work. This on-chip network[2][3], termed network-on-chip or NoC, is necessary in order to support the internal traffic between components in the same chip. Protocol commands and memory blocks are forwarded through the network where some applications are executed on the CMP. These applications share the whole NoC resources. As some cores or applications can access to the same data at the same time a memory coherence protocol is needed to guarantee accesing data in a consistent way. Therefore, the NoC must be extremely fast and capable of serving data at very low latencies, otherwise the overall chip performance will be negatively affected. On the other hand, due to the intrinsic design restrictions inside the chip, the NoC must be carefully designed according to tight constraints in terms of area and power. In initial multi-core designs, since only a few nodes were typically implemented, such processors usually relied on simple buses or rings, as is the case of the IBM Power8 [4], shown in Figure 1.2. Bus and ring topologies are simple and relatively inexpensive. However, since all nodes connected to the network share the same physical media, they do not scale in performance with the number of interconnected nodes, since the network can be used by only one node at a time (for the bus case). Therefore, to make many-core processors feasible, other network designs must be used in order to allow concurrent communication between all nodes. In this sense, currently, point-to-point mesh network topologies are emerging as the most popular interconnect strategy in many-core systems. Indeed, as microprocessors are manufactured over a 2D silicon substrate, 2D meshes fit naturally well in the floorplan. An example of this network topology is the Tilera TILE-Gx72 platform[5] with 72 cores shown in Figure 1.3 which is provisioned with 5 completely independent 2D mesh networks each one intended to a specific type of traffic.

NoCs have been deeply researched for the last 15 years. These NoCs adopted many design styles and methods from networks designed for HPC systems. NoCs have been researched with two main goals in mind, network throughput and network latency. While network throughput is an important aspect of the NoC, the network latency becomes a key design point due to the low network load requirements inside the chip. Moreover, these NoCs are not exempt from well known problems such as network routing deadlock, failures, network congestion and security issues, to name a few. Also, the increment in

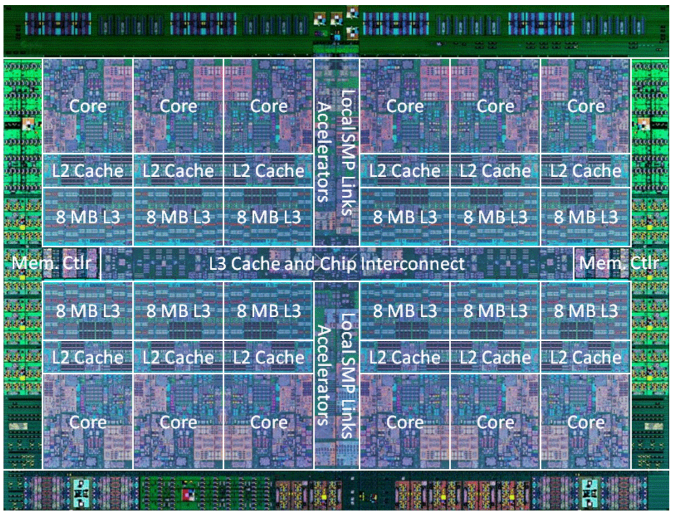


FIGURE 1.2: IBM Power 8 CMP chip layout.

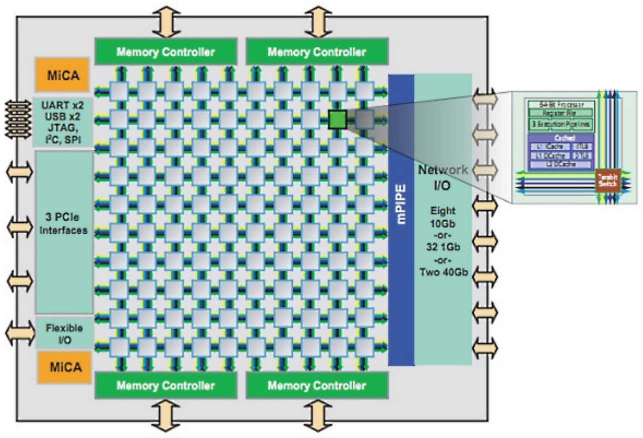


FIGURE 1.3: TILE-Gx72 platform provided with 100 tiles. 5 2D-meshes built in.

the number of units in CMPs and MPSoCs, makes the network utilization to increase, therefore, its design increases in size and complexity in order to allow better performance, leading to an increment of network latency.

In this thesis we address the improvement of NoC performance. There are different ways to address NoC optimization and they depend mainly on the context where the NoC is deployed and the application requirements set on the communication infrastructure. In this thesis we address different contexts thus providing different solutions for NoC performance improvement. Next, we describe each context and then we describe briefly each proposal.

Context 1: Adaptive Routing and Congestion

Routing in NoCs is quite similar to routing on any high-performance interconnection network. A routing algorithm determines the path that the message must take from source to destination. Routing algorithms are divided in two categories depending on where the routing function is applied. If the path along the network is determined before the packet is injected from the source node, then, the routing algorithm is termed source routing. However, if the routing algorithm chooses hop by hop the path that the packet must follow, then, it is termed distributed routing. Depending on the adaptability of the routing algorithm to the network status, routing algorithms can be further categorized into deterministic, partially adaptive and fully adaptive.

Fully adaptive routing algorithms achieve higher performance and lower network latency in highly loaded conditions. These algorithms are usually preferred to avoid or minimize congestion effects. Indeed, congestion is one of the most complex problems in interconnection networks. It occurs when network resources are oversubscribed and network bandwidth is lower than the requested one. As network size increases, this effect is more apparent and problematic. Under an scenario where congestion is severe, adaptive routing algorithms may even spread congestion, thus worsering the congestion problem.

The root cause of performance degradation in a congested situation is the Head-of-Line (HoL) blocking effect caused by congested packets to non-congested ones. Packets passing through congested spots block at the head of queues, keeping resources and impeding packets not passing through those spots from advancing. This phenomenon produces an increase in network latency, getting down network performance.

Although there are many techniques to solve the congestion problem (mainly by injection throttling or resources over provisioning), or the HoL blocking problem (resources over provisioning), they either require sophisticated implementations [6] or exhibit reaction times dependent of network size.

Context 2: Memory Access Latency

Switching techniques refer to the allocation of network resources at each hop when a packet travels through the network. Switching can be divided into Circuit Switching (CS) and Packet switching (PS). Packet Switching is a buffer switching technique where the packet is stored at every router. Virtual Cut-through (VCT) switching allocates the downstream buffer space and the packet is transmitted as soon as possible, not necessarily waiting the whole packet to be received. In Wormhole (WH) a similar approach is used, however, WH requires a buffer size lower than the packet size. In both approaches, the header flit carries the address information, which is used to set the path and the rest of the packet carries the data. However, wormhole switching imposes large performance penalty. This penalty occurs when a packet is blocked and blocks at several routers, then, it may introduce severe congestion problems increasing network latency.

Circuit Switching is a bufferless switching technique, where the links between the source and destination nodes are globally reserved to form a circuit. A request is propagated from source to destination, booking at each hop the links. When the request reaches the destination an acknowledgement (ACK) is sent back to the source. Once the ACK message arrives to the source, the packet or stream of packets are sent through the circuit. Later, an *END-OF-CIRCUIT* message is sent to tear down the circuit. Long messages are good candidates to configure circuits, as they will amortize the time spent in setting up the circuit. However, in CMPs where the messages are not large, the network performance is affected due to the setup delay. Some more aggressive proposals implement a multi-hop circuit switching allowing the network to forward the packet more than one hop each cycle. Following this approach, SMART [7] proposes an architecture where a message can cross the whole network in a single cycle.

One critical aspect of CMP systems is memory access latency. Data needs to travel from memories to the cores (caches mainly) via the NoC. If we could smartly set circuits between memories and caches then the performance of applications would be boosted.

Context 3: High-Assurance NoC

Network performance (both latency and throughput) can be severely impacted when security property has to be enforced. An example of security break occurs when a malicious application injects messages at a high rate, flooding the network, and thus, performing a denial of service attack on the memory controller, causing the rest of applications not to continue with their execution, as they cannot access the memory controller. On the other hand, side channel attacks are one of the most dangerous treats that target hardware components and specially the NoC can suffer. Usually, timing leakage is used by the attackers on the cache memory to monitorize the network latency and indirectly obtaining information about the other applications (Meldown). In order to guarantee security, Time Division Multiplexing (TDM) is widely used. In TDM the network usage is divided in time slots and each slot is assigned to an application. The network only transports messages belonging to a slot at a time, thus, avoiding traffic collisions between slots (applications). By doing this, application flows are isolated. However, this approach harms both latency and throughput of the network.

1.1 Thesis contributions

This thesis focuses on improving NoC performance for the three context scenarios presented above. Different strategies are proposed, presented and evaluated. The different contributions in this thesis address the network performance improvement from different requirements perspective and with different systems, representing the three previous contexts. In particular, we address the two standard problems of performance improvements both in terms of network throughput (under congested scenarios) and network latency. Indeed, one central contribution of the thesis is the achievement of network latency reduction by combining both types of switching strategies, circuit and packet switching combining it with the SMART technology. We also address the achievement of a high-assurance NoC for perfect application flow isolation guarantees.

Figure 1.4 shows the different contributions of the thesis. The first contribution is a novel flow control strategy, referred to as Type-Based Flow-Control (TBFC). This mechanism is tailored to buffer resources with minimum capacity but still allowing virtual cut-through switching (thus enabling its benefits). In addition, TBFC is prepared for a new type of routing algorithms which, depending on the type of a packet may take different routing decisions. Indeed, we apply a novel adaptive routing algorithm on top of TBFC. The algorithm, referred to as Safe/Unsafe routing (SUR), classifies packets as safe of unsafe depending on the chances of packets to induce deadlock. Safe packets move through the network in an unrestricted manner, while unsafe packets are routed only through deadlock-free paths. When both methods are combined, TBFC and SUR, the performance results show a boost in performance when the algorithm is used in 2D torus networks. Also, performance is kept maximum in 2D mesh configurations while using less resources. This proposal does not improve the network latency compared with previous fully adaptive routing algorithm, but is able to reach a higher injection rate before reaching the saturation point.

The second proposal addresses the problem of high network latency in congested situations. We propose a congestion filter to be used together with the adaptive routing algorithm. The filter, referred to as End-Point Congestion Filter (EPC), prevents congested packets from spreading through the network. The filter disables temporarily adaptivity for those packets that participate in a congestion situation. By doing this, congestion is prevented from spreading and taking much network resources, thus allowing the rest of non-congested packets to adapt and avoid congested ones. This proposal shows a total decoupling of congestion from adaptive routing, thus guaranteeing no interference on network and system performance.

The main contribution of this thesis is the one listed as the third contribution. We propose PROSA, a PROtocol-oriented circuit Switch Architecture, which co-designs both the NoC and the coherence protocol and put them to play together. Our approach enhances the NoC with a new clustered component, the PROSA controller (PC), which is in charge of managing circuits and to resolve any possible conflict. The controller is in charge of four NoC PROSA routers (PR) and steers their local circuits when needed. The proposal sets circuits only for long multi-flit protocol messages. Circuits for those messages are configured before they are indeed injected, taking advantage of the time slack obtained with the cache access and for the processing delay incurred in the network interface (NI). Thus, hiding the circuit setup time.

The fourth proposal of the thesis is PROSA-DD. This proposal enhances the previous work. In that case PROSA-DD tries to setup circuits for all messages, either short or long taking as a premise that even with a small delay at injection (because we need

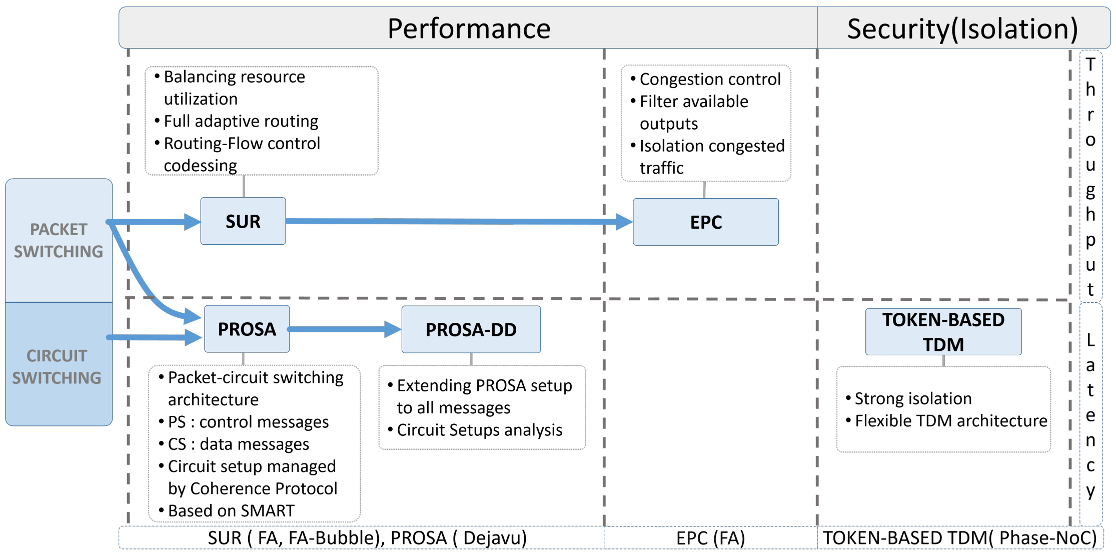

Performance
Security(Isolation)
PACKET SWITCHING
CIRCUIT SWITCHING
• Balancing resource utilization
• Full adaptive routing
• Routing-Flow control codessing
SUR
• Congestion control
• Filter available outputs
• Isolation congested traffic
EPC
PROSA
PROSA-DD
• Packet-circuit switching architecture
• PS : control messages
• CS : data messages
• Circuit setup managed by Coherence Protocol
• Based on SMART
• Extending PROSA setup to all messages
• Circuit Setups analysis
TOKEN-BASED TDM
• Strong isolation
• Flexible TDM architecture
Throughput
Latency
SUR (FA, FA-Bubble), PROSA (Dejavu)
EPC (FA)
TOKEN-BASED TDM(Phase-NoC)

Figure 1.4: Thesis outline.

to setup the circuit) the transmission latency of the message will still be smaller than the transmission time in packet switching (PS) mode (as we use SMART technique). Another alternative relies on message distance to destinations, using circuits only for messages with closer destinations, thus circuit setup time is shorter. Finally, few additional cycles (slack) can be provided to the circuit setup process, increasing the success rate of circuits established and used by the application. All these alternatives are explored in PROSA-DD.

The last contribution of this thesis is Token Based-TDM. It put the focus on security problems. Token based-TDM creates a TDM network based on the Channel Dependecy Graph (CDG) for an efficient time slot propagation through the network. The arbitration domain is performed at each router instead of being performed at network level; as is the case of standard TDM network. Tokens are used to set the arbitration domain at each router and these tokens are propagated following the CDG along the network. This new approach enables the network to achieve low latency while ensuring strong isolation. Moreover, our contribution is able to support different number of domains without any change in the network, being highly flexible. State-of-the-art contributions propose static solutions working for a hardwired number of isolated domains and predefined network-on-chip characteristics.

To summarize, our contributions in this thesis are the following ones:

- TBFC+SUR: co-design of flow control and routing algorithm to achieve a balanced buffer utilization for fully adaptive routing algorithms.
- EPC: A congestion control mechanism that avoids spreading of congestion by isolating the congested flows when using fully adaptive routing.

- PROSA: a new circuit-packet switching NoC architecture driven by the coherence protocol. PROSA reduces the latency of data messages using the coherence protocol knowledge to setup the circuits before they are needed and only for the time period they are required.
- PROSA-DD: PROSA extensions to setup circuits for both type of messages, control and data, including new functionalities such as additional slack for setting up circuits and distance-driven circuit setup strategy.
- Token-Based TDM: a strong isolation network based on the CDG to improve the flexibility of TDM designs. It supports different number of domains without any change in the NoC design.

1.2 Thesis Outline

Following the rules of *Universitat Politècnica de València*, this thesis has been written as a compendium of articles and is structured as follows:

- In Chapter 2 the background of this thesis is described as well as related work. Although subsequent chapters will include related work sections, we provide in this chapter a complete and integrated background and related work description in order to provide a unified view to the reader.
- In Chapter 3 we put together all the descriptions of this thesis contributions, together with their evaluations and an assessment of their similarities, differences and complementaries. The goal of this chapter is to ease the understanding of the contributions and to let the reader be focused on them.
- From Chapter 4 to Chapter 8 reflects the compendium of publications arranged as follows:
  - In Chapters 4 we describe the first NoC architecture proposed, TBFC+SUR.
  - In Chapters 5 we propose EPC to avoid spreading the congestion.
  - In Chapters 6 and 7 we propose PROSA and PROSA-DD, a new NoC architecture driven by the coherence protocol.
  - In Chapters 8 we propose a high assurance NoC.
- Finally, in Chapter 9 we expose the conclusions and enumerate all conferences in which the articles of this thesis have been published in.

Chapter 2

Background and Related Work

In this chapter we collect basic concepts required to fully understand all the contributions described in this thesis. First, we describe basic concepts and terminology related to networks on chip. Then, in each specific topic we provide the state-of-the-art.

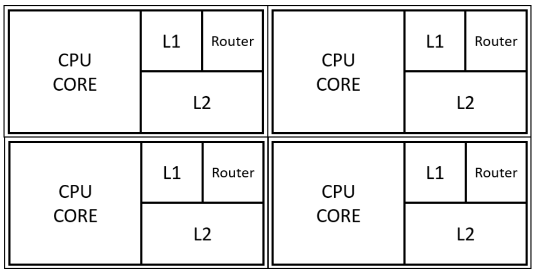
2.1 Network on Chip (NoC)

The Network on Chip (NoC) concept is the result of several design choices such as network topology, switching and flow control techniques or routing algorithms. The network topology defines the physical interconnection between the different elements inside the chip. Switching and flow control define how the information is transmitted through the network and the routing algorithm chooses the path to communicate end nodes through the network.

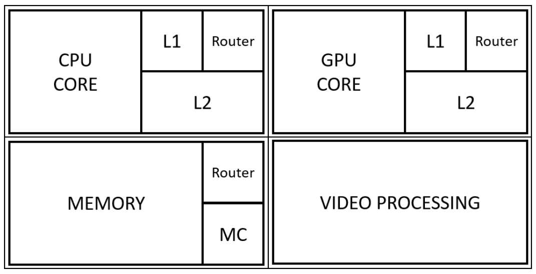
The elements that the NoC is composed of are nodes, routers, and links. The nodes are typically the compute elements and they communicate with other nodes through the network using the network interface. Nodes may include caches, which are smaller, faster memories. These caches store copies of the recently used data. Main memory is not accessed directly. Instead, a memory controller is accessed through the NoC to reach main memory. Routers provide the connectivity between end nodes by connecting to other routers and to the end nodes. The links are the physical connection that connect all the elements in the NoC (nodes, routers, and memories). In this section we describe briefly these concepts. Deeper details can be found in [3].

2.1.1 Topology

Before NoCs were conceived, communication within a chip were performed with busses, as is the case of the Cell processor [8], were all the elements access to the same bus

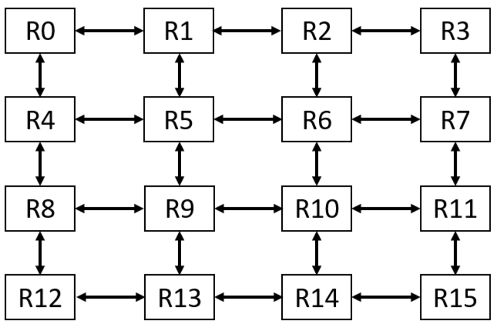


(A) Tiled CMP.

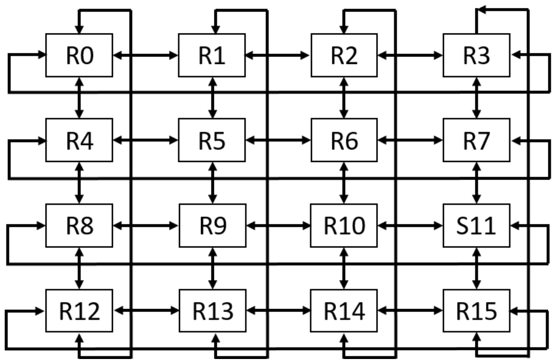


(B) Heterogeneous CMP.

FIGURE 2.1: Different CMP designs.



(A) Mesh topology.



(B) Torus Topology.

FIGURE 2.2: Different CMP desings.

to communicate. However, this approach is only suitable for systems with a small number of nodes. Indeed, nowadays industry integrates many simpler cores on the same chip. Buses in these systems make nonsense as they offer a poor scalability and limited bandwidth.

The NoC concept emerged as the solution to achieve effective on-chip bandwidth. Current multiprocessor systems are typically composed of a collection of tiles. A tiled chip multiprocessor (CMPs) design is shown in 2.1a. The node, cache memories, and routers are the elements that compose the tile. Every tile is connected to a subset of tiles through an on-chip network. There is a major difference between homogeneous (inducing regular topologies) and heterogeneous designs (more suited to irregular topologies). Instead, high-end multiprocessor systems-on-chip (MPSoCs) are an example of heterogeneous designs, Figure 2.1b shows an example, where tiles are different in many aspects: size, functionality, performance, throughput, etc.

CMPs usually derive in orthogonal topologies. This type of design allocates nodes in regular patterns, making easier and regular the tiles connection. Orthogonal topologies allocate nodes in an n-dimensional array with k nodes along each dimension. Every router has at least one link connecting to a neighbour router for each direction and every router is labelled with an identifier depending on its coordinates. The communication links between a pair of routers are bidirectional, having one channel in each direction.

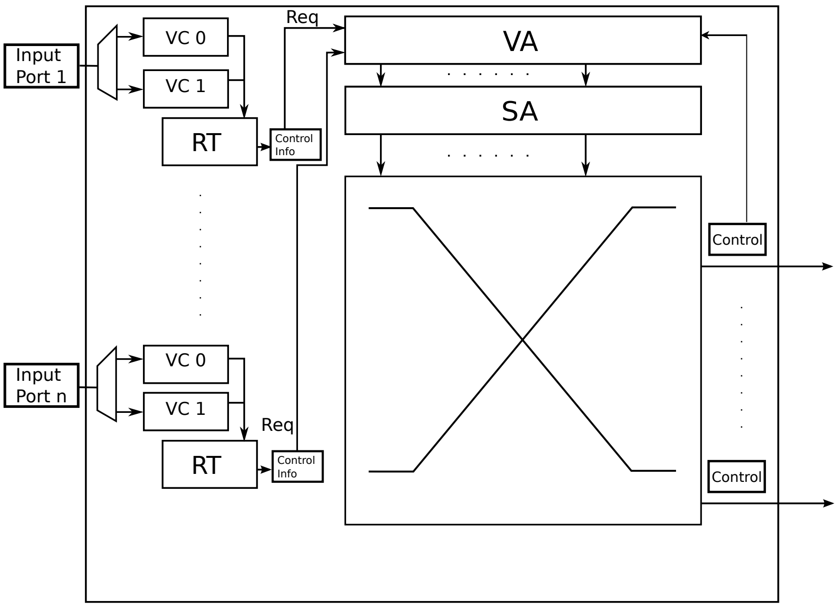


FIGURE 2.3: Cannonical architecture of the router.

The most popular design in NoC architectures is the n-dimensional mesh, shown in Figure 2.2a, used in most of the commercial and non-commercial NoCs designs. The 2-dimensional mesh is the most widely studied topology as it fits with the chip surface layout. However, some proposals assume 2-dimensional torus, Figure 2.2b. The torus floorplan is similar to the one assumed for the mesh topology, but connecting the nodes at the boundaries of every row and column. These long links are denominated wraparound links. We assume 2-dimensional mesh and torus topologies in our proposals.

2.1.2 Router

Routers, or switches, are the building block of the network. They forward messages from their inputs to their outputs. Nodes are attached via a link, and routers are connected to other routers. Routers are pipelined and have typically the following modules: buffers, routing unit, arbiter unit, and crossbar.

When a message arrives at an input port, it is stored in a buffer. Buffers are the most important element inside the router. Buffers are typically associated to one channel, also called port (input or output). The buffers associated to an input port receive the messages and store them waiting for routing decisions. The buffers occupy a large part of the router area and they are the main power consumers inside the NoC. Notice that, to save area and power, usually buffers at output ports are not implemented.

The routing unit is the next step inside the router. This unit is in charge of deciding the output port a message must take. Routing unit feeds the arbiter unit. This unit

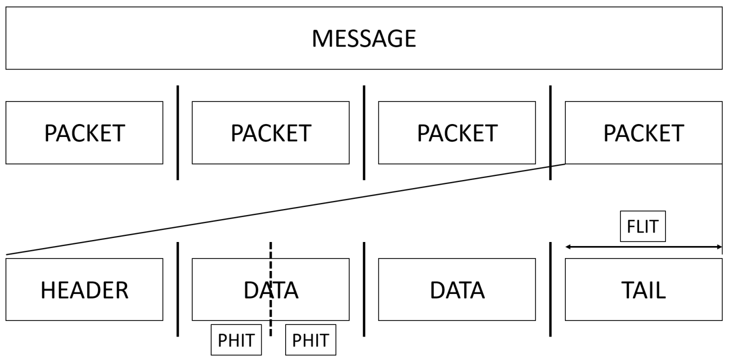


Figure 2.4: Data Units.

arbitrates between the different requests to allow the input ports to access the output ports.

The crossbar is a non-blocking switching element. The crossbar allows connecting all the input ports with all the output ports. These connections are set depending on the arbiter unit decisions.

2.1.3 Data Unit

The Data Unit, or message, is the information that one node wants sends to another node. The message is a collection of bits that the sender transmits to the destination. Depending on the switching strategy the message can be divided into smaller units, called packets. A packet is divided further into flits (flow control digits), which are the smallest unit of information flow controlled by routers. As the width of the link can be lower than the size of a flit, the flit is further divided at the physical level, into phits (physical digits). It is left to the designer and the parameters involved, to determine the size of every unit. However, in on-chip networks, due to the vast amount of bandwidth available, the phit size usually equals to the flit size. Message splitting into packets and reassembly of these packets back to a message are performed at the network interface. Commonly, the packet or message is composed of the header flit, body flits and tail flit. The header flit contains the information for routing and control. Body flits contain data and the tail flit determines the end of the message or packet. Often, packet and message terms are interchangeable by the community, when both are equal in size.

2.1.4 Switching Technique

The switching techniques determine how messages or packets advance through the routers. Indeed, switching sets connections between the input and the output ports. Depending the switching technique implemented the router may have a significant impact on performance, area and power consumption.

In Circuit Switching (CS) the routers establish a reserved path between the source and destination nodes before sending data. A *circuit request* message is injected into the network. It contains the destination information and reserves the required channels for message transmission at each hop. When the request reaches the destination an acknowledgement is sent back to the source. Once the acknowledgement arrives to the source, the message or multiple messages are sent to the destination not using any buffer and without any conflict with other flows inside the network. The tail flit releases the booked links allowing other flows to get the resource. The effectiveness of the CS approach relies on the type of traffic that will be using circuits. This switching technique is a perfect choice for long messages or very frequent messages (bursty traffic). Nevertheless, if the circuit setup time is longer to the transmission time of data, then, the network performance will be strongly penalized.

Packet Switching (PS) is a buffer switching technique where the packet is stored at every router, instead of reserve the whole path at a time. The three main packet switching alternatives in a NoC are store and forward (SAF), Virtual Cut-Through (VCT) and Wormhole (WH).

Store and forward (SAF) is the easiest packet switching technique. When a packet arrives to a router, it is fully stored at the input port. Once is completely received, the message can be forwarded to the next router or node. Notice, that with SAF the buffer size should be at least equal to message size. Moreover, the packet latency is multiplicative with the number of hops along the path.

Another switching technique with the same buffer resource requirement is Virtual Cut-Through (VCT). However, with this technique, the message can be forwarded when the header flit arrives at the input port buffer without having to wait all flits are received. This approach improves the network latency, in that case, the base latency is mostly additive to the distance between the source and destination nodes. In fact, in high load traffic conditions, VCT behaves as SAF. VCT is commonly used in off-chip high-performance networks, where the buffer size is not a critical design factor.

Both previous switching techniques require a high amount of buffer resource. However, in Wormhole (WH) switching buffers at the ports have to provide enough space to store a few flits instead of the whole packet. For efficiency reasons, the buffer size depends on the round-trip-time delay (RTT). RTT is the delay period between a flit is forwarded and the acknowledgement of the transmission is received. As it happens in VCT, the message is forwarded as soon as possible, without waiting to receive more flits. However, the buffer can not store the entire message, so, a blocked message will be stored in several routers along the path. This effect is the most important drawback because it could lead to a high contention effect inside the network, increasing the network latency and reducing network throughput.

To solve this effect, virtual channels are proposed in (ref VC). VCs divide the buffer into a set of virtual buffers and the port and the channel are shared between these virtual buffers. This technique requires a multiplexing and also has to take into account for switching techniques and flow control. Virtual channels offer an improvement in latency and performance to the network. The weakest point of this approach is that when multiple VCs are mapped on one physical channel, then, the link bandwidth is shared between all the VCs. VCs are not restricted to WH switching, they are also used for example to support adaptive routing algorithms and quality of service.

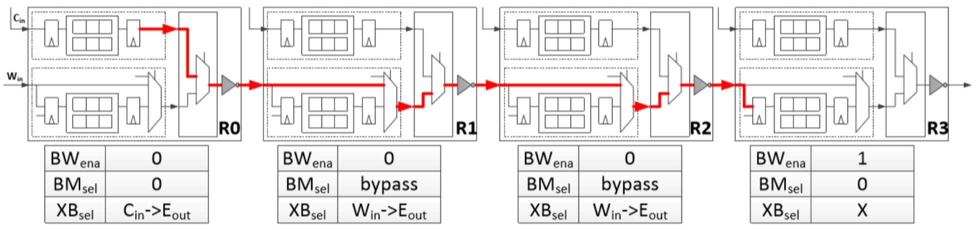
Related Work Linked with Switching

As a summary, circuit-switching has been used in NoC architectures in order to reduce on-chip communication latency. Once a circuit is set, data does not travel through the routing and arbitration stages on each router. However, setup time usually causes low resource utilization and performance degradation. On the other hand, packet switching improves resource utilization and network performance, splitting the entire message into smaller blocks and forwarding them along the network.

Some works try to get benefit from both mechanisms by implementing a hybrid circuit-packet switching strategy. Kumar [9] proposes Express Virtual Channels (EVC) allowing packets to bypass intermediate routers along their path. EVCs only allow to connect nodes along the same dimension, so circuits cannot turn from one dimension to another.

Jerger [10] proposes circuit switched coherence, setting permanent circuits between pairs of frequent data sharers instead of tearing them down. It allows to quickly send data between the same nodes. However, if another circuit requires the resource, the data is switched to packet switching until it reaches destination. Yin [11] proposes a hybrid circuit-packet switched network in which the packet is forwarded along the packet network while the circuits can be set in parallel, using TDM. Yim's proposal also spends time in the setup latency. Mazloumi [12] proposes another hybrid packet-circuit switched router. This mechanism setups the circuit along the network while the request message is being forwarded between the requestor and the destination. When the request reaches the destination and the data is ready, the mechanism sends a probe message activating the reserved circuit, after that the data is sent. Chen [13] proposes an implementation of a hybrid circuit-packet switching. Ma, et al. in [27] proposes a hybrid wormhole/VCT switching technique to reduce buffering while improving the performance of fully adaptive routing. All these mechanisms require a setup period.

Van Lear [16] proposes a coherence-based message predictor for optical interconnection networks. In the proposal a global predictor establishes the circuits between nodes. All the traffic in the network has to cross the predictor, thus potentially causing a bottleneck in the network. This proposal is also for optical interconnects where a full optical crossbar is assumed. This makes scalability a major issue. Shacham [17] proposes



BW_{ena}	0
BM_{sel}	0
XB_{sel}	C_{in}->E_{out}

BW_{ena}	0
BM_{sel}	bypass
XB_{sel}	W_{in}->E_{out}

BW_{ena}	0
BM_{sel}	bypass
XB_{sel}	W_{in}->E_{out}

BW_{ena}	1
BM_{sel}	0
XB_{sel}	X

FIGURE 2.5: Single-cycle Multi-hop Asynchronous Repeated Traversal Example

an hybrid optical-electrical network. In this proposal the electrical network is used to establish the optical circuit. Peh [18] presents flit-reservation flow control. In this proposal the circuit is setup hop by hop.

Liu et al. [19] propose an effective setup circuit procedure, in which the setup procedure is guaranteed to terminate in 3D+6 cycles, where D is the distance. Xue et al. [20] propose a general mathematical framework for reconfigurable networks in order to optimize the network by configuring subnetworks to transmit data. Hollis et al. [21] propose a reconfigurable NoC using the Skip-link. This proposal configures the Skip-link to allow packets to bypass the channel, avoiding to cross the router.

A cluster approach has been proposed previously by some authors. Xue et al. [22] propose a cluster approach to send multicast messages, encoding these messages and decoding them at destination. They support dropping parts of the message. Qian et al. [23] propose a cluster approach by connecting clusters through Express Virtual Channels (EVC) and a Hub router. They use an adaptive routing algorithm to choose between regular network, the EVC or the Hub router. Both approaches implement a packet switching approach.

Abousamra [14] proposes Deja Vu. This proposal pre-allocates the circuit between nodes in order to hide the setup latency by dividing the NoC in two planes: control and data plane. The control plane is in charge of configuring the circuits. This plane has higher voltage and frequency, so being faster. In this NoC, the request packet pre-allocates the path in backward direction as it approaches destination. The destination node can forward the response to the requestor whenever data is ready with no circuit setup. This approach can produce conflicts. Deja Vu configures the circuits in the order they are reserved. Then, the selected order schema can produce underutilization of network resources. In [15] authors alleviate the problem by using a different order. However, it still requires the high frequency and voltage control plane. Two of our contributions, PROSA and PROSA-DD are compared with the current Deja Vu.

Krishna in [7] presents SMART (Single-cycle Multi-hop Asynchronous Repeated Traversal). This is a proposal more aggressive to reduce latency network. Smart proposes a multihop network with single-cycle data-path for all the communications between source to destination. So, any communication can be performed in the best case in one cycle

Smart implements an extra network to setup the circuits. Figure 2.5 shows an example of how SMART implements a multihop network. SMART try to setup the circuit one cycle before the data is sent and partial circuits can be established. SMART do not add any additional fast physical express links in the data-path; instead it drive the shared crossbars and links asynchronously up to multiple-hops within a single cycle. Our contribution, PROSA is based on this technology forwarding data flits in a single cycle from the memory controller to L2 cache memories and between L2 and L1 cache memories.

2.1.5 Flow Control

Flow control dictates how traffic flows advance between routers. Buffers are a finite resource, so they can not store more data than their capacity. The NoC needs a flow control technique to avoid buffer overruns. The flow control mechanism dictates when a flit, message, or packet can be sent, guaranteeing always that the flit, message, or packet will be stored at the receiving router.

There are three flow control techniques that are frequently used: *ACK/NACK*, stop&go, and credits. The *ACK/NACK* flow control mechanism is the simplest one. When a router sends a flit it has to wait until the *ACK* message is received keeping the flit. If a *NACK* is received, then, the flit is retransmitted again. When a flit arrives at the input port, if the buffer has space available, then the flit is stored in the buffer and an *ACK* message is sent back. Otherwise, the buffer is full and a *NACK* message is sent back and the incoming message is dropped.

ACK/NACK requires a high amount of control traffic. Contrary to this, Stop&Go reduces the control traffic between routers. Stop&Go is a flow control mechanism based on two thresholds set in the receiving queue. Thresholds are set based on the round-trip time of the link. When the buffer is getting full and the occupied space of the buffer reaches the stop threshold, then a stop signal is sent back to the sender. As the mechanism takes into account the round-trip time, all the flits sent by the sender before the stop signal is received can be stored in the input port buffer. When the buffer occupation goes below the go threshold, a go signal is sent back to the sender, in order to resume the injection of flits.

Credit-based flow control is based on the knowledge of the sender of the free space at the receiving buffer. Every output port keeps a count of credits, which is equal to the number of flits that can be stored at the input port connected to the output port. When a router transmits a flit for one output port, then the count associated with this output port is decreased by one. This counter with a zero value means that there is no available space at the input port, thus the router can not forward more flits along this output port. On the other hand, when a flit is forwarded and frees the allocated space at the receiving buffer, a credit signal is sent back to the previous router to increment the credit counter associated with this port.

Related Work Linked with Flow control

These three are the main flow control mechanisms used. However, there are other proposals that improve performance. Tang in [24] proposes a flow control in which they limit the injection rate dynamically in the network. This flow control strategy can be only used in meshes. Nousias in [25] proposes an adaptive rate control strategy in wormhole switching with virtual channels. When the contention changes, the destination node sends a signal to the source node to regulate the injection rate accordingly. Avasare in [26] proposes a centralized end-to-end flow control for packet switching. This flow control requires two networks, the control network, and the transmission data network. Ma, et al. [28] proposes a flit bubble flow control scheme by refining the baseline bubble flow control scheme. Chen, et al. [29] proposes worm bubble flow control (WBFC), which reduces the buffer requirements and improves buffer utilization in torus networks. However, the methods in [27–29] still need to separate the virtual channels into adaptive and escape channels.

2.1.6 Routing Algorithm

Network topology defines the physical organization of the network and the possible paths between the elements. The routing algorithm, instead, computes which is the path that the message takes to reach its destination. Routing algorithms must be carefully designed in order to avoid collateral issues such as deadlock, livelock, and starvation.

Deadlock occurs when a set of messages can not advance because they request buffers that are occupied by those messages and a cyclic dependency appears between all the messages. There are two common ways to deal with deadlock events. The first one is implementing deadlock-free routing algorithms, usually based on designing an acyclic Channel Dependency Graph (CDG) and the second one providing a recovery methodology to escape from the deadlock situation.

In a livelock situation, a message keeps moving but never reaches its destination. In this case, the message is misrouted and never reaches its destination because the required links are always reserved to other messages. Livelock arises when non-minimal paths are allowed. Livelock can be easily avoided by limiting the number of misrouted hops performed by a message.

The starvation issue occurs when a message is permanently blocked holding a resource and cannot advance because the network traffic is high and the resource requested is granted to other messages with higher priority. This issue can be avoided by a proper arbitration design with a correct priority mechanism.

Related to routing algorithms support, there are two implementation trends, table-based, and logic-based. The first approach, table-based is based on row-like structures that match destinations with the table entries. So, given a destination, the row of the table associated with the destination is accessed, in order to know the routing output. Usually, these tables are implemented with memory structures. On the other hand, in logic-based routing, the algorithm is implemented as a set of logic gates. Then, when a header flit arrives at the input port and it is decoded, the output port is computed based on the logic function. Table-based routing algorithms are flexible in the sense that many algorithms can be encoded in the tables. However, logic-based routing algorithms get more efficient implementation in terms of delay, area, and power consumption.

Routing algorithms can be classified by several key factors. One possible classification, as said above, is based on where the routing function is implemented. In source-based routing algorithms the complete path to follow is encoded in the packet's header at the source end node. Alternatively, in distributed-based routing the path is computed at each router. Another possible classification takes into account the number of receiver end nodes of messages. Following this approach, the routing algorithm can be classified as unicast, multicast, and broadcast. In unicast communication, packets only have a single destination. In some situations, however, a message must be sent to several destinations. In this case, the routing algorithm must support collective communications (multicast and broadcast). If one message is sent from a source to the rest of the nodes in the chip, the routing operation is termed broadcast communication. However, if the message is sent to a group of end nodes, the routing operation is termed multicast communication.

Depending on the adaptability of the routing algorithm, it can be further categorized into a deterministic, partially adaptive and fully adaptive. In deterministic routing, the packets from one source to one destination always take the same path. On the other hand, fully adaptive routing algorithms allow the packet to choose between all the output ports to bring the packet closer to its destination. Fully adaptive routing algorithms should have lower latency network, but the routing functions are more complex than deterministic ones. Partially adaptive routing algorithms try to combine the advantages of the two mentioned above. These algorithms provide limited adaptability for packets.

Related Work Linked with Routing Algorithms

A large set of works related with routing algorithms have been proposed. Here we briefly comment on some basic references mainly proposing adaptive routing algorithms (since one of our proposals addresses adaptive routing algorithms). It is not our intention to cover most of the related work spectrum of routing algorithms. Dally and Aoki [30] described the dynamic miss routing algorithm by tagging packets based on how many misroutes they have incurred and allow any packet to request any VC as long as it's not waiting for a packet with a lower dimensional reversal number.

Glass and Ni in [31] proposed turn model for designing partially adaptive deadlock-free algorithms in a mesh. The west-first routing algorithm in a 2D mesh traverses the west hops first, if necessary, and then adaptively south, north and east. The negative-first routing (NFR) algorithm in a 2D mesh routes a packet first adaptively west and south, and then adaptively east and north. Chiu [32] proposed the improved partially adaptive routing algorithm odd-even turn model by constraining turns, that can introduce deadlocks, to occur in the same row or column. Wu [33] proposed a fault-tolerant odd-even turn model-based routing algorithm for 2D meshes.

Dally and Seitz in [30] presented the sufficient and necessary condition for deadlock-free routing in an interconnection network. Several routing algorithms were proposed for meshes and tori [32, 34, 35]. Load-balanced, non-minimal adaptive routing algorithms for tori were proposed by Singh, *et al.* [35, 36] with three virtual channels. The method in [37] presented an adaptive minimal deadlock-free routing algorithm for 2D tori. However, the number of virtual channels required by the method was not well-controlled in [37].

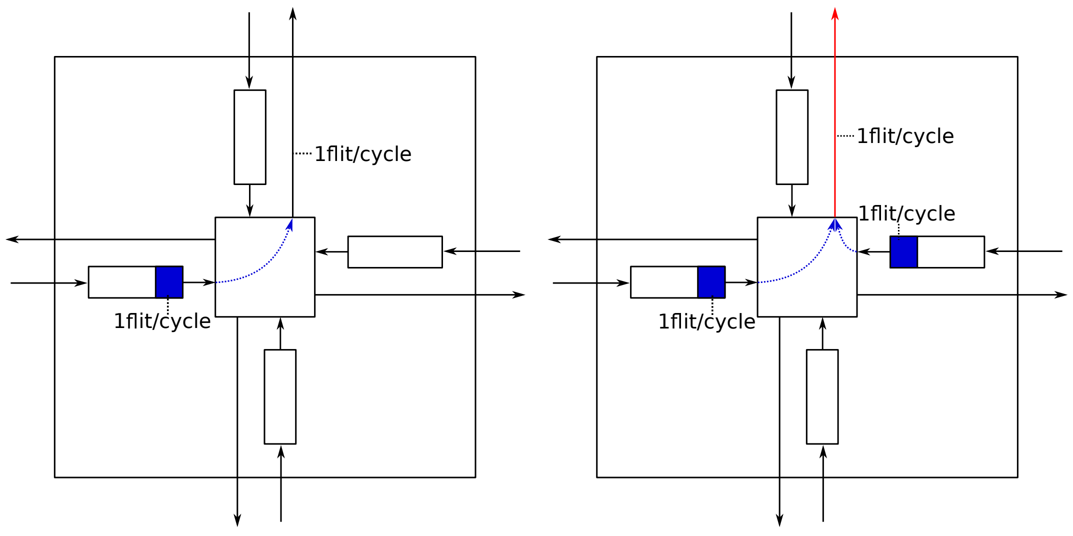
Duato [38] proposed a necessary and sufficient condition for deadlock-free adaptive routing in WH-switched networks. Methodologies for a design of deadlock-free adaptive routing algorithms are also presented in [39]. The adaptive bubble router [40] for the VCT-switched torus is based on Duato's protocol. It requires an escape channel with dimension-order routing (DOR) and an adaptive channel. A flow control function is added to the escape channel in order to avoid deadlocks.

In NoCs, Marculescu in [41] proposed a new routing technique (DyDA) which switches between deterministic and adaptive routing based on the network's congestion conditions. When the network is not congested DyDA router works with deterministic routing. When the network becomes congested, then DyDA router works with adaptive routing. Ebrahimi in [42] proposed a new fully routing algorithm (DyXYZ) for 3D NoCs. In this new routing, the congestion information is used as a congestion metric to select the best output port.

Mejia [43] proposes Segment-based routing algorithm. This proposal divides the topology into subnets, and subnets into segments. Then, a bidirectional turn restriction is placed locally within a segment. As segments are independent, the restrictions can be placed inside the segment independently where the turn restrictions are allocated on the other segments. This technique reaches a high flexibility compared with previous proposals.

2.1.7 Congestion

A highly loaded NoC can easily become congested, even if this situation occurs during a limited period of time. To define congestion, first we need to define contention effects.



(A) One flow forwarded at high data rate (B) Two flows causing contention

FIGURE 2.6: Contention example.

Contention is defined as the situation when two packets arrive to the same router (possibly through different input ports) and they request the same output port, as shown in Figure 2.6b. In that situation only one packet will win the access to the output port and will advance. The other packet will block temporarily. This is a small contention effect and is due to the excess input bandwidth requesting the same output port.

Contention can occur without degrading significantly system's performance. If contention occurs sporadically, the router buffer absorbs accumulated traffic in a lesser or greater extent, depending on the router buffer size, softening contention effects, thereby causing negligible network turbulence. However, when the contention occurs for moderate or large period of time, buffers will quickly fill. This causes flow control to trigger the stop signal to upstream routers, stopping the incoming flows. The stop signals cause those router buffers to be filled as well, therefore propagating this effect to the rest of routers. When this occurs the contention is spread over the network starting from the first contended router and creating branches due to the interaction of other data flows with the congested flow. When this happens, the network becomes congested.

Congestion is defined as the effect of suffering contention along the time, moreover, when a high congestion appears on the NoC, the NoC performance decreases due to the network saturation.

Related Work Linked with Congestion

Congestion can be addressed in different ways. First, congestion avoidance techniques guarantee congestion never builds in the network (e.g. ATM networks [44]). However, this leads to low network throughput and utilization. The second approach is to detect congestion, notifying the sources, and removing congestion by injection throttling [45].

Although it is effective in some scenarios, effectiveness of these techniques depend on the network bandwidth and network size as they rely on a closed control loop approach. The third approach is to attack directly to the side effects of congestion: HoL blocking [46]. In this case, mechanisms detect congestion and dynamically allocate new queues to isolate congested packets. One clear example of this approach is RECN [6]. In these cases, however, the implementation overhead is non-negligible. Another example of this approach is speculative reservation [47], which provides end to end flow control in order to alleviate the congestion using different VCs with different priorities, first sending the speculative packet and some flits with high priority and the rest of the packets with low priority. VOQ solutions [48] statically separate traffic, thus may alleviate congestion. However, they impede the use of adaptive routing.

Some works propose to replace DOR by dynamic routing policies which collect some networks metrics and use this information to decide an alternative path to route the messages avoiding congested areas, thus increasing network performance. This adaptive routing approach is the basis of solutions like RCA [49], which uses a composition of multiple global metrics collected using a piggybacking data in the messages from the whole network to decide which is the output port selected to forward a message, then avoiding the hotspots. Similarly, in [50] authors propose to collect congestion information from the whole network and to take routing decisions based on network status. On the other hand, PARS, proposed in [51], avoid the piggybacking problem on a saturated network using a dedicated network for sending congestion metrics based on the buffer state at certain routers. Like RCA, PARS uses the buffer state metrics to select proper paths in order to avoid hotspots. However, the effectiveness of such methods depend on the severity of congestion since the adaptive routing algorithm may also spread congestion over the network, thus worsening the situation.

2.1.8 Cache Coherence Protocols

Chip multiprocessor systems usually employ a shared memory programming model, which requires a cache coherence protocol to keep the information stored in cache memories coherent along the cache hierarchy. The on-chip cache is organized hierarchically, smaller and faster caches are allocated at cache levels near to the processors and the bigger and slower caches are allocated at lower cache levels. This structure provides to the processors an efficient on-chip storage capacity. The last-level cache (LLC) in CMP systems can be implemented in a distributed manner using cache banks, one on each different tile. While the higher levels of the cache hierarchy are always private to the associated core, different policies can be implemented for the LLC, but the common choices are two. First choice, an LLC bank can also be private to a core, then this LLC bank extends the core private cache capacity. The second choice is when each LLC bank can be a slice of the shared distributed LLC. This approach is usually preferred because, although this approach has higher access latencies than private LLCs, it provides higher cache capacity, thus avoiding to access main memory accesses that are more frequent when private LLCs are used. Notice that main memory accesses suffer an expensive access time to get the required data.

To keep the stored information coherent, the cache hierarchy requires a cache coherence protocol. The cache coherence protocol keeps data coherent among the different cores, private caches and shared caches. Several copies of a block can be allocated at different caches, thorough the cache hierarchy, but only one of such copies can be written at a time in oder to guarantee data coherency between all the block copies. Therefore, the cache coherence protocol usually is implemented as Single-Writer, Multiple Readers (SWMR). Where one cache has a copy of the block with write permissions, and the other copies of the block allocated on different caches only can read the data.

Cache coherence protocols can be classified in two types, invalidation-based protocols, and update-based protocols. When a write operation is performed on the cache, it has to be propagated along the network and notify to caches that store a copy of the block affected by the write operation. Update-based protocols send a copy of the new value to the caches with a copy (sharer caches). Invalidation-based protocols instead of sending the new value, they send an invalidation request to all the sharers. When the invalidation reaches the L1 cache the block is invalidated and an acknowledgment is sent back to the writer cache. If the cache wants to access to an invalidated block, it has to request the block again. Finally, when the writer cache receives all the acknowledgments from the sharer caches then the block is written.

The typical way of defining and implementing a cache coherence protocol is through a finite state machine (FSM) which indicates the evolution of the state of a cache block depending on the type access and coherence actions performed to it. One of the design choices of a coherence protocol is the number of steady states the blocks can have in L1 caches; typically the properties of each cache block are encoded using the five states proposed by Sweazey and Smith [52]; focusing on a private L1, the state of a cache block can be in one of the following states:

- M (Modified): only this L1 cache has a copy of the block with read and write permissions; the copy has been modified and the copy in the L2 cache is thus stale
- O (Owned): this L1 cache has a copy of the block with read-only permission and must provide the block when it is requested by other L1s; other L1 caches may have a read-only copy of the block in state S; the copy in the L2 cache may be stale
- E (Exclusive): only this L1 cache has a copy of the block with read and write permissions; the copy has not been modified

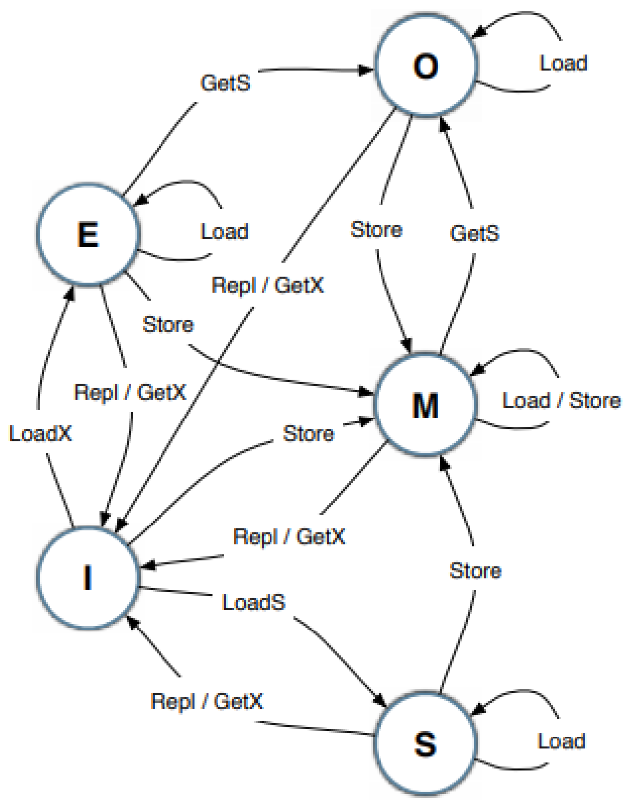


FIGURE 2.7: Simplefied FSM for the MOESI protocol.

- S (Shared): this L1 cache has a read-only copy of the block; other L1 caches may also have a read-only copy
- I (Invalid): the block is either not present in this cache or it is present but not valid

The three states M, S and I are the basic ones and allow to define the simpler MSI protocol, while states O and E are two optimizations which can be used to extend the MSI protocol, thus obtaining MOSI, MESI and MOESI protocols. An L1 cache which has a copy of a block in state M, O or E is referred to as the owner of that block, while the set of L1 caches holding a copy of a block in state S are referred to as sharers of that block.

Figure 2.7 shows a simplified FSM of the MOESI protocol (transient states are not shown). A cache block is initially in state I; then, when the core issues a read in exclusive mode (LoadX) and the block is not present in the cache, the block is fetched in the upper level cache hierarchy, and when arrives the block state changes to E. However, if the block issues a regular load (LoadS), the block state, when received, is switched to S. If a write (Store) request is triggered, the block goes to M state. The block is requested by sending respectively a GetS or a GetX request through the NoC; the requested data is

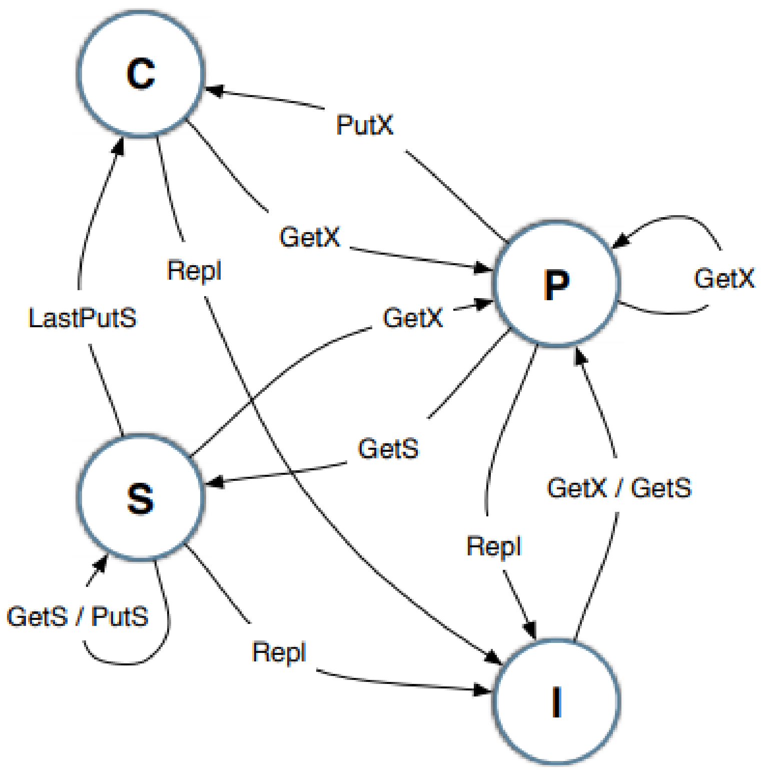
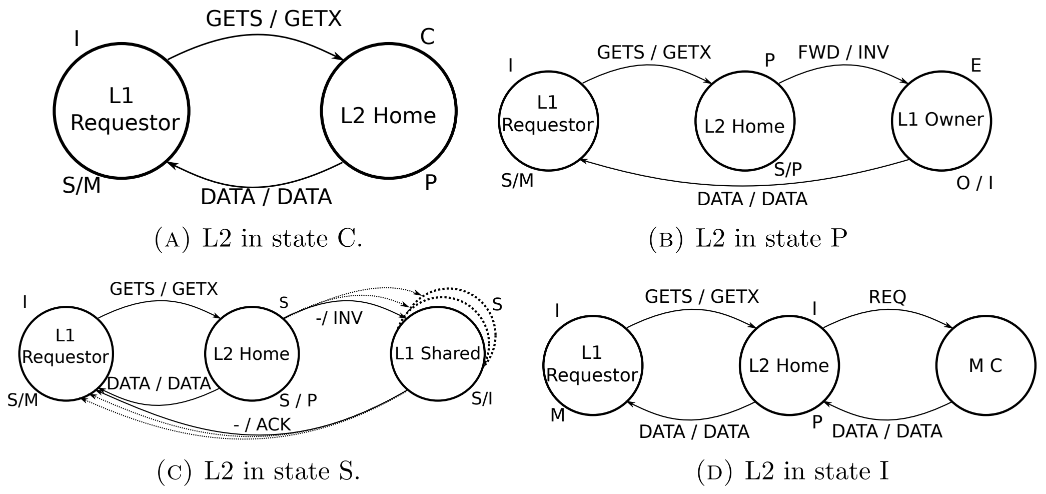


FIGURE 2.8: Simplefied FSM for L2 caches.

provided by the LLC or by the owner L1 depending on the state of the block. If the cache line is replaced (Repl), the block state goes back to I. The block is also invalidated if the cache receives a write request issued by another core (GetX), to preserve the SWMR invariant. If a store request from the local core is received in the L1 cache while the block state is E, O or S, then the block state changes to M. However, before writing the block the L1 cache must trigger a coherence request to get write permissions and must invalidate all the copies in the L1 caches. If a read request from the local core is received in the L1 when the state is not I, then the block keeps the same state and the read operation completes. If the cache receives a read request from another core (GetS) while the block state is M or E, the block is now shared, but this L1 cache changes the block state to O, this owner cache still having to write permissions and in charge of providing the block when it is requested by another L1 cache.

In the PROSA and PROSA-DD contributions in this thesis, we assume the MOESI protocol [53] at L1 while at L2 blocks we assume the protocol with the following cache block states: P (private), S (shared), C (cached), and I (Invalid) state. The Figure 2.8 shows a FSM of the protocol for L2 cache. A block in a P state means that only one L1 cache has a copy of this block. A block in S state on the L2 cache means that more than one L1 cache has this block allocated with read permission, but only one has the write



(A) L2 in state C. (B) L2 in state P

(C) L2 in state S. (D) L2 in state I

Figure 2.9: Coherence protocol transactions.

permission. In C state no L1 cache has a copy of the block. If the block is in I state, then the cache hierarchy has no copy of this block, and a new request to main memory is triggered. Inclusive caches are assumed and a write-back policy is used.

Figure 2.9 summarizes how the protocol manages the load and store transactions along the network and the cache hierarchy. In the figure, nodes are represented by circles. The current block state is represented with text placed just above the circle while the new state after the transaction is represented under the circle. Messages sent between nodes are represented by arrows. Messages for load and store operations are combined (e.g. GetS/GetX).

Whenever a L1 load miss occurs, a GetS message is sent to the L2 Home bank. Based on the block state at L2 different actions are performed. If the block is in S or C state (Figures 2.9a and 2.9c), the L2 sends the data to the L1 requestor. If the block is in P state (Figure 2.9b), the L2 sends a forward (FWD) message to the L1 with the block and the block state at L2 changes to S. When the FWD message arrives, the L1 cache sends the data to the L1 requestor cache and the block state changes to O. Finally, if at the L2 the block is in I state (meaning a miss occurs, Figure 2.9d), a REQ message is forwarded to the MC. The L2 receives the data and forwards it to the L1 cache requestor. The state of the block is set to P in the L2 and to E in the L1.

Whenever an L1 store miss occurs, a GetX message is sent to the L2 Home bank. Similarly, if the block is in C state (Figure 2.9a), the L2 sends the data to the requestor and changes the block state to P. A different case occurs when the block is in P state (Figure 2.9b). The L2 sends an invalidation message (INV) to the L1 owner cache. When the INV message arrives, the data is sent to the L1 requestor and the block changes from E to I. When the block is in S state (Figure 2.9c), the L2 sends the data to the L1 requestor and sends INV messages to all the L1 sharers. When each L1 sharer receives the INV message, it sends an ACK message to the requestor and changes the block state to I. Finally, if the block is in I state (miss) (Figure 2.9d), then a REQ message is forwarded to the MC. Once the data is received, it is forwarded to the L1. The block is put in P state in L2 and in M state in L1.

As we see, the protocol faces mainly four possible paths depending on the type of operation (load/store), L1 access type (miss/hit) and L2 access (miss/hit). Whenever a miss occurs at L1 and L2 the NoC gets involved and the memory transaction latency is increased. Factors that may affect significantly memory latency are the distance between the L1 requestor and the L2 Home bank, distance between the L2 Home bank and the MC, and the network congestion.

2.2 High-Assurance NoCs

Different security levels may be required when multiple applications run on the CMP at the same time. Sometimes, mission-critical applications require their execution in a bounded time period and/or these applications transport data that must be inaccessible by other applications. When high-assurance property is required, CMPs need to be adapted in order to deliver security and reliability guarantees.

In high-assurance systems it is a common practice to break the system into a set of domains, which are to be kept separate and should have no effect (i.e., interference) on one another. In CMPs, however, such controlled domain partitioning is not straightforward since the on-chip interconnection network (NoC) is a shared resource by all domains. Even assuming to spatially isolate domains inside physical compute and memory partitions, memory controller (MC) reachability becomes an issue, due to the need to cross intermediate partitions. As a result, the NoC resources are necessarily shared between communication flows from different domains, and the proper course of action should be taken to avoid domain interference.

Several degrees of non-interference can be enforced on the NoC. A first approach to loosening interdependencies among communication flows consists of delivering quality of service (QoS) guarantees. In fact, most QoS techniques aim at limiting flow rates, while restoring nominal rates in the absence of contention. While QoS-augmented NoCs can typically protect from denial-of-service (DoS) and bandwidth depletion attacks between domains, they cannot easily avoid an information leak associated with latency and throughput variations of communication flows as a function of network state. In fact, they can be used as timing channels by an attacker either to infer confidential information from a protected high-security program (side channel attacks) or to have a malicious program deliberately leak information covertly when direct communication channels are protected (covert channel attacks) [54].

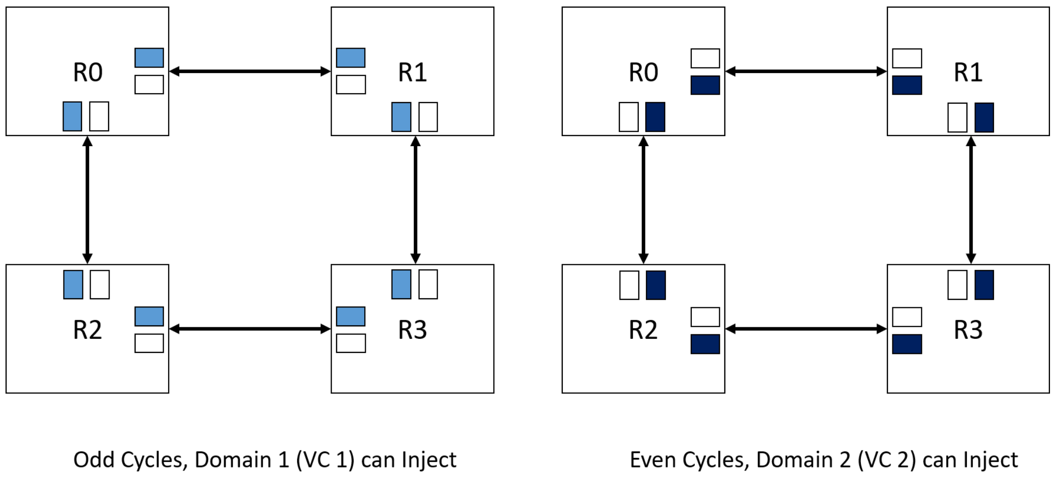

R0
R1
R2
R3
Odd Cycles, Domain 1 (VC 1) can Inject
R0
R1
R2
R3
Even Cycles, Domain 2 (VC 2) can Inject

FIGURE 2.10: Reference solution: VC partitioning coupled with time-division multiplexing.

When the protection against such timing channel attacks is required, even cycle-level variations of communication performance should be prevented, a scenario that we hereafter denote as *strong isolation* of domains. Interestingly, implementing strongly isolated domains makes it also easier to contain the propagation of faults, and does not require to account for all possible system-level interactions for the sake of certification.

In order to deliver strong isolation to networked domains, the NoC must be designed to guarantee the non-interference property in its strictest sense: injection of packets from one domain cannot affect the timing of packet delivery from other domains.

2.2.1 Time Division Multiplexing

The reference solution to avoid any kind of domain interference consists of partitioning the virtual channels and time-multiplexing the physical channels and crossbars between different domains such that channels are only allowed to propagate packets from different domains on different cycles. Figure 2.10 shows an example of this approach, in that case with two domains, where domain one (VC 1) only can transmits messages in odd cycles, and domain two (VC 2) transmits the even cycles. This TDM partitioning scheme ensures that latency and throughput of each domain are completely independent of the other domain's load. However, this baseline scheme is heavily sub-optimal and non-scalable, since packets will have to wait as many cycles as the number of concurrent domains minus one at each hop. The incurred penalty grows significantly with the physical distance of the receiver end node.

Related Work Linked with Time Division Multiplexing

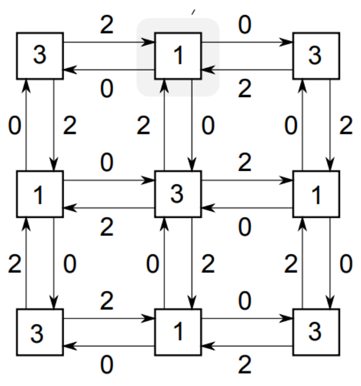
Prior approaches to TDM-based scheduling in NoCs loose relevance when they are viewed from the viewpoint of the concurrently conflicting requirements of latency optimization, area efficiency and architectural flexibility. Numerous designs perform TDM scheduling at the time-slot level [55][56][57]. When using such architectures, the scheduling is typically performed offline (and assumes perfect a priori knowledge of the applications expected to be running on the system), and then statically applied to the entire NoC [58] [7]. However, in this type of approach the latency overhead can be quite substantial.

AEthereal [55] employs pipelined TDM (at the time-slot level) and circuit-switching to guarantee performance services. Traffic is separated into two main classes: 1) guaranteed service (GS) and 2) best effort (BE). Excess bandwidth not used by GS flows is given to BE flows. Packets on a single connection are always ordered, but ordering cannot be enforced between connections. This approach incurs a substantial programming overhead of time slots at network interfaces.

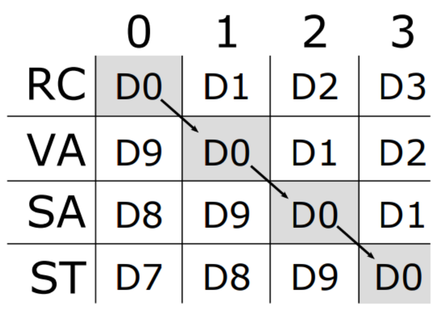
The SuperGT NoC [59] is an evolution of AEthereal providing three QoS classes. Aelite[56] simplifies the router architecture by providing only GS, and AElite moves one step further by including multicast traffic and fast virtual-circuit setup. Argo [57] allows schedules to evolve at the granularity of a single cycle, even when the routers have more than one pipeline stage. The resulting hardware cost is quite low, but the latency overhead can be substantial. In [54], static network partitioning in space and time is employed to provide multi-way isolation among the supported domains. This multi-way isolation property comes at a high performance cost, which is alleviated by the introduced reversed priority with static limits (RPSL) mechanism. It uses priority-based arbitration and static limits to guarantee one-way isolation between high-security and low-security flows.

A recently introduced architecture, called SurfNoC [60], employs optimized TDM scheduling, also applied at the VC level, to minimize the latency overhead. However, the required hardware is expensive. Achieving low-cost implementations with SurfNoC would increase the latency overhead of static scheduling. The current state-of-the-art in TDM-based scheduling is PhaseNoC [61]. It improves the VC-level scheduling proposed by SurfNoC by pre-configuring the network in order to receive packets from the same domain at all the input ports in the router each cycle, and performing the arbitration in the next cycle of this incoming domain,

Figure 2.11a shows an example of PhaseNoC network configuration for a 2D mesh. Notice that for each router all the incoming links transport data for the same domain. Following this approach, PhaseNoC implements a perfect scheduling behavior. This perfect schedule allows PhaseNoc forward messages along the network without any stop, due to the domains are scheduled following the router pipeline at the same time as the the message advances crossing the router, Figure 2.11b shows an example of cycle-by-cycle operation for a 4-stage pipelined PhaseNoC router. In each cycle, all router parts are utilized, with each pipeline stage serving (allocating or switching) a different domain (group of VCs). PhaseNoC meets the first requirement, by minimizing the



(A) NoC Domain Configuration.



	0	1	2	3
RC	D0	D1	D2	D3
VA	D9	D0	D1	D2
SA	D8	D9	D0	D1
ST	D7	D8	D9	D0

(B) PhaseNoC Scheduling Pipeline.

FIGURE 2.11: PhaseNoC Scheduling and PhaseNoC Network Configuration

latency overhead. However, it lacks flexibility, because PhaseNoC needs to modify the network (adding stages at the router) to support a higher number of domains. PhaseNoC proposes to divide the network into $x + y+$ and $x - y-$ to support a higher number of domains. However, following this approach, it can not guarantee the non-interference property any more, for which it would need input speedup similarly to SurfNoC. One of our contributions, Token-based TDM, is compared with the current state-of-the-art on TDM, PhaseNoC.

In this chapter we have introduced background for networks-on-chip and some related work addressing the topics this thesis tackles. In the next chapter we introduce and evaluate all the contributed techniques.

Chapter 3

Thesis Contributions

In this chapter we describe the different techniques provided in this thesis and their evaluation. For the sake of understanding, they are collected from the associated publications exposed in the previous chapters, avoiding in this chapter any duplicity between publications.

In this thesis, we propose some techniques to network performance improvement following different approaches. As said previously, the two first proposals are oriented to maximize the throughput of the network for NoCs when using fully adaptive routing algorithms. One is oriented to improve buffer utilization and in that way improving the network throughput. The second one is focused on congestion situations identifying the congested destinations and isolating the flows with this destination, thus, reducing latency of non-congested traffic.

The second approach focuses on network latency reduction, this proposal is a novel hybrid circuit-packet switching network. This approach gets benefit from the coherence protocol to setup the circuit before it is needed, thus, hiding the setup latency.

The last proposal of this thesis is a new TDM network based on the CDG. This approach changes the basic idea of TDM networks using a router arbitration domain instead of a network arbitration domain to reduce the network latency in high assurance scenarios.

3.1 Performance Improvement with Fully Adaptive Routings

3.1.1 Type Based Flow Control and Safe/Unsafe Routing

The first proposal is a codesing of flow-control and an adaptive routing algorithm. A novel flow control mechanism is presented, referred to as Type-Based Flow Control

(TBFC), which implies a the reduced flow control strategy using minimum buffer resources, while still allowing virtual cut-through switching. Then, on top of TBFC we implement the Safe/Unsafe Routing algorithm (SUR). This algorithm achieves the same low network latency as previous adaptive routings, however, our proposal has a higher performance due to a properly balanced utilization of input port buffers. Then, our proposal achieves a lower network latency after the saturation point for adaptive routing algorithms.

3.1.1.1 Type Based Flow Control (TBFC)

TBFC aims to offer balanced buffer utilization to the routing algorithm. But, before entering into details, we need to differentiate between two crossbar switching strategies that may be implemented inside the router. The first one is termed *flit-level switching* and improves buffer utilization by allowing the router to multiplex flits of different packets to advance through the crossbar while directed to the same output port, but mapped to different virtual channels. The second one is termed *packet-level switching* and consists in preventing the router to multiplex flits from different packets to the same output port. In this approach, when a packet header gets access to the crossbar, the remaining flits of the packet will keep the crossbar connection and follow without interruption.

Flit-level switching is conceived for wormhole switching while packet-level switching is conceived for virtual cut-through. However, both approaches can be used for any switching mechanism. Nevertheless, taking flit-level or packet-level switching into account is important since it affects how flow control can be implemented. In the next two sections we describe our flow control method for both crossbar switching strategies.

3.1.1.1.1 TBFC with Flit-Level Crossbar Switching

Figure 3.1 shows a traditional credit-based flow control implementation for a pair of output-input ports. Flit-level crossbar switching is assumed. At each output port the router needs some control information. Indeed, for each VC we need: one field for the number of credits available (*CRED*), one field to determine whether the VC is being used or not (*USED*), and the input port and VC that has this VC granted (establishes a link between the input port and the granted VC).

When a packet header is routed, the router sends a request to the target output port. At that port, the virtual channel allocator (VA) checks whether there is any free VC that has enough credits at the next router for the whole packet (we assume virtual cut-through switching). Then, the router arbitrates (in round-robin fashion) among all the requests and assigns the VC to the winning request. It stores the winning input port

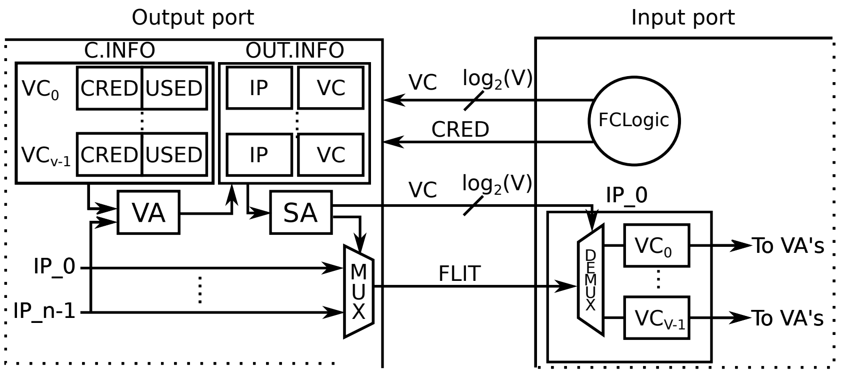


Figure 3.1: Traditional credit-based flow control assuming flit-level crossbar switching.

and virtual channel in the control info structure associated to the VC. It also decrements the available credits in the control info associated to the VC.

At Switch Allocation (SA) stage, the arbiter selects the input port that will send a flit through the output port the next cycle. SA selects this port between the input ports assigned to this output port by the VA stage. The arbiter rotates the priorities whenever an input wins the access, thus implementing flit-level crossbar switching. The router sends the flit together with the VC ID to the next router. The next router uses the VC ID to demultiplex and allocate the flit into the correct VC. When a tail flit is forwarded the VC is freed and can be assigned again to a new packet header.

At the input port, when one flit is forwarded, the Flow Control Logic (*FCLogic*) sends a credit back to the upstream router. To do this, the router needs at least $log_2\ (V)+1$ wires to indicate the VC that will receive the credit (signals *VC*), where *V* is the number of virtual channels at each input port. It also sends the control signal *CRED*. Upon reception, the credit counter associated with the VC is incremented.

Figure 3.2 shows TBFC when applied to flit-level crossbar switching. The first difference between the traditional flow control and TBFC is the flow control information structure. TBFC adds two new fields per output port: *FREE* field, which accounts for the number of available VCs, and *TYPE* field, which accounts for the number of packets stored at the input port labelled with a particular type (we will later describe the type usage in the routing algorithm). Then, for each VC, the control info keeps the *CRED* counter and the associated info for the assigned input port and VC. However, the *USED* field is removed.

Contrary to the baseline flow control, the rules (at VA stage) to assign a VC to an incoming request are different. An improvement in the VC selection was proposed in [62]. In our design, the VA stage checks only the number of free VCs and the number of labelled packets (*TYPE* field) (more details described in the next section). When one request wins the output VC, the input port of this request is assigned to the output

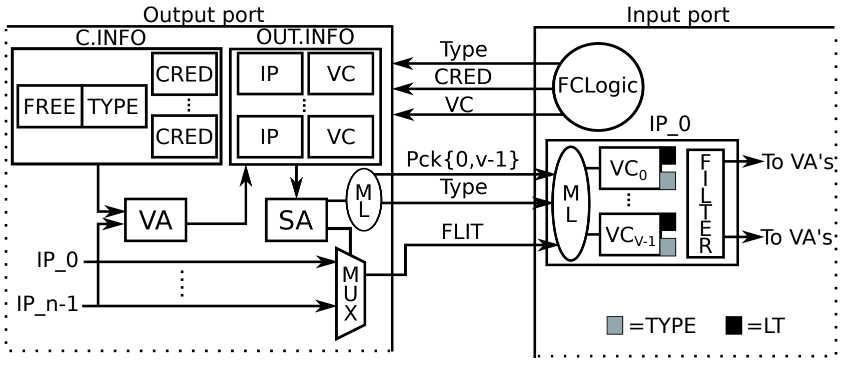


Figure 3.2: TBFC flow control assuming flit-level crossbar switching.

VC. The winning input port and input VC are associated to the control info for the VC. The number of FREE VCs is decremented by one and, if the packet sent downstream is labelled, then the TYPE field is incremented by one.

At SA stage, the router selects the input port to pass through the output port and forwards the flit to the next router. At crossbar stage, the router does not send the VC selected. Instead, it performs a *packet* $\rightarrow$ *ID* mapping (*ML* block) to assign one identifier to the packet. When a head flit is sent, the router sends also the type of the packet. The identifier, the packet type and the flit are sent through the link to the next router. All the flits of the same packet will use the same identifier and only the packet header will contain the type field.

When the downstream router receives the head flit, the packet type and the identifier, a new mapping is performed (*ML* block). In this case, an $ID \rightarrow VC$ mapping is performed, thus allocating the new packet in one free VC. After the head flit, all flits that arrive with the same identifier are kept in the same VC through the mapping logic.

Each input VC has one bit associated, referred to as Last token (LT). When one head flit arrives and is allocated in one VC, this VC sets its LT bit to one and the LT bit of the other VC is reset to 0.[1] This field is used to guarantee in-order delivery of packets. Indeed, if two VCs at the same input port have a header packet with the same destination, then the oldest one (the one with LT bit set to zero) is the one to access the VA stage. Otherwise, both packets may access the VA stage. Notice that if the routing algorithm implemented on top of the flow control allows out-of-order delivery, then the LT bit and its associated logic can be removed. This will be the case of the SUR algorithm. In addition, each input VC will contain a *TYPE* bit which will indicate if the packet allocated on that VC is labelled or not. This bit is updated with the type information received when a header flit arrives.

Whenever a head flit is sent downstream, the *TYPE* bit is transmitted upstream. Upon reception, the upstream router decreases the *TYPE* counter. In any case, the *FREE*

[1]If the router has more than 2 VCs, the LT field will need log_2 *(V)* bits and will be updated following an algorithm similar to the ones used in caches with Least Recently Used (LRU) replacement policies.

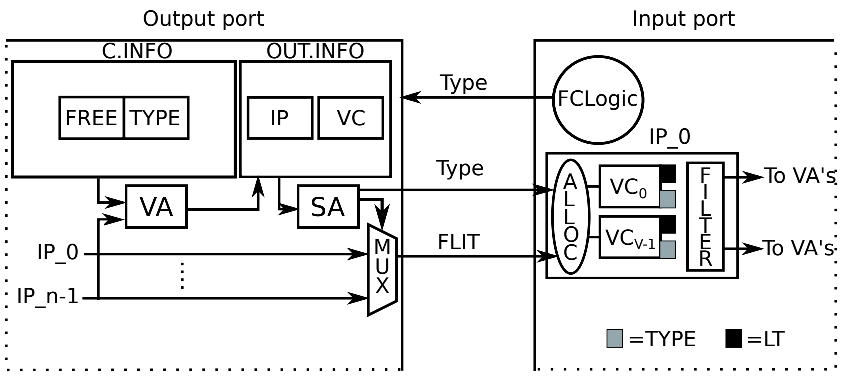


FIGURE 3.3: TBFC flow control assuming packet-level crossbar switching.

field is increased by one. Notice also that the *CRED* field is still used in TBFC. This is needed as we are assuming flit-level crossbar switching, which may provoke different reception and transmission rates at the input ports.

3.1.1.1.2 TBFC with Packet-Level Crossbar Switching

Now, we focus on the TBFC mechanism when packet-level crossbar switching is enabled. Notice that in this case packets will not be mixed in the crossbar. This fact, together with the VCT switching we assume will guarantee that reception and transmission rates of packets at the input ports will be equal. This means that whenever a packet header wins the access to the crossbar, the whole packet can be transmitted and will not stop its transmission until reception at the downstream router. This fact simplifies greatly the TBFC mechanism, as we will see.

Figure 3.3 shows the TBFC mechanisms with packet-level crossbar switching. The first thing to notice is the simplification of the control structures. Now, we do not need credits anymore and we only need to keep which input port and VC got access to the VCs downstream through an output port. In particular, the *FREE* and *TYPE* fields are still used. Also, the mapping logic blocks are removed. Indeed, when a packet gets access to the output port will be transmitted uninterruptedly.

The VA stage is not modified as it takes into account only the number of free VCs (*FREE* field) and number of packets labelled at the downstream router (*TYPE* field). The SA stage is also simplified since there is no flit multiplexing at the output. The SA stage needs only to arbitrate among competing packets but must keep the token priority fixed until the packet's tail leaves the router. This guarantees no multiplexing at the crossbar.

At the downstream input port side the logic is also simplified. There is no *ML* logic and the FCLogic only sends back upstream the type of the packet that just started to leave the input port. LT bits are still used if in-order delivery is needed to be guaranteed and the type field per VC is needed to remember whether the packet in the VC is labelled or not.

3.1.1.2 Safe/Unsafe Routing Algorithm (SUR)

Safe/Unsafe Routing (SUR) is a new fully adaptive routing algorithm adapted to the TBFC strategy. Each input port contains two VCs, while each VC is assigned a buffer to keep the whole packet. The SUR algorithm is fully adaptive and relies on an escape path to prevent deadlocks. The underline routing algorithm to implement this escape path is XY. The algorithm can work either on switches using flit-level crossbar switching or packet-level crossbar switching. SUR works on n-dimensional meshes and n-dimensional tori.

TBFC enables packet labeling and exposes this information to the routing stage. In our case, the SUR algorithm labels packets as *safe* or *unsafe*. Packets are labelled when they are sent to a downstream router as follows:

- In an n-dimensional mesh a packet is delivered and kept in the next router as a *safe* packet if the next hop conforms to the baseline routing algorithm. Otherwise, the packet is labelled as *unsafe*.
- In an n-dimensional torus a packet is delivered and labelled in the next router as *safe* if one of the following conditions is met:
  - The next hop of the packet is to traverse a wraparound link along dimension d, and the packet does not need to traverse a wraparound link with a lower dimension than d.
  - The packet does not need to traverse any wraparound link from the current router to the destination and the next hop conforms to the baseline routing.

  If any of these two conditions is not met, then, the packet is delivered and labelled as *unsafe* packet.

With this classification, the routing algorithm will decide which outputs ports are eligible for packets. In detail, output ports along the minimal paths to destination will be eligible. *Safe* packets will be routed without any restriction and *unsafe* packets will be routed only in some particular conditions. To assist this routing algorithm we define a *check port* function suitable for meshes and tori. Algorithm 1 show the function *check-port*(f,s). This function avoids filling any input port with only *unsafe* packets. It checks, for a given input port, the number of free VCs (f) and *safe* packets (s) as follows:

```
Input:
    The number of free VCs in the downstream node, $f$;
    The number of safe packets in the downstream node, $s$;
Output:
    Whether the packet can route to the downstream node;
1: if $f > 1$ then
2:     return true;
3: end if
4: if $f = 1$ and $s \geq 1$ then
5:     return true;
6: end if
7: if $f = 1$ and $s = 0$ then
8:     and the packet will be delivered and labelled as a safe packet in the next router
9:     return true;
10: end if
11: return false;
```

Algorithm 1: check-port(f,s)

- $f > 1$, the packet can be delivered because there is more than one free VC in the input port at the next router.
- $f = 1$ and $s > 0$, the packet can be delivered because there is at least one *safe* packet in the next router.
- $f = 1$ and $s = 0$, the packet can be delivered only if the packet is safe at the next router; otherwise, the packet is blocked or takes another output port.
- $f = 0$, the packet is blocked or takes another output port.

The proposed fully adaptive routing algorithm for 2-D mesh is shown in Alg. 2, where f_i+, s_i+ represent the number of free VCs and *safe* packets in the input port in the neighbor router attached to the current node C along dimension i in the positive direction, respectively. Similarly, f_i- and s_i- represent the number of free VCs and *safe* packets in the input port along dimension i in the negative direction, respectively. The algorithm takes as inputs the coordinates of the current and destination nodes, number of free slot and number of safe packets of all neighboring input ports. The available channel set and the selected output channel are initialized to $\emptyset$ and *null*, respectively. If the current node is the destination, the internal channel is selected to consume the packet. Otherwise, if the offset along dimension i (dimensions 1 and 2 in Alg. 2) is greater than 0 and check-port(f_i+,s_i+) returns a true value, then the packet can be delivered along a c_i+ channel. If the offset along dimension i is less than 0 and check-port(f_i-,s_i-) returns a true value, the packet can be delivered via a c_i- channel. The check-port(f_i,s_i) function allows to add the channel c_i+ or c_i- to S if the packet can advance along dimension i.

```
Input:
    coordinates of the current node $C : (c_1, c_2)$,
    coordinates of the destination $D : (d_1, d_2)$,
    free buffers: ($f_1-$,$f_1+$, $f_2-$, $f_2+$),
    safe packets: ($s_1-$, $s_1+$, $s_2-$, $s_2+$);
Output:
    selected output channel;
 1: S=0;ch=null;
 2: if $C == D1$ then
 3:     ch=internal; return true;
 4: end if
 5: for $i == 1$ to 2 do
 6:     if $d_i - c_i > 0$ and $check - port(f_i+, s_i+)$ then
 7:         $S =\leftarrow S \cup \{c_i+\}$;
 8:     end if
 9:     if $0 > d_i - c_i$ and check-port($f_i-, s_i-$) then
10:         $S \leftarrow S \cup \{c_i-\}$.
11:     end if
12: end for
13: if $S \neq \emptyset$ then
14:     $ch$ = select($S$);
15: end if
16: if if $S = \emptyset$ then
17:     $ch = null$;
18: end if
```

Algorithm 2: safe-unsafe-2D-meshes

Alg. 3 presents the fully adaptive routing algorithm for 2-D torus. The difference lays in the computation of the direction to take in each dimension. Also, the *check-port* function must take into account the additional rule to define a packet as unsafe based on the crossing of wraparound links (see previous conditions).

Finally, the proposed routing algorithm randomly selects an output channel from S if it is not null. Otherwise, the packet is blocked and routed in the next cycle.

3.1.1.2.1 Deadlock-freedom Property

Deadlock-freedom is a very important property to routing algorithms. In our case, SUR algorithm is deadlock-free. We deduce this property using a contradiction approach. We first focus on 2-D meshes and then extended it to 2-D torus.

Let us assume we have a cycle in a 2D mesh. Such a cycle will have dependencies between $x \to y$ and $y \to x$ channels. $x \to y$ dependencies are allowed by the underlying routing algorithms but $y \to x$ dependencies are not allowed. Packets stored in an Y input port will be labelled as unsafe as they are requesting an X output port. In order to create deadlock, packets inside a cycle should not advance. This means either all the

```
Input:
    coordinates of the current node $C : (c_1, c_2)$,
    coordinates of the destination $D : (d_1, d_2)$,
    free buffers: $(f_1-, f_1+, f_2-, f_2+)$,
    safe packets: $(s_1-, s_1+, s_2-, s_2+)$;
Output:
    selected output channel;
 1: S=0;ch=null;
 2: if $C == D1$ then
 3:     ch=internal; return true;
 4: end if
 5: for $i == 1$ to 2 do
 6:     if $0 < d_i - c_i \leq k/2$ or $d_i - c_i < -k/2$ and $check - port(f_i+, s_i+)$ then
 7:         $S = \leftarrow S \cup \{c_i+\}$;
 8:     end if
 9:     if $d_i - c_i > k/2$ or $-k/2 \leq d_i - c_i < 0$ and check-port($f_i-$,$s_i-$) then
10:         $S \leftarrow S \cup \{c_i-\}$.
11:     end if
12: end for
13: if $S \neq \emptyset$ then
14:     $ch$ = select($S$);
15: end if
16: if if $S = \emptyset$ then
17:     $ch = null$;
18: end if
```

Algorithm 3: safe-unsafe-2D-torus

buffers are full in the cycle or the routing restrictions do not allow packets to advance. The first condition does not hold since it would mean that in the Y input port both VCs are storing unsafe packets. This can not happen since unsafe packets can be forwarded only if both VCs are available, or one VC is available and the other VC is holding a safe packet. The second condition (the routing restrictions do not allow packets to advance) does not apply neither. Indeed, if one packet is at a Y input port requesting an X output port the associated X input port will store either one, or two safe packets, or will be completely empty. In the case of storing one safe packet or being empty the unsafe packet at input Y can advance, thus no deadlock. In case of storing two safe packets, both can advance since they will always have in front of them safe packets, which potentially will move as they are using acyclic paths (conformed by safe packets using the underlying deadlock-free baseline routing algorithm). Therefore, not blocking packets in the cycle. In other words, safe packets, stored through deadlock free paths have always a reserved VC, thus always advancing. Unsafe packets can cross cycles but never filling up input buffer, thus avoiding deadlocks. Therefore, for any potential cycle, unsafe packets will never take all resources in an input buffer. They, in turn, will also advance when both VCs are available to them.

For the n-D torus case we follow a similar approach. In this case wraparound links

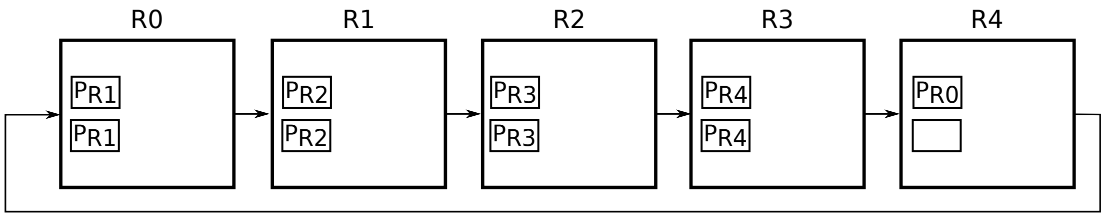


FIGURE 3.4: Example of deadlock-freedomness in a 1D ring network.

take also an important role. If the packet does not need to traverse any wraparound link, the packet follows the behaviour described above. So, the packet will not create a deadlock. If the packet is stored in a router connected to a wraparound link, then if the packet requests the wraparound link with the lowest dimension that the packet needs to traverse, the packet is following the baseline routing. This means that the packet will be delivered and labelled as *safe*. On the other hand, if the packet requests an output port connected to a wraparound link and the dimension of this wraparound link is not the lowest dimension that the packet needs to cross, then the packet will be delivered and labelled as *unsafe*. As we have shown before this happens only if the next input port is empty or has one free VC and the packet stored in the other is safe. So, input buffers will never fill with unsafe packets. In other words, in the case of 2D torus, all the input buffers will always allow safe packets to advance.

Let us expose an example in Figure 3.4, to simplify only west input port are shown. In this 1-D torus all switches want to send messages to a router located at two hops to their right. R0 keeps in the west input port two packets from R4 with destination node R1, packets labelled as P_{R1}. These two packets are safe because they arrived from R4 crossing the wraparound link with the lowest dimension required. R1 contains two packets with destination R2 that came from R0, packets labelled as P_{R2}. These two packets are safe because they do not require to cross any wraparound link and the packets follow the baseline routing. R2 and R3 have the same situation as R1. R4 has one packet with destination to R0, packet labelled as P_{R4}. This packet is unsafe because the packet comes from a router not connected to the wraparound link and needs to traverse the wraparound link to reach its destination node. Therefore, R3 will not send another unsafe packet to R4. Then all the packets can advance because the packets at R3 can be forwarded and consumed in R4.

3.1.1.2.2 TBFC+SUR Example

Figure 3.5 shows an example of TBFC+SUR. Suppose two switches, one above the other. The north input port of the router located below is empty, so the values stored in the fields FREE and SAFE at the output port control info are 2 and 0 at router located above, respectively. Also let us assume 1 cycle of fly link. At time t=1 the header flit of packet one (P1) with destination located at south, in the figure the blue packet,

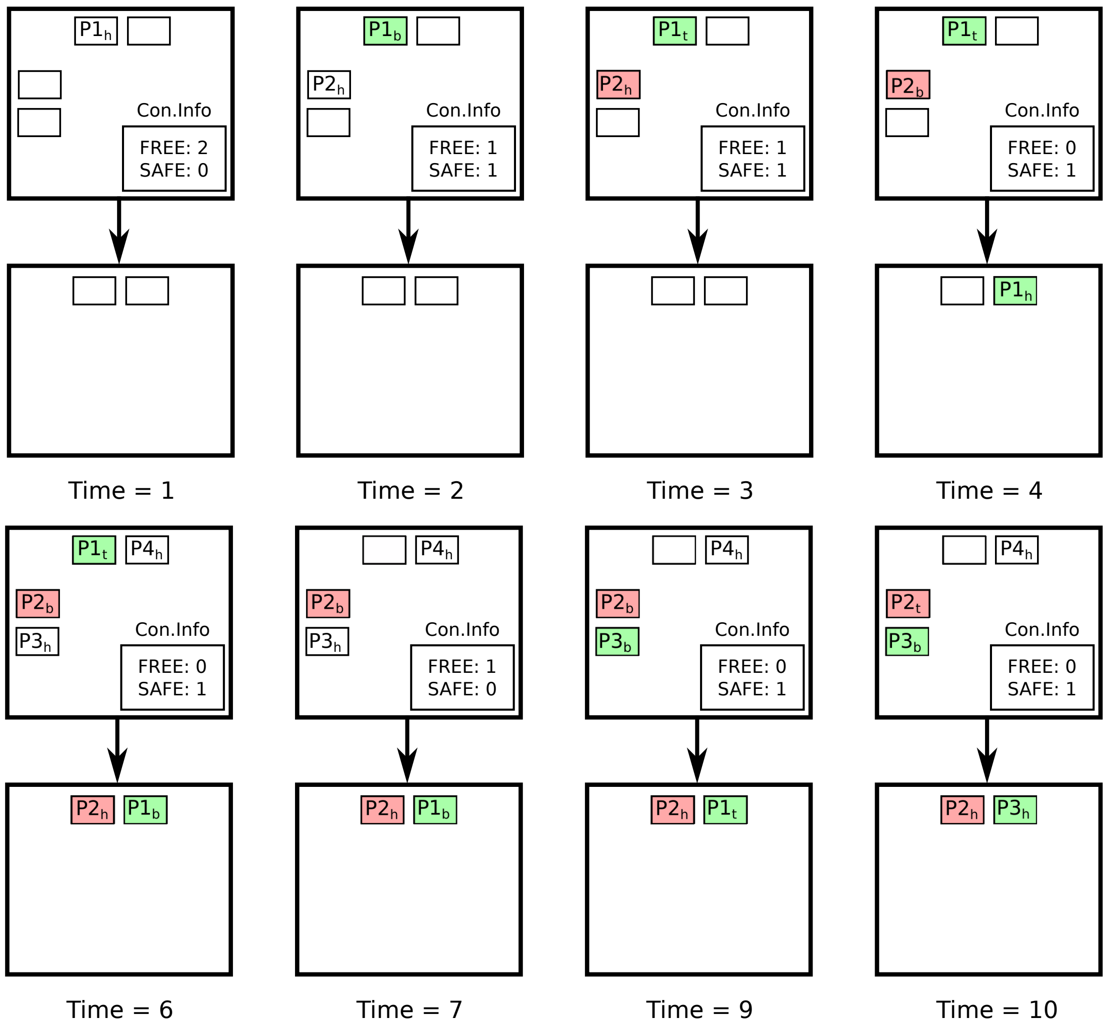


FIGURE 3.5: Logic example at the router's input port.

arrives to VA/SA stage. The arbiter selects this packet to be forwarded in the next cycle through the south output port, as this packet will be forwarded following the baseline routing it will be sent as safe packet. The output port control is updated, FREE is decremented by 1 and SAFE is incremented by 1 because the packet will be forwarded as safe packet. At t=2 the header flit is in the crossbar stage then the mapping logic computes the ID and the router sends this ID, the packet type (safe) and the flit to the next router. In this cycle the next flit of P1 will be forwarded in the next cycle. At time t=3 a new header flit arrives to VA/SA stage, in this case the destination of this new packet (P2), red packet, is located at south-east. The arbiter checks if this packet could win the output port (check-port() function). Using the south port P2 will be delivered as unsafe packet because it does not follow the baseline routing. However, the field safe is 1, meaning that the next input port has one safe packet, so an unsafe packet can be forwarded. Then, the arbiter selects this new packet to be forwarded in the next cycle through the south output port. The output port control is updated, Free is decremented by 1. In this case the SAFE field is not incremented because the router sets the packet type as unsafe.

At t=4, the header flit of P1 arrives to the input port with identifier equal to 1. Both VCs are empty, and the local variables to assign the virtual channel to an identifier are empty also. Then, the mapping logic allocates the flit to VC_1, and keeps the identifier and the packet type in the local variable. P2 is in the crossbar stage and follows same procedure as P1. At time t=5 a body flit of P1 with identifier 1 arrives. The mapping logic knows where it has to allocate the flit and puts it in VC_1. In the next cycle, t=6, packet header of P2 arrives to the input port with identifier equal to 0, then the mapping logic allocates the new message at VC_0 and keeps the identifier in the local variable. Also, packet P1 arrives to VA/SA stage in the second router and wins the output. So the router sends back the flow control information, VC FREE and the type of the packet stored in the local variable. And in the router above, two news packets arrive at the VA/SA stage, but the function check-port() does not allow to win this output port. Therefore, the tail flit of the packet P1 is selected to be forwarded in the next cycle.

At time t=7 flow control information is received, so the router updates the output port control information, increasing FREE by 1 and decreasing SAFE by 1. This means that a new packet could be forwarded through of this output port. Packets P3 and P4 are waiting to be forwarded. P3 has the destination at south, and P4 at south-east. This means that the function *check-port* allows P3 to be forwarded and blocks P4, because if P4 is forwarded both packets in the input port will be unsafe, and this could produce deadlock. Then P3 is selected to be forwarded in the next cycle as safe packet. At time t=9, the tail of P1 arrives at the input port, as the header has been forwarded, then the mapping logic cleans the identifier from the local variable. If at this point a new header flit arrives, this VC can be allocated again. In the router located above, the packet P3 is at crossbar stage, and is forwarded as we explain above. At time t=10, the flit header of P3 reaches the input port and the mapping logic allocates this new message to VC_1, keeping in the local variable the identifier and the type of the message.

3.1.1.3 Performance Evaluation

In this section, we perform an evaluation and analysis of TBFC+SUR. In particular, we first describe the analysis tools and simulation parameters. Then, we show the performance results for TBFC and SUR. We analyze two scenarios: a 2D mesh with 64 switches and a 2D Torus with 64 switches.

3.1.1.3.1 Analysis Tools and Parameters

The tool we use for this analysis is an event-driven cycle-accurate simulator coded in C++. The simulator models any network topology and router architecture. We modelled a 4-stage pipelined router with VCs and flit-level crossbar switching. Table 3.1 shows the simulation parameters used for the 2D mesh scenario. Transient and permanent messages relate to the number of messages processed until the simulator enters the permanent state and finishes the simulation, respectively. In this scenario, we analyze three routing algorithms: deterministic routing (XY), fully adaptive routing (FA) using the typical credit-based flow control and SUR routing (SUR) with TBFC.

Parameter	FA, SUR_2VC, XY
Network topology	4x4 mesh
VCs at each input port	2
Message size, Flit size	80 bytes, 4 bytes
Queue size	20 flits
Fly link	1 cycle
Transient, Permanent msgs	10000, 10000

Table 3.1: Parameters and values used for the experiments in 2D mesh.

For the torus scenario, the same parameters were used except for the number of VCs and queue size at each input port. These parameters depend on the routing algorithm and flow control scheme used. Table 3.2 shows the values in the torus scenario. In torus scenario we analyze five routing algorithms: deterministic routing (XY), fully adaptive with one adaptive channel and two escape channels (FA), fully adaptive with the bubble flow control [40] (FA bubble) using one adaptive channel and one escape channel with double size (to implement the bubble). Also, we analyze SUR with two and three virtual channels (SUR 2VC and SUR 3VC). These configurations (except SUR 3VC) are the ones with minimum buffer requirements to become deadlock-free and to guarantee VCT switching.

Routing	VCs	Queue Size
FA_Bubble	2	20 flits adap, 40 flits esc
FA	3	20 flits
SUR_2VC	2	20 flits
SUR_3VC	3	20 flits
XY	2	20 flits

Table 3.2: Parameters and values used for the experiments in 2D torus.

We evaluate four traffic distributions: bit-reversal, transpose, uniform and hotspot. In bit-reversal traffic, the node with binary value a_{n-1}, a_{n-2},..., a_1, a_0 communicates with node a_0, a_1,..., a_{n-2}, a_{n-1}. For transpose traffic with binary value a_{n-1}, a_{n-2},..., a_1, a_0 sends packets to node $a_{n/2-1}$,... a_0, a_{n-1}, ...$a_{n/2}$. Finally, in hotspot traffic, ten randomly chosen nodes send 20% of their traffic to an specific node and the rest of traffic to any other node with equal probability. The rest of nodes keep injecting using a random uniform distribution.

3.1.1.3.2 Performance Results

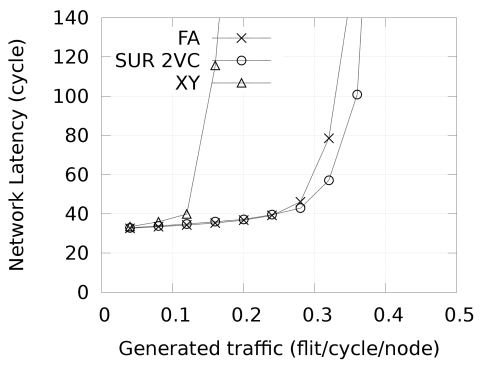
Figures 3.6 presents the results for the 2D mesh scenario. Figures 3.6a and 3.6b show the performance results for the bit-reversal traffic. In this scenario, our method reaches similar results on throughput than the ones achieved by FA. However, SUR improves latency close to saturation, when compared to FA algorithm. In any case, both adaptive algorithms (SUR and FA) outperform XY. With transpose traffic, Figures 3.6c and 3.6d, SUR outperforms FA by about 10% in network throughput. Latency is also improved by SUR when working close to saturation. For the other traffic distributions (uniform and hotspot; rest of Figure 3.6) we see similar results for the three routing algorithms. SUR, FA, and XY achieve similar network throughput. However, SUR latency is slightly improved close to network saturation.

Next, we analyze the results for the torus scenario. In this case, differences are much more significant. Figures 3.7a and 3.7b show the performance for bit-reversal traffic. SUR 2VC improves network latency achieved by FA and FA bubble. This is achieved by using less buffer resources (2VCs each with 20 slots, instead of either 3 VCs each with 20 slots or 2 VCs one with 20 slots and the other with 40 slots). Moreover, for the same number of resources, SUR 3VC works much better than FA and FA bubble on both, network latency and throughput (9% better). Also, in transpose traffic, Figures 3.7c and 3.7d, SUR routing performs much better than FA and FA bubble. In this case, both versions of SUR achieve a boost in throughput of 20% when compared to FA. Also both SUR versions perform better on network flit latency.

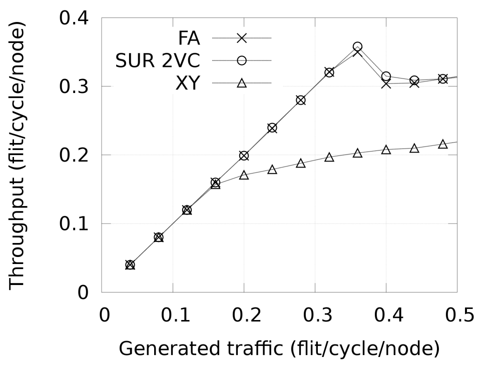
Next, we analyze the results for the torus scenario. In this case, differences are much more significant. Figures 3.7a and 3.7b show the performance for bit-reversal traffic. SUR 2VC improves network latency achieved by FA and FA bubble. This is achieved by using less buffer resources (2VCs each with 20 slots, instead of either 3 VCs each with 20 slots or 2 VCs one with 20 slots and the other with 40 slots). Moreover, for the same number of resources, SUR 3VC works much better than FA and FA bubble on both, network latency and throughput (9% better). Also, in transpose traffic, Figures 3.7c and 3.7d, SUR routing performs much better than FA and FA bubble. In this case, both versions of SUR achieve a boost in throughput of 20% when compared to FA. Also both SUR versions perform better on network flit latency.

Finally, improvements are also achieved in uniform and hotspot. Figures 3.7e and 3.7f present the performance comparison for uniform traffic. SUR 2VC and FA perform similar on latency and throughput. SUR 3VC has the best performance on network flit latency and throughput (14% better than FA). With hotspot traffic, Figures 3.7g and 3.7h illustrate as SUR 3VC works better than FA and FA bubble on network flit latency, and all routing algorithms have similar throughput.

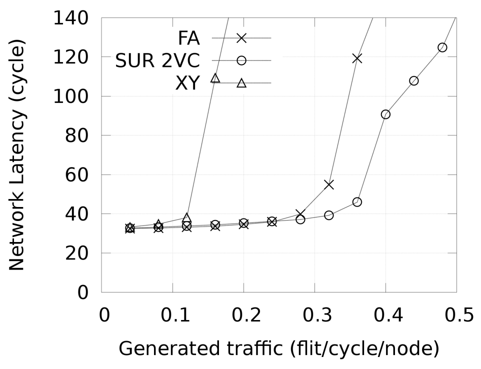
As we have seen in the results, SUR algorithm (together with the TBFC strategy) improves network throughput and latency over FA. In Figure 3.8 we show an example



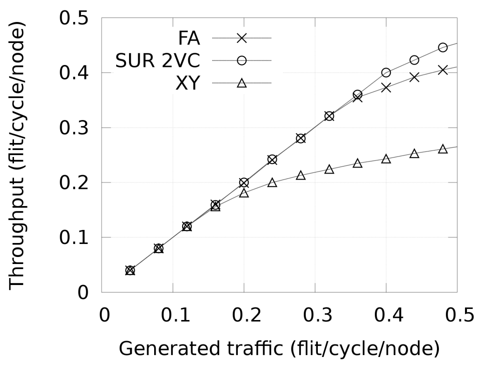
(A) latency bit reversal.



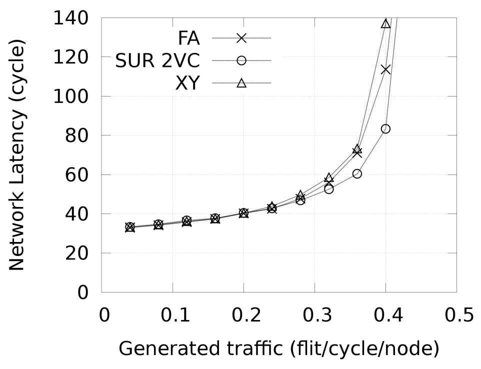
(B) throughput bit reversal.



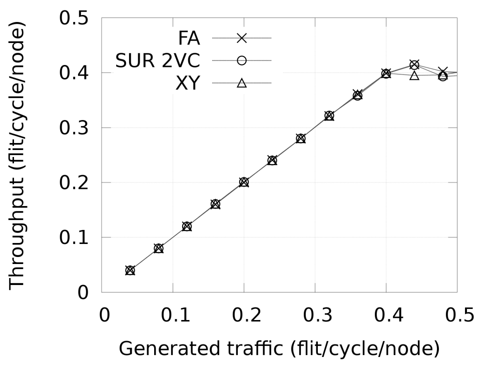
(C) latency transpose.



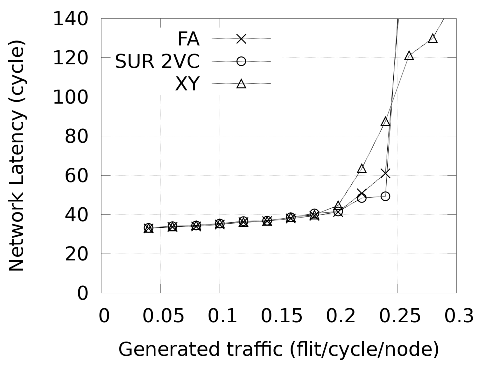
(D) throughput transpose.



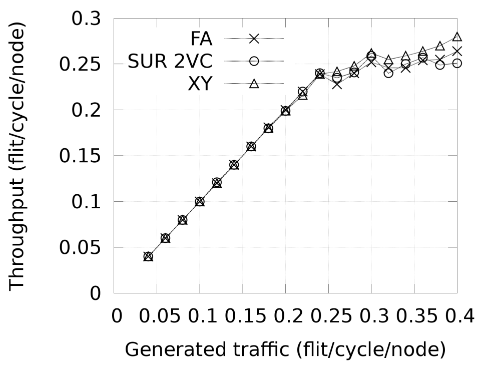
(E) latency uniform.



(F) throughput uniform.

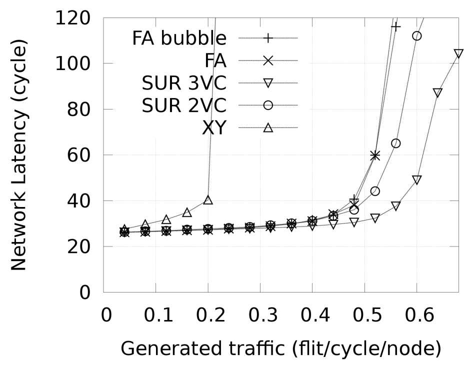


(G) latency hotspot.

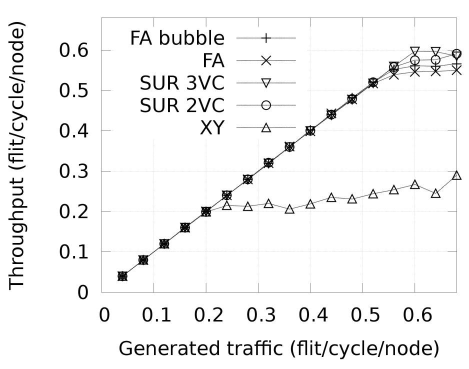


(H) throughput hotspot.

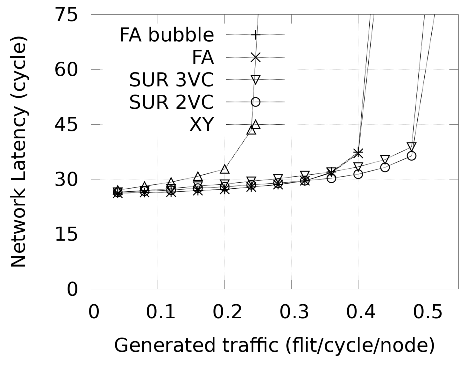
FIGURE 3.6: TBFC+SUR performance evaluation in 8×8 mesh networks.



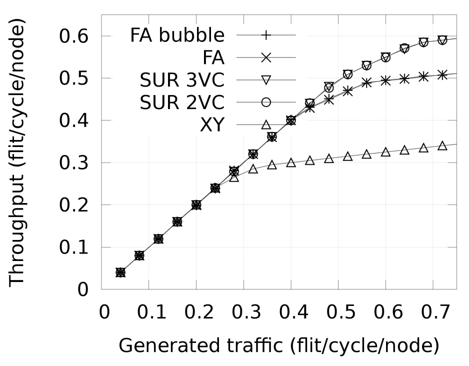
(A) latency bit reversal.



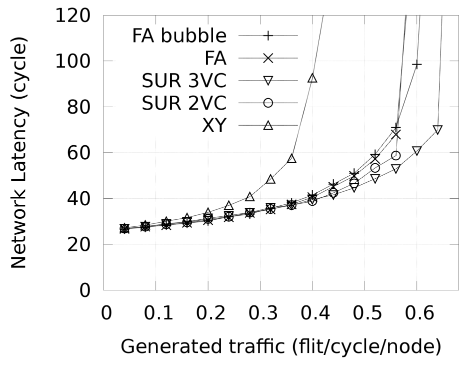
(B) throughput bit reversal.



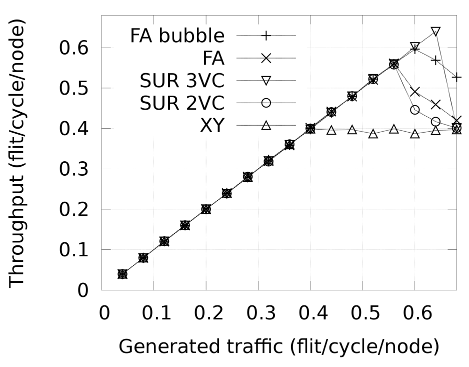
(C) latency transpose.



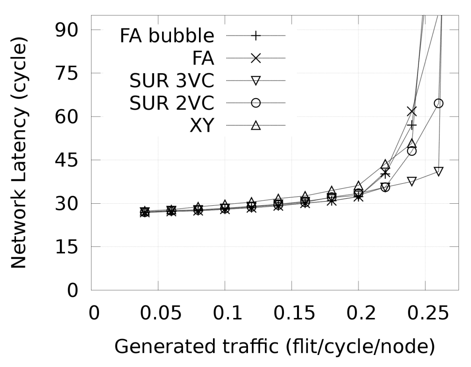
(D) throughput transpose.



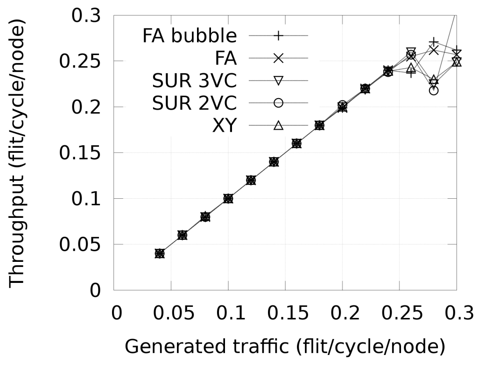
(E) latency uniform.



(F) throughput uniform.



(G) latency hotspot.



(H) throughput hotspot.

FIGURE 3.7: TBFC+SUR performance evaluation in 8×8 torus networks.

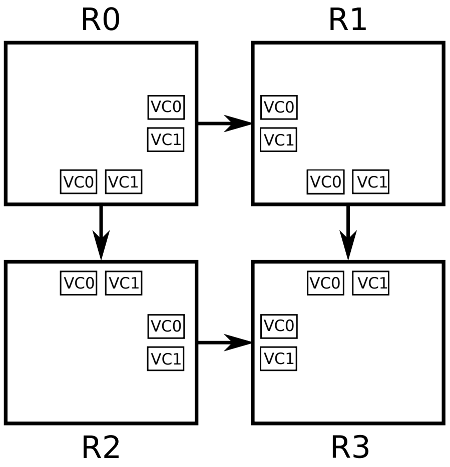


FIGURE 3.8: 2D Mesh Example.

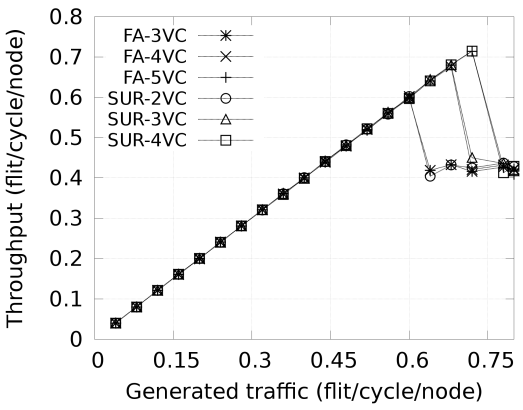


FIGURE 3.9: TBFC+SUR scalability

that highlights why we are achieving such improvement over FA. The Figure represents a 2x2 mesh. Assume that R0 wants to send a packet to R3. In FA, R0 can send the packet to R1 or R2. In case of R1 it can send the packet adaptively or through the escape channel (conforming to XY routing), and in case of R2 can send the packet only via the adaptive channel. Therefore, it can allocate this packet only in two VCs at R1 or in one VC in R2. The default fully routing algorithm (which promotes adaptive VCs over escape VCs) would then use only two possible VCs (one in each router). In case of an optimized FA algorithm (which gives the same priorities to adaptive and escape VCs), three VCs can be used (two in R1 and one in R2). However, in SUR algorithm safe packets can be allocated in any of the four VCs. Even unsafe packets can use any of the four VCs (taking into account there is an empty VC in the input port router). So SUR has more options to allocate the packet, allowing SUR to improve the performance obtained by FA.

Figure 3.9 shows how the benefits of TBFC+SUR scale with the number of VCs. TBFC+SUR with one VC less than FA achieves the same behaviour on throughput.

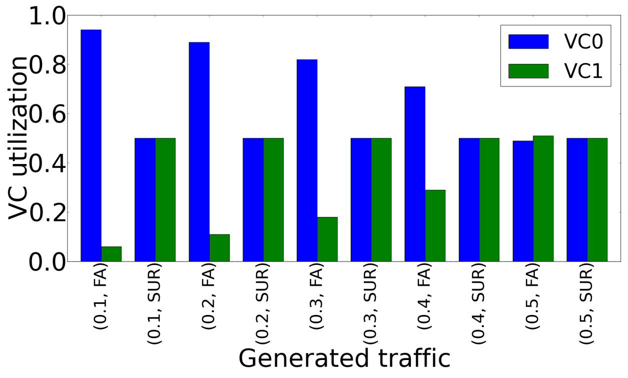
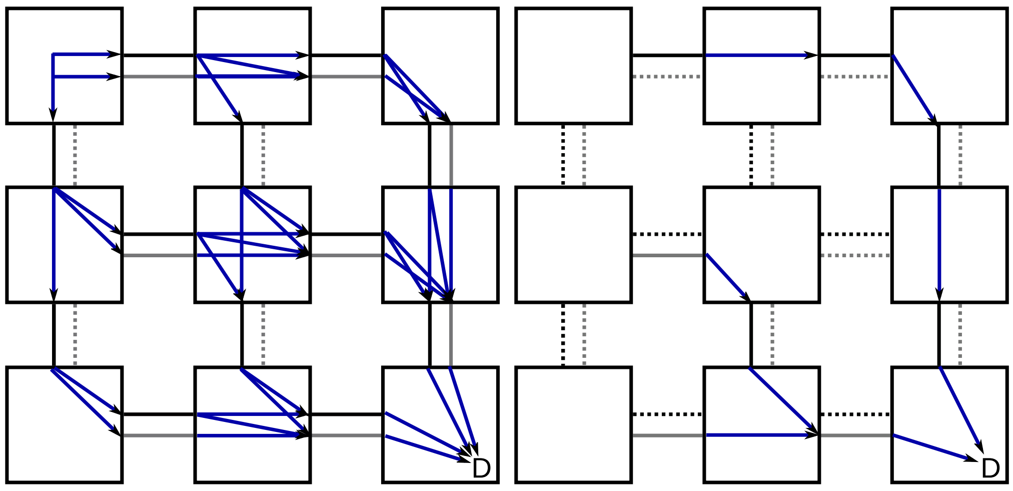


FIGURE 3.10: VC utilization.

As can be appreciated TBFC+SUR with 3VC achieves the same maximum throughput as FA with 4 VC, and the same for TBFC+SUR with 4 VC compared to FA 5VC.

Figure 3.10 show the VC utilization at the mesh scenario, with uniform traffic. SUR with low traffic achieves a balanced use of the resources. However FA use mainly the VC0, the adaptive channel.



(A) Fully adaptive. (B) Fully adaptive with EPC.

FIGURE 3.11: Example of congestion spread.

3.1.2 End Point Congestion Filter

In this section we describe the second contribution of this thesis, which aims to reduce the side problems created by congested scenarios, and focuses on improving performance of non-congested traffic (in terms of latency and throughput).

End-Point Congestion Filter, EPC, detects congestion formed at the end-points of the network, and prevents the congestion from spreading through the network. Basically, EPC disables adaptivity of congested packets.

The EPC filter works as follows. When the router has a packet (p_a) to forward, it checks whether the packet potentially contributes to a congestion situation. If the router recently forwarded a packet (p_b) with the same destination, then p_a's adaptivity is forbidden until p_b makes progress at the downstream router. Otherwise p_a is forwarded using the adaptiveness capability of the routing algorithm. The fact that p_b moves is a clear indication that packets towards that destination are not building a congestion situation. Thus, EPC enables again adaptivity for p_a. Notice that p_b, while its adaptiveness is disabled, may potentially take the same port used by p_a.

Fig. 3.11 shows an example of a 3×3 mesh with 2 VCs per port (a line represents a VC) and assuming fully adaptive routing algorithm with one adaptive VC (black color) and one escape VC (implemented with DOR routing, grey color). All nodes send traffic to node D (hotspot). Solid lines mean the VC is assigned to those packets. Dotted lines represent unused VCs. Figure 3.11a shows almost all VCs (adaptive and escape VCs) being used for the hotspot destination, thus, building a congestion situation. In our proposal, Figure 3.11b, the router only allows one VC to forward messages to hotspot destination per router. Thus, congestion is not spread and non-congested traffic can still advance. Notice that in the central router the south port is selected to forward traffic

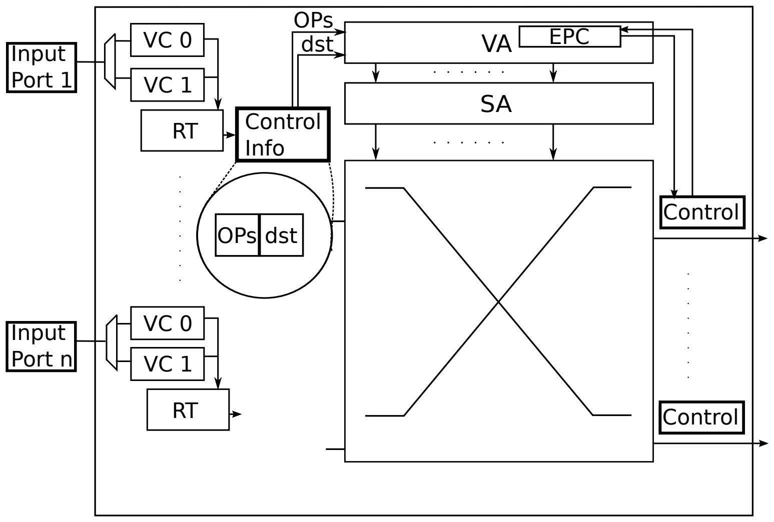
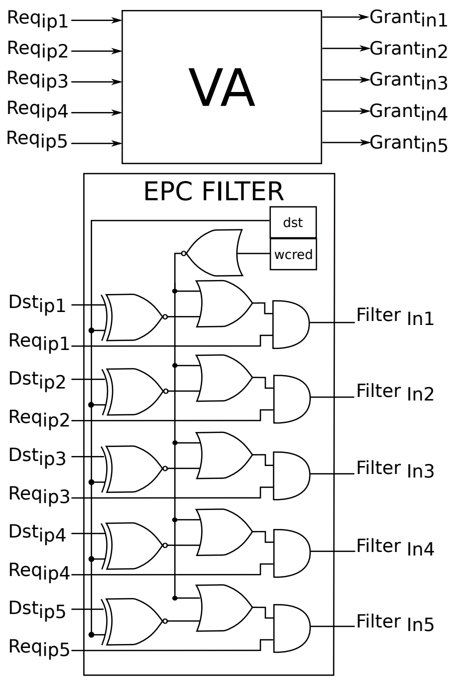


Figure 3.12: Baseline router architecture including EPC.

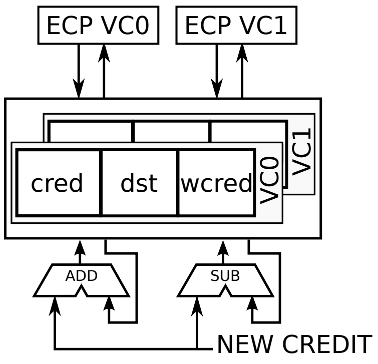
to D, then east port cannot be selected as an output until this traffic is completely forwarded.

Fig. 3.12 shows the pipelined router architecture with the EPC logic. In RT, the router keeps the requested outputs based on the fully adaptive routing algorithm [63]. Output port IDs (`OPs`) and destination ID (`dst`) are kept in the Control Info. In VA, the router allocates the resources (VC) to the requesting flits. In SA, the arbiter selects which input port is selected at each cycle to forward a flit through an output port. Both stages, VA and SA, are arbitrated following a round robin strategy (RR). A 3-phase arbiter (request, grant, accept) is implemented and VA and SA stages run in parallel. Output port selection is made on buffer occupancy. The crossbar is multiplexed (only one flit can access the crossbar at a time from each input port). Flit multiplexing is allowed (also known as wormhole flow control [64]). The router is implemented with credit-based flow control and Virtual Cut-Through (VCT) switching. Notice that the EPC mechanism can also be used in a WH network.

EPC is implemented in VA at each output port VC (Fig. 3.13a) and keeps the following info (Fig. 3.13b): the number of available credits (`cred`) located at the control information, the destination (`dst`) of last forwarded packet, and number of credits to wait in order to unlock the destination (`wcred`). When a header flit arrives to VA, it provides the `OPs` and the `dst` of the packet from routing control info. Then, EPC matches the `dst` with the one located at the output port control info with a non-zero `wcred` value. If there is no match, the flit accesses VA as usual. Otherwise, it will wait for the next cycle. To do so, the *Filter in_x* signals are sent back to the input ports and they disable the generation of accept signals in the third phase of the arbiter.



(A) EPC.



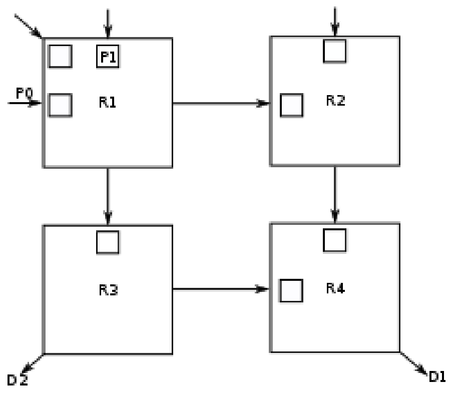
(B) Fully adaptive with EPC.

FIGURE 3.13: EPC and Output port control.

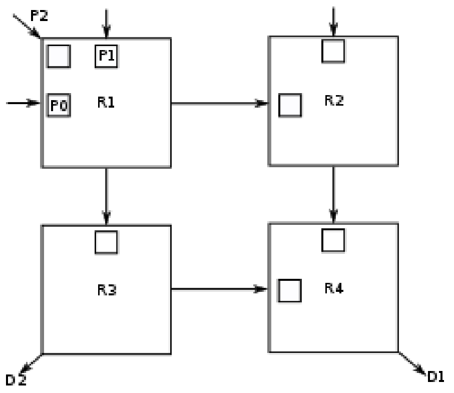
When the header flit wins one VC, the router sets the `dst` and `wcred` registers at the output port information accordingly. In `dst` the router sets the destination of the packet. The `wcred` field is updated as follows: `wcred=queue_size-cred+1`, where `queue_size` is the length of the queue in flits and `cred` is the number of credits available at the VC. This value guarantees that whenever the header leaves the downstream router the `wcred` register will reach the zero value, enabling again packet forwarding to that destination. When a credit is received, the router updates the fields, adding 1 to `cred` and subtracting 1 from `wcred`.

3.1.2.1 EPC Example

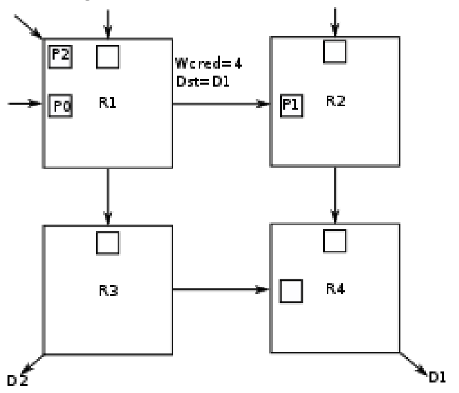
Fig. 3.14 shows an example of a 2×2 mesh assuming fully adaptive (FA) routing algorithm. At t_0, p_0 arrives to r_1 and p_1 is in the RT stage both packets have the same destination, d_1. At t_1, p_0 is in the RT stage at r_1 and p_1 competes for the outputs, then p_1 is forwarded. At t_2, p_1 has been forwarded and the filter is set with $W_{cred} = 1$ and $dst = d_1$. This means that p_0 cannot get any output port until r_1 receives 1 credit from r_2. Then p_0 is blocked because the filter disables adaptivity. At the same time, packet p_2 arrives to r_1 with destination d_2. At t_4, p_2 arrives to the VA/SA stage and wins the south output port. Finally, at t_5, p_2 is forwarded and r_1 sets the filter in the south output port with $W_{cred} = 1$ and $dst = d_2$. p_0 will be routed once the filter at the east output port is removed, meaning that p_1 is leaving router r_2 and, thus, experiencing no congestion.



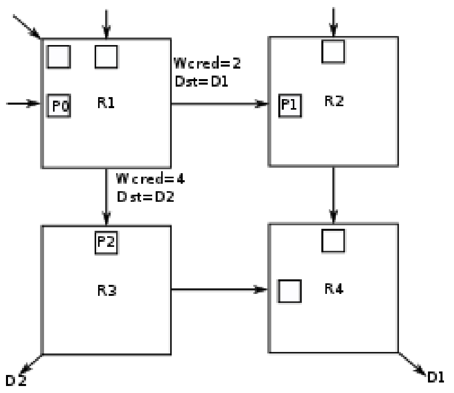
(A) t_0; p_0 arrives to r_1 and p_1 at RT stage with same destination d_1.



(B) t_1; p_0 in RT and p_1 at VASA stage in r_1.

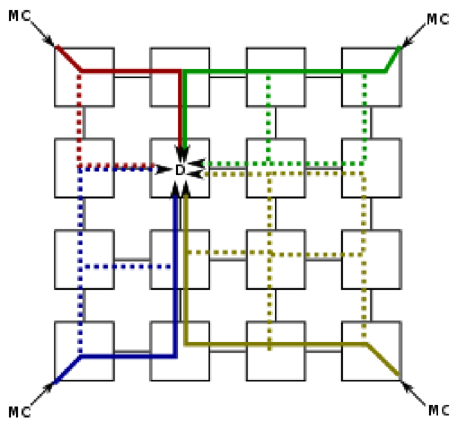


(C) t_2; p_0 blocked and p_2 arrives.

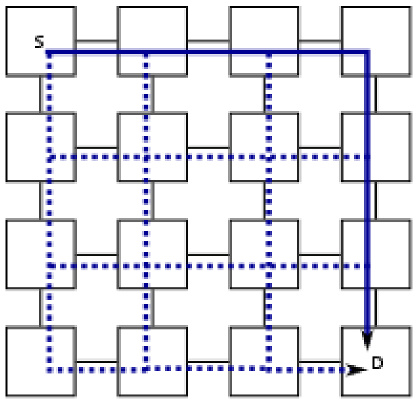


(D) t=5; P2 advances.

FIGURE 3.14: EPC filter walk-through example.



(A) cache refill.



(B) Burst traffic.

FIGURE 3.15: EPC application scenarios.

3.1.2.2 EPC Application Scenario Examples

Fig. 3.15a shows an scenario in which EPC can be useful. Node *D* requests a cache refill using fully adaptive routing. The node sends the requests to the four memory controllers (MCs), and then, the MCs send data to D. In this case, when XY paths get congested, the routing algorithm starts to use alternative paths spreading this way congestion over the network. With EPC, only XY paths get congested. Another useful case for EPC can be seen in Fig. 3.15b where bursty traffic from S to D is injected, potentially by a multimedia application.

3.1.2.3 Switching, VC, and Routing Impact

The previous conditions are set for a VCT router. For a WH router those conditions would slightly differ, being the *wcred* count dependent of message size instead of queue size. Indeed, in WH what is difficult to achieve is a clear detection of the congestion situation to disable adaptiveness. However, HOL blocking will be more pronounced due to the typical less buffering exercised.

On the other hand, the more VCs implemented the less HOL blocking will be seen, thus less congestion effect (making the baseline case perform better). However, if the same amount of traffic is sent to the same destination, regardless of the number of VCs, the same degree of congestion will be seen, thus, having the same impact on the network. Moreover, VCs are typically used to classify traffic, thus, congestion can occur in isolated traffic classes, at each VC. Thus, the effectiveness of EPC will be lower as network has more buffer capacity and congestion is less severe. But for a declared congestion situation EPC will help. Related to routing, EPC does not change the routing algorithm, and is orthogonal to the number of VCs. Therefore, is orthogonal to deadlock conditions, at network and protocol level.

EPC does not isolate congested packets in separate VCs. In the presence of several hotspots, and two packets addressed to different hotspots reach one router, both will be requesting the same resource (VCs) their previous counterparts (previous packets) were using, thus, those packets will not spread congestion. Indeed, VC strategies for congested packets are orthogonal to EPC.

3.1.2.4 EPC Overhead Comparison

EPC is compared against two congestion control techniques, ICARO [65] and RCA [49]. Both need more resources than EPC. ICARO needs an extra Virtual Network for bursty traffic and a Dedicated Signaling Network to notify the hotspot situation to end nodes. In addition, it needs at each node two vectors with length equal to the number of nodes and some logic to manage bursty traffic. RCA needs a low bandwidth monitoring network to propagate the congestion information. RCA routers need two extra modules per port for aggregating and propagating the congestion information. RCA needs also Congestion Value Registers (CVR). Notice that the notification network is not required in EPC, thus exhibiting lower area overheads. Note that EPC just prevents congestion ramification by the adaptive routing algorithm, thus being complementary to ICARO/RCA.

3.1.2.5 Performance Evaluation

This section presents an evaluation and analysis of the EPC filter, first describing the analysis tools and simulation parameters and then, analyzing the performance results for three configurations. In the first two, we use deterministic (XY) and fully adaptive (FA) routing algorithms, in both cases without EPC. FA uses two VCs (one for adaptive and one for escape paths[63]). In the third one, we use FA with EPC (FA-EPC).

3.1.2.5.1 Analysis Tools and Parameters

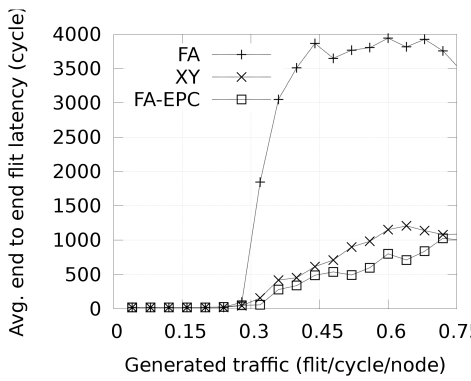
We model a 4×4 mesh with 2 VCs per port, virtual cut-through switching, 4-stage pipeline routers, and 16-byte message size, 4-byte flit size, and queues with 4 flit slots. XY, FA and FA with EPC (FA-EPC) is modelled.

Three scenarios are analyzed. The first one (UNIF) refers when a uniform random distribution is used and no congestion in the network is produced. The second one (C_LOW) refers when there is a small congestion spot in the network (background traffic). In particular, eight nodes (selected randomly) send traffic to node 11 with a 30% probability (the rest of traffic is uniformly distributed). The third one (C_HIGH) refers when the hotspot probability is increased to 70%.

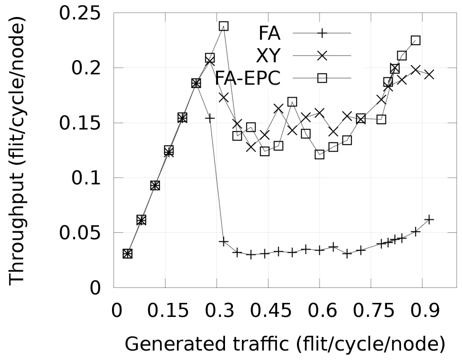
Results for two types of traffic are shown. The first one is for the uniform (foreground) traffic and the second one is for the hotspot (background) traffic. For this, the first 10000 packets generated after the stable network state has been reached (after initial 100000 packets reached destination) are labelled. We take into account only those packets at reception for statistics purposes. Doing this, the traffic distribution is kept the same both, at generation and at reception time, thus ensuring traffic distribution is not modified by the congestion situation.

3.1.2.5.2 Performance Results

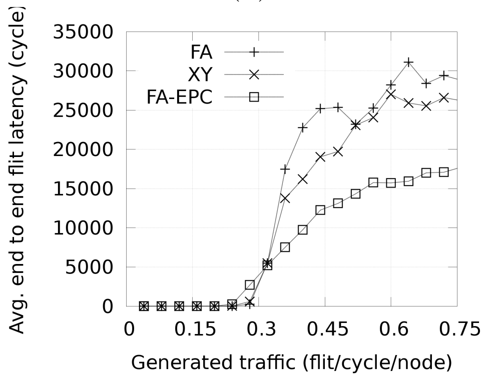
Figs. 3.16a and 3.16b show results for foreground (uniform) traffic in C_LOW scenario. End to end latency in FA-EPC is up to four times lower than the one achieved by FA. As seen, flit latency for FA has a sharp increase around 0.3 flit/cycle/node injection rate. This is when congestion affects the uniform traffic. Also, FA-EPC slightly improves the results achieved by XY. However, in FA-EPC and XY, latency increases linearly due to the increase of foreground traffic only. This difference in latency is produced because the hotspot spreads through the network when no EPC filter is used with FA. Regarding network throughput for uniform traffic (Fig. 3.16b), FA reaches saturation at 0.25 flits/cycle/node, whereas EPC reaches 0.32 flits/cycle/node injection rate. Notice that even XY behaves better than FA. For the hotspot traffic (Figs. 3.16c and 3.16d)



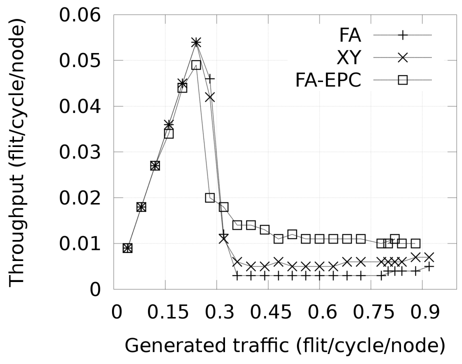
(A)



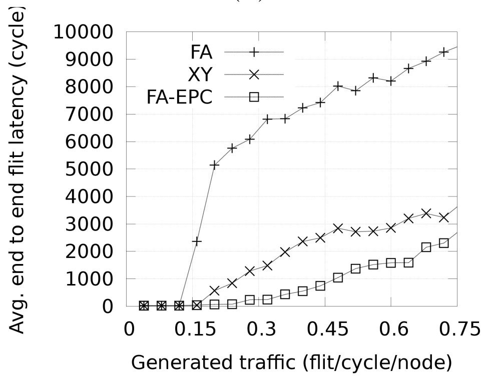
(B)



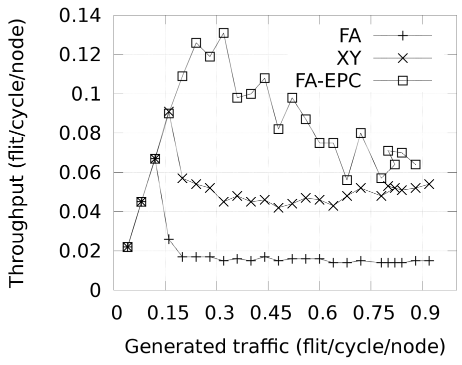
(C)



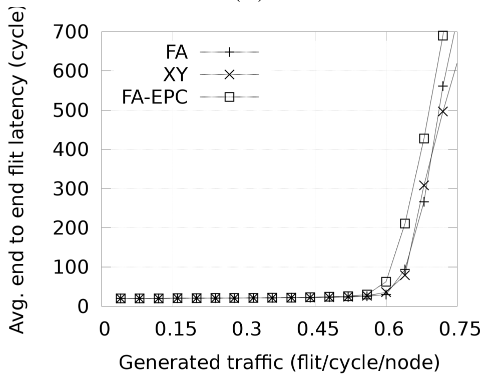
(D)



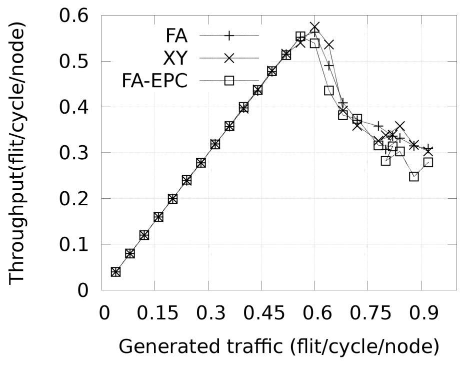
(E)



(F)



(G)



(H)

FIGURE 3.16: EPC results. C_LOW (abcd) C_HIGH (ab), UNIF (cd). Uniform traffic (abefgh), background traffic (cd)

FA-EPC keeps packet latency lower (50 percent lower) during the network congestion situation (beyond network saturation point).

The impact of EPC in C_HIGH is higher (Fig. 3.16e). Foreground end to end latency in FA becomes up to five times higher than the one achieved by FA-EPC. FA throughput (Fig. 3.16f) is doubled when using EPC. The hotspot traffic achieves a very similar behaviour. Finally, in UNIF, the EPC filter has about an 8% end to end latency overhead, as the filter without congestion situation may also block non-congested packets temporarily. Network throughput, however, is roughly the same for the different routings. This small overhead in uniform traffic suggests us to use the EPC filter in very specialized scenarios, and possibly dynamically.

One interesting comparison comes from Figures 3.16e and 3.16g. Comparing the latency of background traffic for C_HIGH and the packet latency for uniform traffic (without hotspot). As shown, packet latency is very similar for both cases. This means that congestion traffic effects are decoupled from background traffic.

3.2 Network Latency Reduction on CMP

3.2.1 Standard PROSA

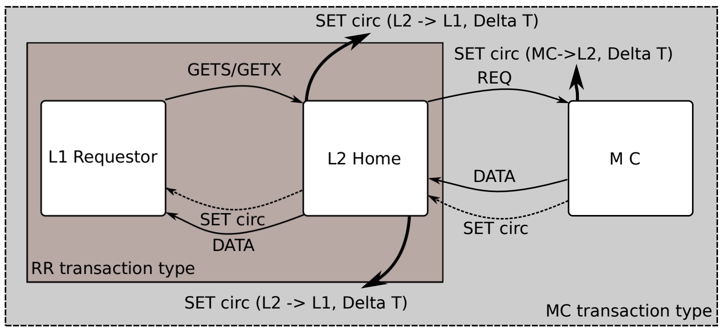
Now, we focus our attention on a different contribution with a different target. Now, we focus on latency reduction. The PROSA (PRotocol Oriented Switch Architecture) will be developed in phases. At each phase additional functionality will be provided, in order to achieve more efficiency and performance. Also, we will extend PROSA with complementary designs such as the memory latency estimator device or the inclusion of slack time to circuit setup logic. For the sake of understanding, we will start with the baseline design that focuses only on predictable coherence actions without slack time for setup circuits. We refer to this method as standard PROSA. Then, we will extend PROSA.

PROSA sets up circuits between the memory controller and the L2 banks (MC-L2) and between L2 banks and L1 caches (L2-L1) before they are needed. First, we show modifications performed in the coherence protocol and then describe modifications needed in the NoC (the PROSA controller and PROSA router) and MC (the Memory Latency Control Unit, MLCU).

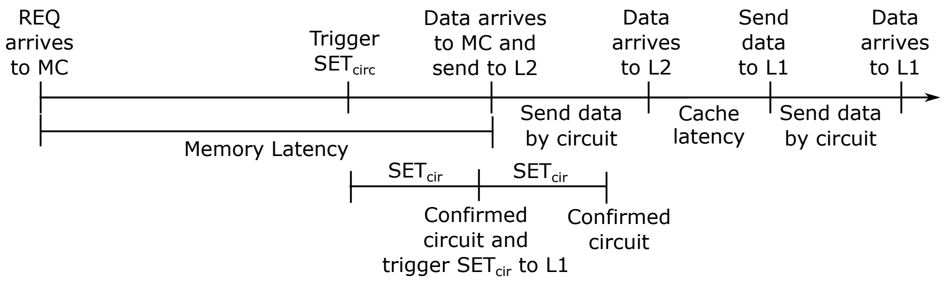
3.2.1.1 PROSA Coherence Protocol

In order to program circuits, we add a new protocol action termed SET_{CIRC}. This action involves the source and destination of the circuit, and the time period the circuit will be needed (Delta T). The action is triggered by the coherence protocol when MC and ReqResp transactions are issued (see Figure 3.17a). In MC transactions, whenever a request is received by the MC, a Memory Latency Control Unit (MLCU; described in Section 3.2.1.4) predicts when the data will arrive from main memory. Based on this delay, the MC triggers a SET_{CIRC} action that will setup a circuit after a delay period. The delay period equals to the predicted memory latency minus the circuit setup period, (CSP), which is the required time to setup a circuit between the MC and the destination node. The SET_{CIRC} action sets the circuit. By the time the block arrives to the MC the circuit has been set and is kept during the time the data will be transmitted. When SET_{CIRC} arrives to L2, the circuit is confirmed (acknowledged) but in parallel a new SET_{CIRC} command between L2 and L1 is triggered. Thus, when the data arrives to L2, after accessing the L2, the block is sent also to L1 using a circuit. These two circuits are predictable as they will be necessary regardless of network status. Figure 3.17b shows timings of both circuits.

In ReqResp transactions (between L1 cache and L2), whenever a GETS/GETX message is received, a SET_{CIRC} action is triggered between the L2 and the L1. Again, the



(A) Transaction.



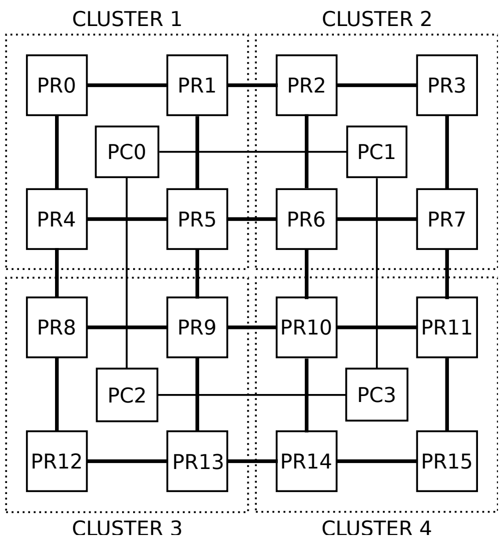
(B) Cronogram

FIGURE 3.17: PROSA new actions triggered by the coherence protocol and circuit establishment time line for MC transaction.

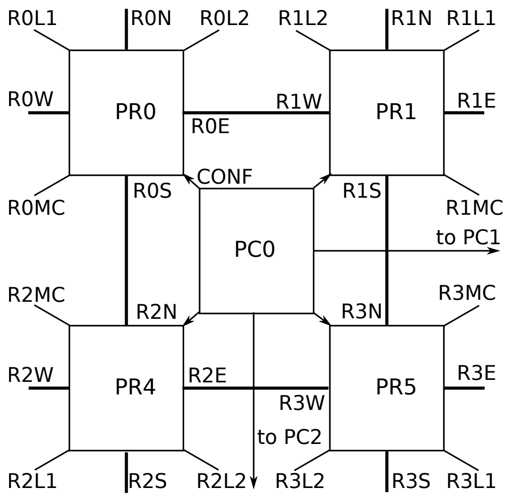
connection will be set only for the time period the data is available. However, contrary to the two previous triggered circuit scenarios (in MC transactions) now this circuit is speculative, in the sense the circuit may not be needed if the block is in P or I state. Therefore, for this type of circuit we will need to automatically and efficiently tear down the circuit when transmission is finished or when it is known the circuit is not needed. Notice that this modification does not introduce out of order delivery or duplicates, because for the same transaction the protocol never sends two messages to the same destination. Either the circuit is used for transmission of the message or the message uses the regular packet-switched (PS) network for transmission.

3.2.1.2 PROSA Circuit Network

PROSA follows a clustered approach where one PROSA Controller (PC) controls four routers (Figure 3.18a). When the SET_{CIRC} action is triggered in an L2 or MC, a request circuit (ReqCir) message is forwarded to the local PC. If the ReqCir wins the needed resources in the cluster, it advances to the next PC along the path. If not, a NACK response is sent to the source of the ReqCir message. When ReqCir reaches the PC that controls the destination and wins the resources, an ACK response is sent back to the source via the PC network. Routers are not affected by ReqCir messages, neither ACKs nor NACKs. All them travel via PCs.



(A) Prosa Network.



(B) Prosa Cluster

FIGURE 3.18: PROSA controller, detailed components.

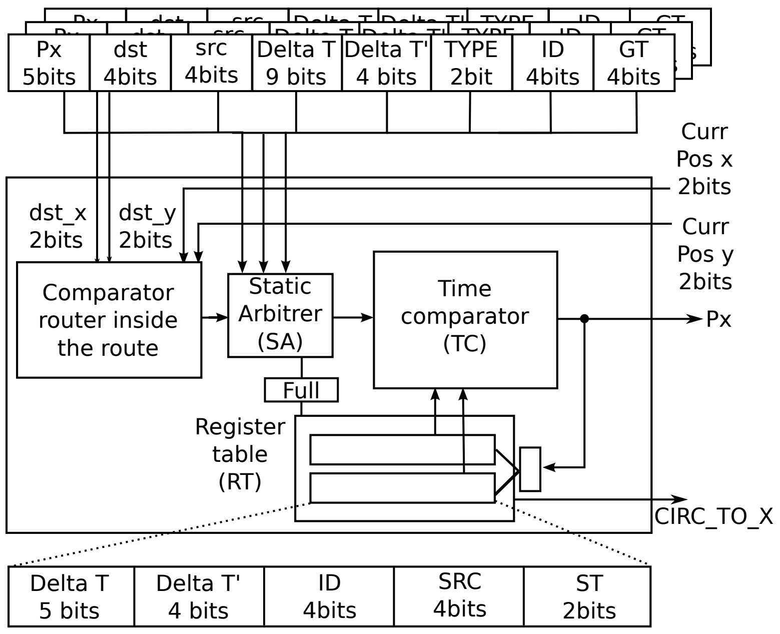


FIGURE 3.19: PROSA basic resource arbiter unit.

On top of Figure 3.19, we show the ReqCir message is composed by eight fields. *P* refers to the input port from which the ReqCir arrives to the cluster. *src* and *dst* are the source and destination for the circuit. *Delta T* and *Delta T'* carry the number of cycles to wait until the circuit will be established and torn down, respectively. *Type* carries the type of the ReqCir message (REQUEST, ACK, NACK). *ID* identifies the circuit inside the network, and finally, *Golden Token*, refers to the distance between *src* and *dst*. This field is used in the PC to assign priorities between requests.

Below the ReqCir structure, in Figure 3.19, we show the Resource Arbiter (ResArb) that controls whether a specific resource can be reserved for a particular period of time. A PC consists of several ResArb modules, one ResArb per output port in the cluster controlled by the PC. In total, each cluster has twenty eight ResArb, all shown in Figure 3.20 (e.g. R_{0S} corresponds to the south output port at the north-west router in the cluster). Each ResArb module arbiters between some requests, the number of requests depends on the number of inputs ports that have dependencies with the resource (the output port). This depends on the routing algorithm, in our case XY routing. In order to support other routing algorithms or topologies, the path comparator logic would need to be adapted together with the wiring connections between resource modules.

When an ReqCir message arrives to ResArb, the *dst* field is checked to decide whether the output port controlled by the ResArb is along the path between *src* and *dst*. If so, the ReqCir goes to the Static Arbiter which arbiters between the incoming requests at this point. The srbiter gives priority to requests with larger values in the *Golden Token* field (longer paths have priority). Then, ReqCir advances to the Time Comparator (TC) module where it checks that the request does not overlap in time with any previously programmed circuit. Finally, if there is no conflict with programmed circuits the ReqCir is stored in the Register Table and forwarded along the PC network. Notice that only one ReqCir message gets access to the table. As we will see, this does not impact performance and reduces complexity.

The register table keeps circuits information. For each circuit the following fields are stored: *Delta T*, *Delta T'*, *ID*, *src*, and *ST*. When an ordinary request wins the resource, *Delta T*, *Delta T'*, *ID*, *src* get the values from the ReqCir message. *ST* keeps the state of the circuit, which can be *unconfirmed*, *confirmed*, or *empty*. *Delta T* is decremented by one every cycle. When it reaches zero, the ResArb module activates the outputs *Cir_to_P* which control the circuit establishment in the PROSA router (described later). *Delta T'* is also decremented by one every cycle. When *Delta T'* arrives to zero the signals and the register are cleared.

When an ACK or a NACK arrives to ResArb, it follows the same path than an ordinary request, with some small differences. First, the ReqCir message advances to Static Arbiter. ACK/NACK messages have higher priority than ordinary requests. By construction we guarantee only one ACK/NACK enters one PC each cycle, thus they will always win the Static Arbiter access. Then, in Time Comparator the ReqCir message checks the information stored in the table. An ACK consolidates the stored information while a NACK removes it. Then, the ReqCir is forwarded (to next ResArb along the path inside the cluster or to the next PC).

The PC is shown in Fig. 3.20. It contains 28 ResArbs, 7 per router. For the sake of simplification, we group all outputs of a router (L1, L2, MC) into a single ResArb (R_{xL}), showing only twenty ResArbs. The PC implements two queues to store generated and incoming ACK/NACKs. ReqCir requests can be dropped. A demultiplexer located at

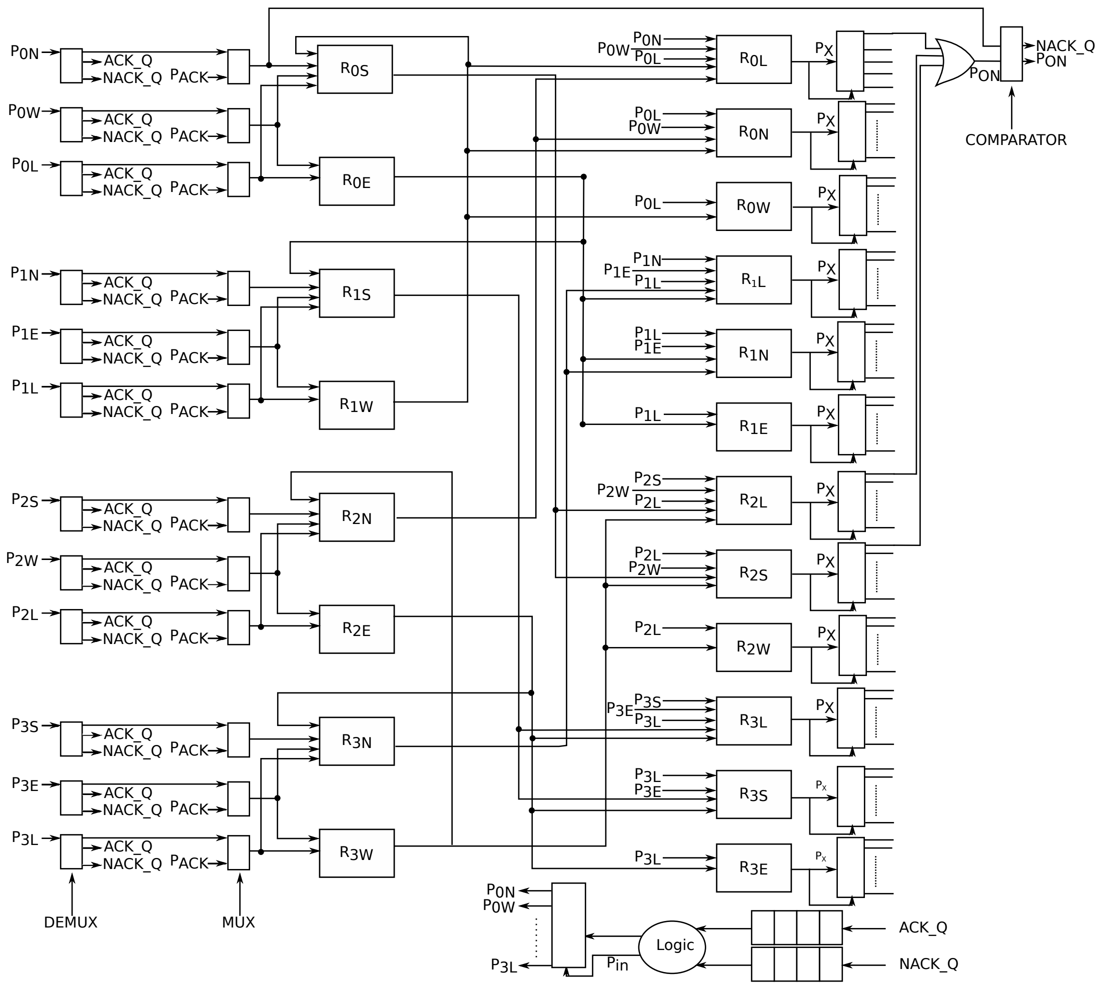


FIGURE 3.20: PROSA Controller.

the input port (left-hand side) divides requests by ACK/NACK messages (*type* field of the ReqCir message is used as selector). When a request arrives, it continues through the PC. However, if the incoming request is an ACK/NACK, it is sent to the corresponding ACK or NACK queue. The next stage in PC is a multiplexer, which multiplexes between incoming ReqCir message and queued ACK/NACKs, giving priority to ACK/NACKs. In case of conflict the request is discarded (generating a NACK which will be queued). Finally, the selected message enters the ResArb tree.

ResArb modules are linked following the resources dependencies imposed by the routing algorithm (XY in our case). An example of linked arbiters are $R_{0E} \rightarrow R_{1S} \rightarrow R_{3S}$ for messages coming through router PR0 and leading to a destination below router PR3. An ResArb module has several input and output dependencies as multiple routing combinations exist. Along this path, if a request wins all the required ResArb modules (from left to right) the request will be forwarded to the next PC, or will generate an ACK if the destination is controlled by the current PC. A comparator at the output of the PC module (right-hand side) checks whether one ReqCir message won all the required resources (it was at the input side and also succeeds at the output side). If not, a NACK message is triggered and stored in the NACK queue. When a ReqCir is sent to the next PC, the value of Delta T is decremented by 1. When this value reaches zero, the ReqCir is discarded by the PC.

Figure 3.20 shows the ACK and NACK queues and its control logic block. The logic guarantees only one ACK/NACK message will access the PC each cycle, ensuring that they will always win all RAs. NACK messages have higher priority than ACK messages. Moreover, the logic changes the input port of the ACK response, computing the input port of the ordinary request corresponding to this ACK response. Thus, ACK messages cross the same ResArb path used by the associated request messages, but in opposite direction. Notice that NACK generated messages locally in a PC need to be re-injected in the ResArb tree again to remove all the reserved resources, and then sent back to the previous PC.

3.2.1.3 PROSA Router

Figure 3.21 shows the modifications performed in the baseline router. We only add one demultiplexer per input port, one multiplexer per output port, connections between those elements and an asynchronous repeater per output port. The repeater allows to forward the flit very fast reducing wire delay, as is used in SMART [7]. This technology allows one flit to cross all the network in one cycle.

The router works as usual until signals *Circ_to_X* are activated, where X is the output port ($L1$, $L2$, MC, N, E, W, S). In this situation, the corresponding input and output ports are switched in and the arrived flits are blindly forwarded through them. At the same time, VA and SA arbiters associated to the output port are disabled. Notice that the circuits will not use buffer resources, therefore circuits cannot introduce deadlocks.

Circ_to_X signals are generated by the PC, each generated by a single ResArb module (the one that manages the output port). *Circ_to_X* indicates the input port that has to be switched in to the 'X' output port. One wire is used for each possible input port. Thus, the demultiplexer at an input port is selected from the ORing of all inputs bits on *Circ_to_X* signals associated to the input port (e.g Circ_to_1[1] or ... or Circ_to_n[1]), and the multiplexer at an output port is selected from the ORing of all the wires from the associated *Circ_to_X* signal. During the setup process, PROSA sets at each router the exact cycle where the ports need to be switched in. Programming those values is the most critical part to guarantee correct operation. Robustness is guaranteed by the correct implementation of the circuit setup process.

Circuits are cleared in a distributed and silent mode. When T and T' values reach zero on a resource arbiter (ResArb), the connection is eliminated as well. This is a valid point for addressing correctly miss speculation of circuits or when the data gets delayed more time than expected, for instance from the memory bank through the MC.

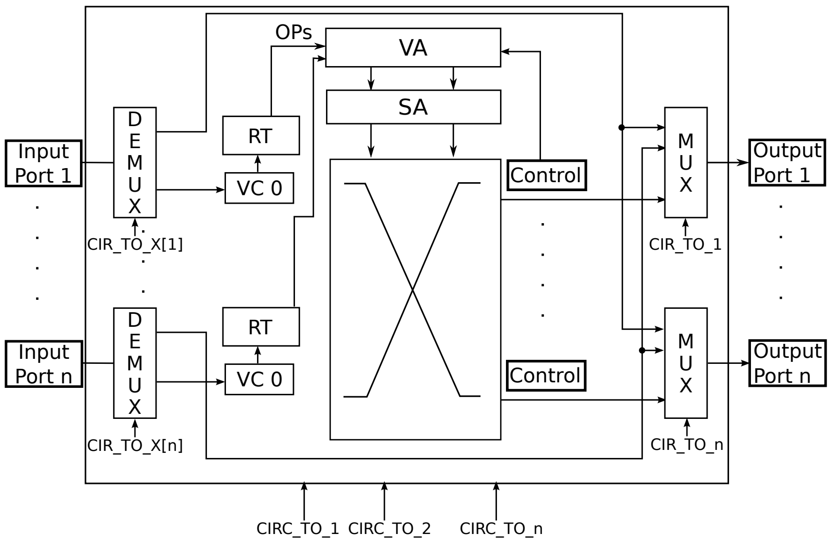


FIGURE 3.21: PROSA router.

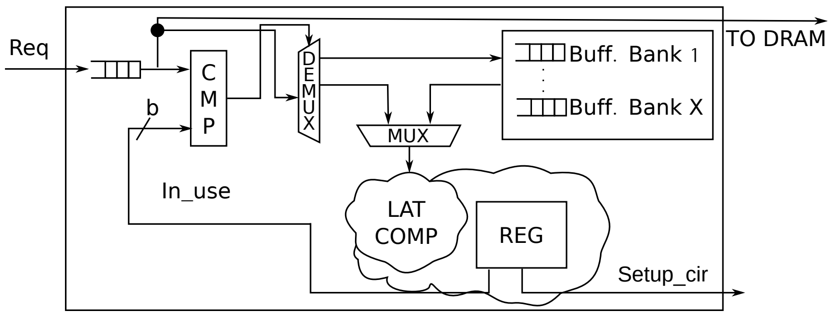


FIGURE 3.22: Memory Latency Predictor information.

3.2.1.4 Memory Latency Control Unit

Figure 3.22 shows the MLCU module that assists the PC at the MC. It predicts arrival time of memory blocks. MLCU works at bank level by monitoring its status (which row is open) and memory requests (which bank/row to access). The REG register keeps the information required per bank to correctly compute block arrivals: *ID* for the request identifier, *ROW* for the bank row where data is located, and T_{pred} and T_{end} define the time period when data arrives to MC. Each bank has also a buffer to store pending requests.

When a request arrives, the MLCU unit computes the bank associated. A comparator checks if the bank is idle (memory controller is not waiting for data from this bank). If idle, MLCU computes the block arrival time (LAT COMP logic block). The request ID is stored in REG register associated to the bank. Otherwise, the address, requestor ID, and required information to compute the memory latency are queued in the buffer associated to the bank. In both cases the request is sent to DRAM.

A circuit setup process is triggered whenever the expected arrival time for a requested stored in REG is equal to the maximum time required to cross the whole PROSA network, in our case is set to 16 cycles. When data arrives from DRAM the data is injected through the circuit (if set) or using packet switching. At the same time, if the buffer associated to the bank has a stored request, the MLCU dequeues the request and computes the new data arrival time from DRAM and stores the request in the REG register associated to the bank.

The MLCU computes to times T_{pred} and T_{end}. The T_{pred} is the time when the MLCU predicts the data arrival from the main memory, and T_{end} is the addition of the T_{pred} and the data length. To properly compute arrival times (T_{pred}), we need some timings from the memory, mainly the activation (Lat_{act}), precharge (Lat_{pre}) and read Lat_{read} latencies. The last T_{pred} and T_{end} values are stored in the register (associated to the bank). If the current row is open the next T_{pred} is set as the maximum between the last T_{end} and current time plus the read latency. Otherwise, T_{pred} is set to the maximum between the last T_{end} and current time plus activation, precharge and read latency. The next algorithm shows how the T_{pred} is computed.

```
if destination row is open then
    $T_{pred} = Lat_{read}$
else
    $T_{pred} = Lat_{act} + Lat_{pre} + Lat_{read}$
end if
if $T_{pred} < T_{end}$ then
    $T_{pred} = T_{end}$
end if
```

Whenever a new T_{pred} is computed, the MC schedules a SET_{CIRC} action. This action is scheduled sixteen cycles before the data arrives to MC, this is the maximum number of cycles that the PC needs to setup a new circuit with the farthest node. When the first flit from the incoming data arrives to the MC, the MLCU dequeues a request from the queue associated to the bank of the incoming data and computes the new T_{pred} value.

3.2.1.5 PROSA Circuit Setup Example

Figure 3.23 depicts an example of a successful PROSA setup circuit process between MC and L2, following a SET_{CIRC} action triggered by the coherence protocol. The circuit establishment process starts when the REQ message arrives to the MC. Let's assume that this event occurs at t_0 and all resources are available (no circuit conflicts will arise). Also, the MC knows the requested data will be available in 10 cycles from memory

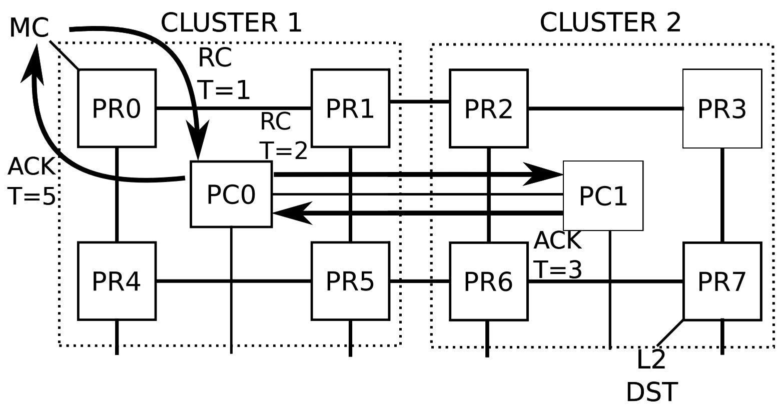


FIGURE 3.23: PROSA setup circuit example.

(suppose 10 cycles of memory latency to simplify the example). MC will process the request in one cycle, then, in t_1, will send a ReqCir message to PC_0.

At t_1, ReqCir reaches PC_0 and attempts to get all the necessary resources in the cluster along the MC-L2 path. The resources ReqCir will compete for are $R_{0E} \rightarrow R_{1E}$ (east port of PR_0 and PR_1). At each ResArb the request will be stored. Delta T will be set to 9 (Delta T at request) in the R_{0E}, R_{1E} ResArb modules. Delta T' will be set to the number of cycles needed for the transmission of the block.

At t_2, the ReqCir message is forwarded from PC_0 to PC_1. Notice stored values of Delta T are decreased by one. At PC_1, the same process described for PC_0 applies, but now for resources $R_{2E} \rightarrow R_{3S} \rightarrow R_{7L}$. After winning the resources, PC_1 generates an ACK message that will be stored in the ACK queue. Delta T fields at ResArb modules for R_{2E}, R_{3S} and R_{7L} are set to 8.

At t_3, all the resources with one circuit established (even in *unconfirmed* state) decrease Delta T value by 1. Also, at t_3, PC_1 processes the ACK message. As said above, ACKs and NACKs messages have higher priority, and then this ACK will win the necessary ResArb modules, thus confirming the circuit at PC_1 and sending back the ACK message to PC_0. Finally, at t_4, the ACK message arrives to PC_0, being processed at t_5 and winning all the ResArb modules and confirming the circuit at PC_0. The ACK message is forwarded to the MC, confirming the circuit at the NIC.

If one of the resources is not available during the circuit setup time, the affected PC will generate a NACK and will enqueue it into the NACK queue. Resources will be freed the next cycle and the NACK message will be transferred to the previous cluster. If a NACK is received at the NIC, the packet will be injected using packet switching (PS).

At time t_{11}, Delta T at ResArb module associated with R_{0E} reaches 0, thus the signal *Circ_to_W* (in R_{0E}) is activated and set to point to the local port. This signal switches in input port from MC and output port E at PR_0. This circuit will last the required number of cycles for the message (Delta T'). All other programmed output ports (R_{1E},

R_{2E}, R_{3S} and R_{7L}) router the input and output ports making the circuit just for Delta T' cycles. In this cycle MC receives the block from memory and injects the first flit. The flit is forwarded and crosses the entire network reaching $L2$ output port at $PR7$. When Delta T' reaches 0, the circuit is torn down in a distributed manner on each router just when the last flit of the message crossed the router.

3.2.2 PROSA Enhancements

As mentioned above, different design alternatives exist when dealing with CS. To analyze them, we extend the baseline PROSA design with four enhancements. First, all the protocol messages (including single-flit messages) will be considered for the setup and usage of circuits (we name this method as PROSA$_{all}$). The fact that circuits, when used, are much faster than the standard PS network suggests that a small penalty can be paid in the circuit setup process. In the second enhancement, we will allow a small slack to the circuit setup process (we name this method as PROSA$_{slack}$). As ACK messages may contend in the PC some delays may be incurred. A single cycle delay will ruin the circuit as the message will not use it. Therefore, by adding a small slack more circuits will be effectively used. The third enhancement refers to the use of circuits only for messages with destinations located closer than a given threshold. The larger the circuit the larger the penalty in setting up the circuit. Different thresholds will be analyzed (we name this method as PROSA$_{dd}$; distance driven). Finally, we will enhance baseline PROSA with different priority schemes in the PCs when dealing with ReqCir, ACK and NACK messages. This will affect the success rate of circuits being used. This method will be named PROSA$_{priorities}$.

3.2.2.1 Coherence Protocol Extension

First enhancement is the use of circuits for all protocol messages. Figure 3.24 shows the new diagrams for the PROSA$_{all}$ protocol. All events (control and data messages) request a setup circuit process by triggering the SET_{CIRC} action. New requests are plotted in red. In ReqResp transactions, the L1 cache requests a circuit between L1 and L2 to send a GETS/GETX message. If the circuit is confirmed then the GETS/GETX message is sent using CS. Otherwise, the message is sent using PS. When the GETS/GETX message arrives to the L2 cache it follows the baseline PROSA behaviour. A speculative circuit is setup between L2 and L1.

In MC transactions the L2 Home cache requests a circuit between L2 and MC to send the REQ message. If the circuit is confirmed the message is sent through the circuit, otherwise is sent using PS. Upon arrival, the MC sets a circuit for the incoming data to be delivered to the L2.

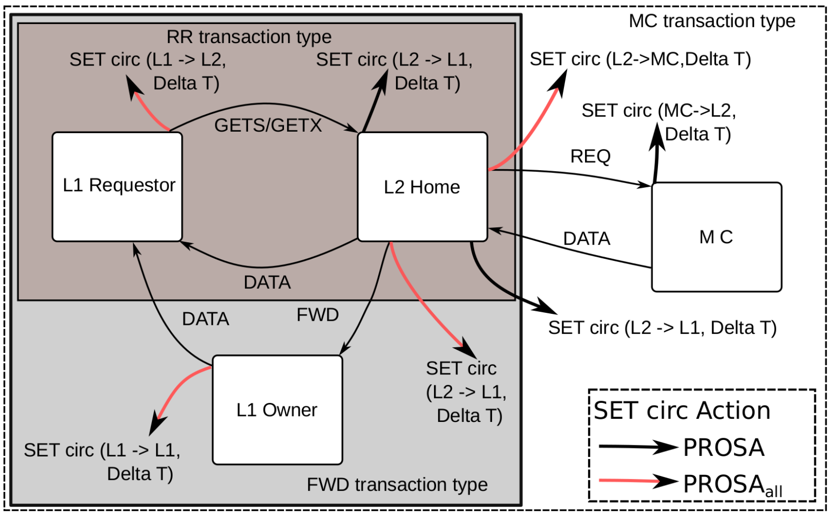

MC transaction type
RR transaction type
SET circ (L1 -> L2, Delta T)
SET circ (L2 -> L1, Delta T)
SET circ (L2->MC,Delta T)
SET circ (MC->L2, Delta T)
GETS/GETX
REQ
L1 Requestor
L2 Home
M C
DATA
DATA
SET circ (L2 -> L1, Delta T)
DATA
FWD
L1 Owner
SET circ (L2 -> L1, Delta T)
SET circ (L1 -> L1, Delta T)
FWD transaction type
SET circ Action
PROSA
PROSA_all

FIGURE 3.24: Transaction diagram for PROSA$_{all}$ coherence protocol.

Finally, in FWD transactions, when a load or store misses in L1, a new circuit is requested between L1 and L2 trying to setup a new circuit for the GETS/GETX message (as is the case for ReqResp transactions). When the request reaches the L2 and the block is in P state, a new circuit is requested between the L2 and the L1 owner to send the FWD message. When the message reaches the L1 Owner, a final circuit is set to send the block to the original L1 requestor.

Every SET_{CIRC} action is classified either as speculative or predictable. Predictable actions occur between the L1 and the L2 cache, between the L1 owner and the L1 requestor, between the L2 cache and the MC controller and between the L2 and the L1 owner (FWD transaction). Speculative actions occur when a GETX/GETS reaches the L2 cache, triggering a request from L2 to L1 requestor.

3.2.2.2 Slack on Circuit Setup Process

Figure 3.25 shows the timings of a message injected through PS and through PROSA. The case refers to an speculative circuit in ReqResp transactions. As we can see, when using PROSA, upon arrival of a message to the network interface a new circuit setup process is triggered. During this process the message is processed and the cache is accessed. Notice, however, that the confirmation message (ACK) may be delayed some cycles as it may encounter contention within the PCs. Therefore, before the circuit is confirmed the data is ready to be injected through the network. With standard PROSA, the circuit would fail and the message would then be injected through the network in PS mode. However, now in PROSA$_{slack}$, a slack of time is allowed to avoid this problem. Thus, the NIC will wait more time for the confirmation of the circuit. Once the confirmation arrives the message is injected through the network in CS mode.

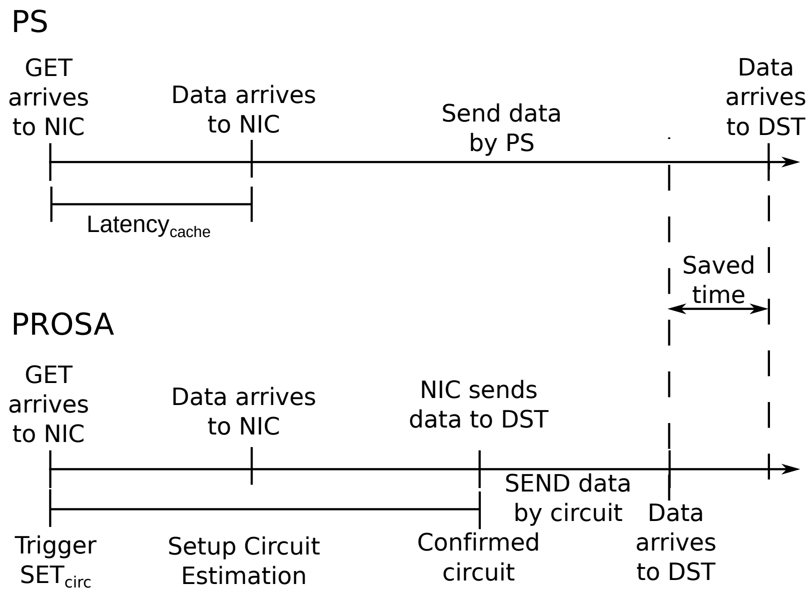


FIGURE 3.25: PROSA$_{all}$ and PS timings.

Notice that depending on the slack and on the relative speed of the circuit compared to the network in PS mode, PROSA$_{slack}$ may still deliver the message faster.

We define a slack constant ($Slack_{cir}$) which is assumed by the PC and by the NIC injector. Notice that by using this slack the number of circuits successfully used increases, potentially increasing performance. Also, notice that the confirmation of a circuit may be received before the slack expires (if no contention is encountered by the ACK message). In that case, the message is injected using the circuit as early as the confirmation arrives and the slack does not expire. Notice that the circuit will be programmed in the network for the transmission time plus the slack constant.

The latency on a circuit setup process is determined by the distance to destination and the slack constant:

$$Setup_{latency} = MAX(cache_{latency}, d * 2 + Slack_{cir})$$

where $cache_{latency}$ is the cache access latency, d is the distance in hops in PROSA (number of PCs crossed) and $Slack_{cir}$ is a small delay due to the contention in the PROSA network. Notice that this formula can get $Setup_{latency}$ smaller than $cache_{latency}$. As an example, if d equals 0 when source and destination are in the same PC and $Slack$ is smaller than $cache_{latency}$, in this case, the $Setup_{latency}$ is set equal to $cache_{latency}$.

When a successful circuit is configured, PROSA$_{slack}$ guarantees the circuit transmission time ($Setup_{latency}$ plus $CS_{latency}$) is lower than the message transmission time ($cache_{latency}$ plus $PS_{latency}$).

3.2.2.3 Distance Driven Setup Circuit

As a further enhancement, PROSA$_{dd}$ allows selective configuration of the circuits based on the distance to destination. Thus, when a circuit is requested to be configured, PROSA$_{dd}$ analyzes the $Setup_{latency}$ which is mainly determined by the number of PC hops. A $MaxHops$ parameter can be configured in PROSA$_{dd}$. If the distance is larger than $MaxHops$, then the circuit is cancelled and the data is forwarded through the network using PS. Later we analyze the behavior of PROSA$_{dd}$ with different thresholds.

3.2.2.4 PROSA Messages Priorities

As a final enhancement, PROSA$_{priorities}$ enables the use of different priorities between ReqCir, ACK and NACK messages within the PC network. In the standard PROSA, NACK messages get higher priority than ACK messages, and ACK messages get higher priority than ReqCir messages ($NACK > ACK > ReqCir$). This priority scheme guarantees all NACK messages get delivered to end points. However, many circuits cannot be set due to a conflict with NACK or ACK messages. Remind that ReqCir messages convert to NACK messages if they conflict with a NACK or ACK message.

Now, in PROSA$_{priorities}$, the endpoints will inject the message through the network in PS mode when the $Setup_{latency}$ expires. Also, PCs will remove circuits automatically. Thus, there is no need to guarantee all NACK messages are delivered to the end points. Indeed, NACK messages could be simply removed from the network. The only benefit they produce is that a message can be injected earlier into the network once it receives the NACK message (before the $Setup_{latency}$ expires).

Thus, in PROSA$_{priorities}$ we can use two different priority schemes. In the first one, ACK messages get now highest priority, so to speed up injection of messages through circuits, then ReqCir messages get higher priority than NACKs, which have the lowest priority ($ACK > ReqCir > NACKS$). The second one is more radical as NACK messages will simply be removed from the system. They will not be generated by PCs. In this scheme, ACK messages get higher priority than ReqCir messages ($ACK > ReqCir$).

Notice that removing the NACK support we ease the design of the PC. Figure 3.26 shows the new PC structure. In particular, the comparator column and the NACK queue structure and associated logic have been removed. Also, the TYPE field is reduced to only two types (one bit encoding). Also the logic at RCs is greatly simplified as partial circuit elimination is not longer needed when NACKs do not exist. This means the area and power overheads of PROSA will be reduced (will be presented later).

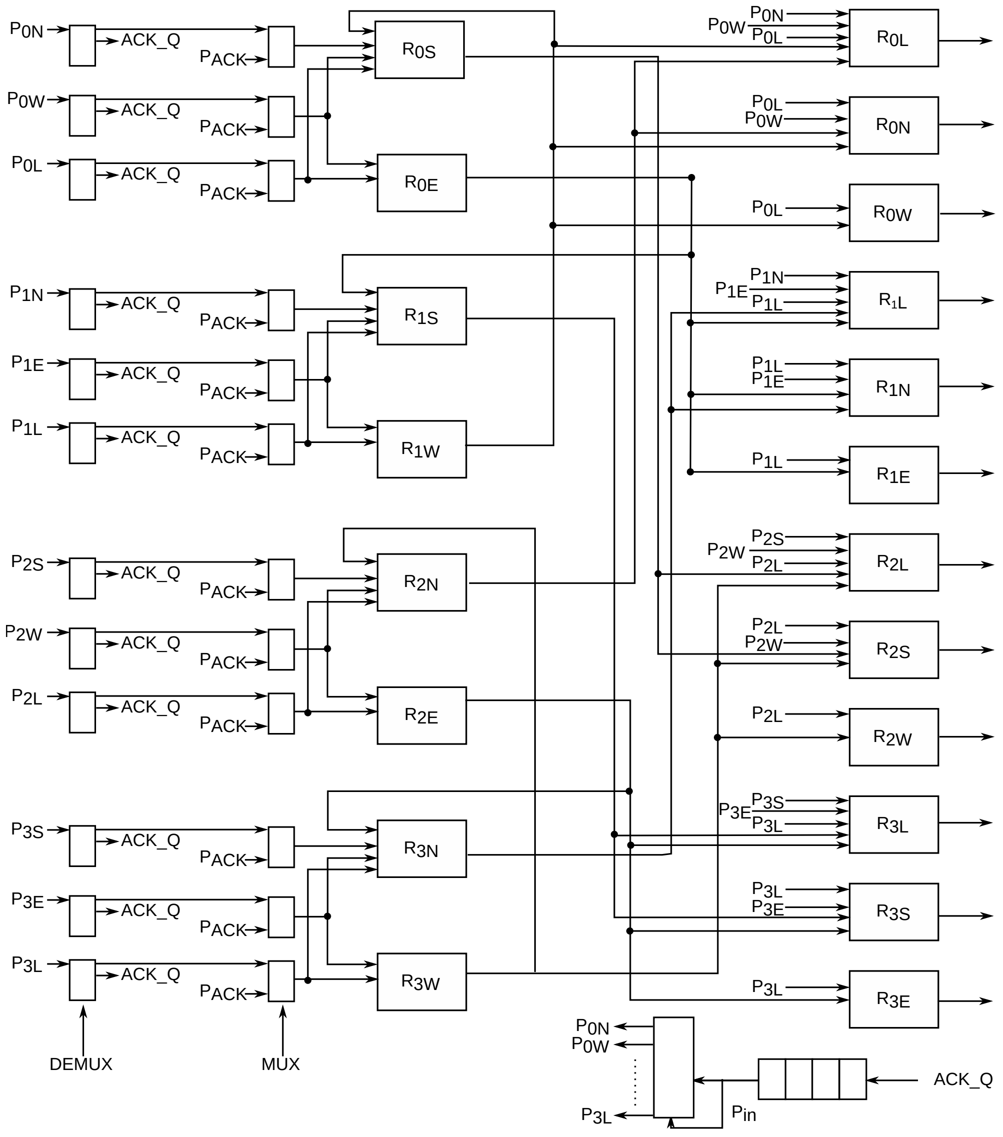


FIGURE 3.26: PROSA Controller without NACKs.

3.2.3 PROSA Evaluation

For the performance analysis we use gNoCsim [66], an event-driven cycle-accurate simulator that models any network topology and router architecture. We model a 2-stage pipelined router with VCs and flit-level crossbar switching, as used in Garnet [67]. Table 3.3 shows the simulation parameters for the router and the cache hierarchy. L1 is private to the core and L2 is shared but distributed among all tiles.

In a first analysis, we evaluate three mechanisms: the baseline router (BASELINE), the Dejavu (DEJAVU) solution [14] and PROSA (no optimizations involved). In PROSA data sent through circuits take one cycle (using SMART). We analyze applications from SPLASH [68] and PARSEC [69]. Table 3.4 shows the applications with their observed loads.

Parameter	Network	L1	L2 per tile	MC
Topology	8x8 mesh			
# VCs/fly link	4/1 cycle			
Message sizes	8/72 bytes			
Flit/Queue size	64 bits/9 flits			
sets/way/line size (B)		32/4/64	128/16/64	
cache/tag latency		2/1	4/2	
# MCs				2
Memory controller delays (open/activate/read rows)				16/16/16

Table 3.3: Parameters and values used for routers and caches.

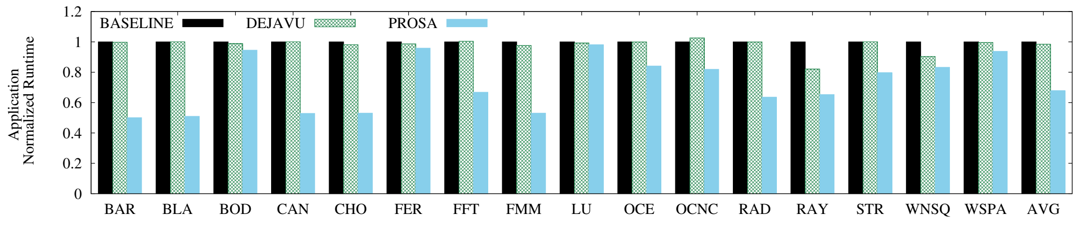
Application	Benchmark	Abbr.	Runtime	L1 miss	L2 miss
BARNES	SPLASH	BAR	High	Low	Med
BLACKSCHOLES	PARSEC	BLA	High	High	High
BODYTRACK	PARSEC	BOD	Med	Low	Low
CANNEAL	PARSEC	CAN	High	High	High
CHOLESKY	SPLASH	CHO	Med	Low	High
FERRET	PARSEC	FER	Low	Low	High
FFT	SPLASH	FFT	High	Low	Med
FMM	SPLASH	FMM	Med	Med	High
LU	SPLASH	CHO	Low	Low	Low
OCEAN	SPLASH	OCE	Low	Med	High
OCEANNC	SPLASH	OCNC	Low	High	Low
RADIX	SPLASH	RAD	Med	High	High
RAYTRACE	SPLASH	RAY	Med	Med	Med
STREAMCLUSTER	PARSEC	STR	High	Low	High
WATERNSQ	SPLASH	WNSQ	Med	Med	High
WATERSPACIAL	SPLASH	WSPA	Med	Med	High

Table 3.4: Applications tested with observed runtime and L1/L2 miss.

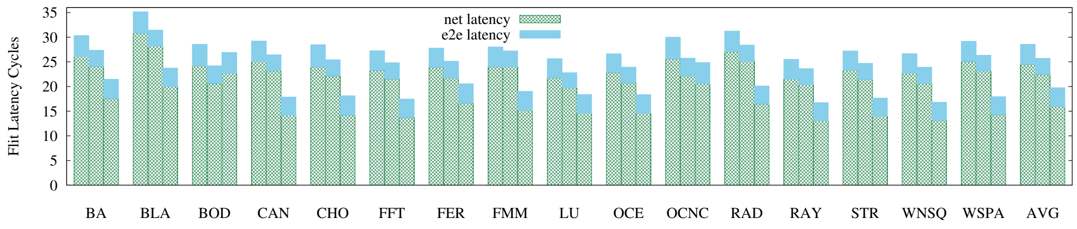
3.2.3.1 Results

Figure 3.27a shows application runtime. PROSA reduces, on average, BASELINE application runtime by 33.11%, reaching lower gains in applications with a low number of L2 misses (BODYTRACK), or with short application runtimes (LU) where the reduction in runtime is 5.45% and 4.0%, respectively. However, with balanced applications (OCEANNC), or applications with a high number of MC requests (FFT, CANNEAL, FMM, BLACKSCHOLES) PROSA reaches an improvement up to 50%. Deja Vu achieves negligible benefits in performance, as seen in [14].

Figure 3.27b shows latency results. DEJAVU outperforms BASELINE by 10% on average. PROSA, in applications with a low number of MC requests and small datasets (e.g. BODYTRACK), achieves a 7% improvement on end-to-end latency. However, PROSA outperforms BASELINE up to 39% on applications with a high number of L2



(A) Application Runtime



(B) Flit Latency

Figure 3.27: Performance results for different architectures (BASELINE, DEJAVU, PROSA in column order).

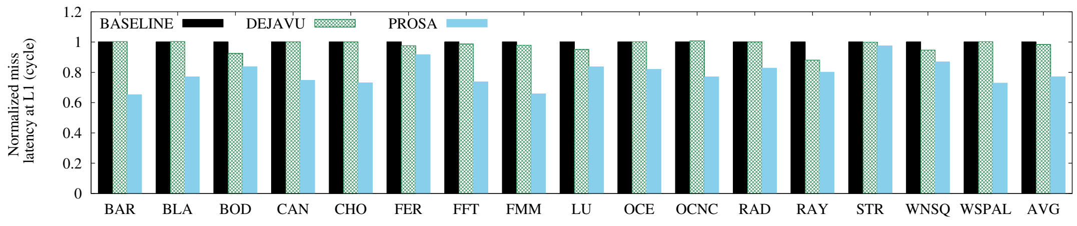


Figure 3.28: Memory latency results for different architectures. Normalized to baseline case (BASELINE, DEJAVU, PROSA in column order).

misses (e.g. CANNEAL). On average, PROSA outperforms BASELINE end-to-end latency by 31.14%. For network latency, blue solid color, on average, PROSA outperforms BASELINE by 34.16%. Notice PROSA outperforms significantly DEJAVU.

For L1 miss latency normalized to BASELINE (Figure 3.28), as it occurs with runtime, DEJAVU slightly improves BASELINE and PROSA achieves better performance. On average, our proposal outperform BASELINE by 23%.

	REC.ACK	REC. NACK
MIN	(OCNC) 85.98%	(LU) 4.28%
AVG	90,62%	9,38%
MAX	(LU) 95.72%	(OCNC)14.02%

Table 3.5: Number of ACK and NACK messages generated in PROSA in PC module.

Table 3.5 shows statistics about PROSA circuits. The first column shows the received ACKs (the number of circuits established successfully). In all cases this is higher than 85.58% and the average is 90.62%. The second column shows the received NACKs and is the sum of the next five columns of Table 3.6, which list the different types of conflicts

CONF. INPUT			CONF. RA		
	NACK	ACK	FB	TMP	ARB
MIN	(LU) 0.01%	(BOD) 0.63%	0%	(WSPA) 1.92%	(WSPA) 0.31%
AVG	0,04%	3,94%	0,00%	4,61%	0,79%
MAX	(WSPA) 0.18%	(WSPA) 9.27%	0%	(OCNC) 9.04%	(OCNC) 1.70%

TABLE 3.6: Number of conflicts generated in PC module.

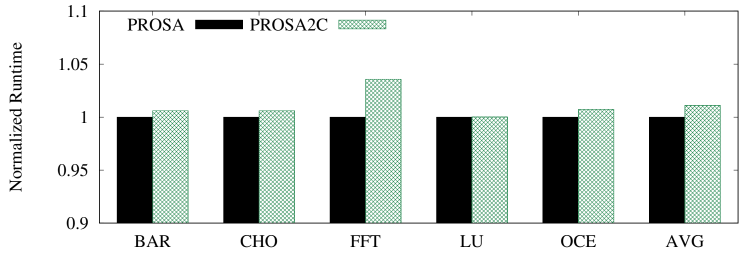


FIGURE 3.29: Results for PROSA with delay circuits equal to 2.

in the PC. The first and second column show the number of NACKs generated at the input of the PC as a normal request and an ACK/NACK conflicted at the same time. The last columns show conflicts generated at the ResArb arbiters due to (1) the register table being full (FB), (2) the required period of time of one request overlaps with one established circuit (TMP), and (3) two requests collide in the same ResArb module (ARB) due to the static arbiter. As we can see, most conflicts are generated because of temporal conflicts in generating circuits or because of concurrent requests conflict with an ACK in the same resource. However, the current table size at ResArb modules (2 entries) seems to be properly sized (even could be reduced thus saving more area) as the number of conflicts due to the table being full is negligible.

One critical aspect of PROSA is the technological ability to transmit via circuit one flit from one network corner to the opposite. To analyze the impact of a more relaxed circuit design, 2-cycle circuits have been tested (PROSA2C). Figure 3.29 shows, for some applications, the runtime application results assuming two 2-cycle circuits (for every source-destination pair). As can be seen, PROSA2C reaches similar results as PROSA, on average worsening performance by 1.2%.

3.2.3.2 PROSA Implementation

We have implemented all the PROSA infrastructure for a 4×4 CMP system. Each PC module has been implemented in Verilog and tested. We use a canonical router design with 64-bit flits, four 9-flit depth VCs, and with seven ports to attach 2D-mesh ports and L1, L2, and MC (including the MLCU). We use Design Vision tool from Synopsys with 45nm Nangate open cell library [70]. Power results are obtained from Orion-3 power library [71].

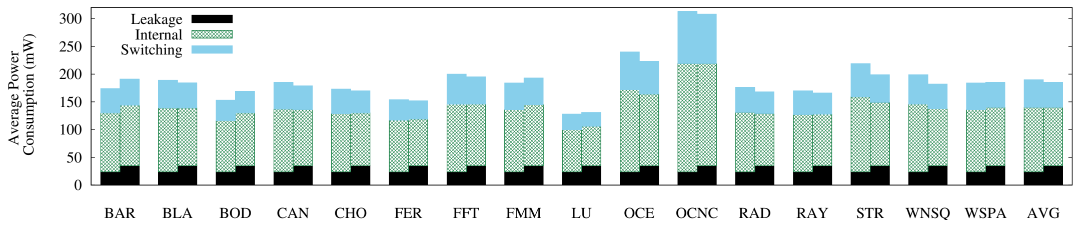


FIGURE 3.30: Power consumption results. First column represents BASELINE and second PROSA.

Table 3.7 shows the area overheads of PROSA for different configurations. In all of them, and for the sake of comparison, we also account for the components to build a cluster of four routers. In the case of PROSA we consider all the components (including the PC and the four PRs) and the logic and resources used for NACK messages.

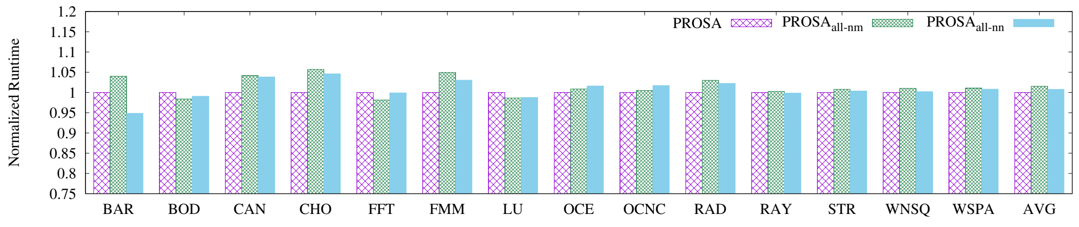
Configuration	area (μm^2)	overhead
Baseline router	243784	-
PROSA router (PR)	248200	1.8%
PROSA Controller (PC)	76547	-
Baseline cluster	975136	-
PROSA cluster	1069347	9.66%
Flattened Butterfly cluster	1380109	41%
2xBaseline cluster	1658560	70%

TABLE 3.7: Area overheads for different router and NoC organizations.

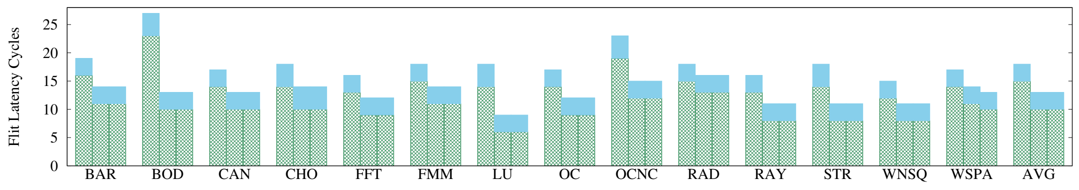
As we can see, the PROSA router takes only 1.8% more area than the baseline router. The PROSA controller (PC) takes less area than a baseline router (22% of baseline router area). However, this component is new and needs to be considered as an additional overhead. To make this comparison fair, the table shows area of cluster regions. In this case, the PROSA cluster takes 9.66% more area. The NIC's overhead can be considered as negligible.

PROSA overheads can be seen as high. However, we should consider the performance gains that PROSA circuits enable. In order, however, to better assess the overheads, the table shows two additional configurations worth being analyzed. The first one, $flattenedbutterfly$, is the overhead for a flattened butterfly topology which has more connectivity along each dimension and direction. In this case, because of the larger number of input ports, the area overhead is increased by 41%. The second one is for the case where the baseline cluster is enhanced with double flit size. This can lead to faster transfer times between the nodes. However, as we can see, overhead skyrockets to 70% additional area. This is mainly due to the larger buffer requirements.

Figure 3.30 shows the power consumption results for the entire network, PROSA and PS network. Although the leakage power increases by 42% the total power consumption is reduced by 3%. This reduction is due to switching and internal power, which are



(A) Application Runtime



(B) Flit Latency

FIGURE 3.31: Performance results for different architectures (PROSA, PROSA$_{all-nn}$, PROSA$_{all-nm}$ in column order).

reduced by 10%. As described in [14] Deja Vu achieves a 30% power reduction, which is similar to our results (but without the latency improvements). Notice that the 30% execution time reduction will translate in major power savings as shown in [72][2].

Notice that in some applications (BAR, FMM, LU, WSPA) overall power consumption is slightly higher with PROSA. In these applications, internal and switching power consumption is roughly the same with PROSA and BASELINE (due to the higher number of delayed circuits that could not be used). This fact, combined with PROSA incurring in higher leakage (due to its additional network), makes the overall power consumption to be slightly higher. Notice, however, PROSA achieves energy savings by reducing application runtime.

3.2.3.3 Enhanced PROSA

In this section we analyze the performance achieved by PROSA with the different added functionalities presented before. First, we analyze PROSA$_{all}$ when circuits are configured for all messages. In this version, no slack is provided nor distance threshold. Different priorities between PROSA messages will be explored. Then, we focus on PROSA version with distance thresholds and slack for circuit setup process. This method is called PROSA$_{all-dd-slack-xy}$ where x and y will identify threshold and slack values.

[2]In [72] authors show that in modern computing systems, DVFS gives much more limited power savings with relatively high performance overhead as compared to running workloads at high speed and then transitioning into low power state.

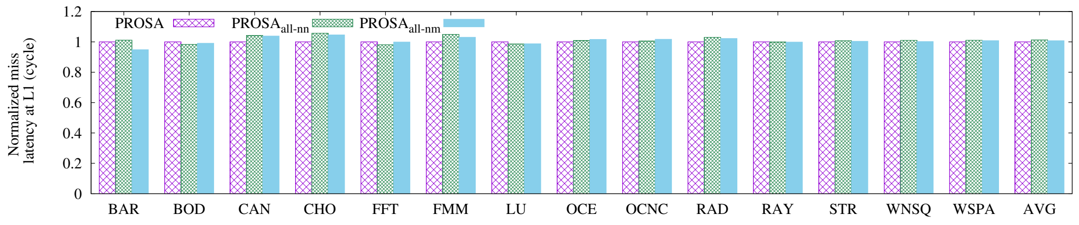


FIGURE 3.32: Memory latency results for different architectures. Normalized to baseline case (PROSA, PROSA$_{all-nn}$, PROSA$_{all-nm}$ in column order).

3.2.3.4 Circuits for all Messages and Different Priorities

Figure 3.31 shows the comparison between baseline PROSA and PROSA$_{all}$ with two different versions. The first one (PROSA$_{all-nn}$) with no NACKs on the PROSA circuit network, and the second one (PROSA$_{all-nm}$) with NACKs with lowest priority ($ACK > ReqCir > NACK$). Figure 3.31a shows the normalized execution time for different applications. Notice the Y axis which does not start from zero value (for the sake of clearly showing differences). As we can see, differences in execution time are very small. However, when comparing flit latency (Figure 3.31b) differences are much more noticeable. Both PROSA configurations (PROSA$_{all-nn}$ and PROSA$_{all-nm}$) reduce baseline PROSA flit latency by 35%. In flit latency PROSA$_{all}$ benefits range from 67% (in applications where PROSA achieves a small improvement in terms of flit latency; see Fig. 3.27), e.g. BODYTRACK) to 18% or 32 % (in applications in which PROSA gets higher improvement in that metric; see Fig. 3.27), e.g. RAD or CANNEAL and FFT respectively.

It has to be noted that although flit latency is significantly improved, the execution time of applications barely changes. This effect is due to the extra delay paid by PROSA$_{all-nm}$ and PROSA$_{all-nn}$ at the protocol level. Indeed, by tunneling all messages through circuits some protocol messages are delivered out of order, triggering race conditions in the coherence protocol. Those races have been fixed by adding additional protocol states or by delaying some protocol requests in order to enforce the strict ordering of messages. Indeed, Figure 3.32 shows the average L1 miss latency results. As we can observe, the different PROSA versions achieve almost identical miss latency values.

More interesting is the fact that both new versions achieve almost the same flit latency values. Indeed, lowering the priority of NACK messages has the same effect as of removing them completely. This result suggests that NACK messages can be removed as they will hardly affect performance. Indeed, some area and power savings will be achieved by removing NACK support. Table 3.8 shows the overhead implementation of the PC without NACKs. As can be seen PC and cluster area is reduced by 16% and 2% respectively. Thus, PROSA$_{all-nn}$ reaches similar performance results as PROSA$_{all-nm}$ but reducing the area.

Configuration	area (μm^2)	overhead
PROSA router (PR)	248200	
PROSA Controller (PC)	76547	
PROSA$_{all,nn}$ Controller (PC)	64918	-16%
PROSA cluster	1069347	
PROSA$_{all,nn}$ cluster	1057718	-2%

Table 3.8: Area overheads for PROSA controller without NACK .

We concluded PROSA$_{all-nn}$ and PROSA$_{all-nm}$ being almost identical in performance was in average terms. However, for every application results are slightly different and significant in BAR. The trend is that not using NACKs increases performance. In BAR, the pollution created by NACK messages leads to critical circuits to get delayed and thus miss shorter message latencies. This is, on average, cancelled due to the large amount of traffic each application injects. However, delaying a process punctually will lead to a higher execution time because of barriers and synchronization events between processes. This is the root cause of higher runtime but similar average latencies.

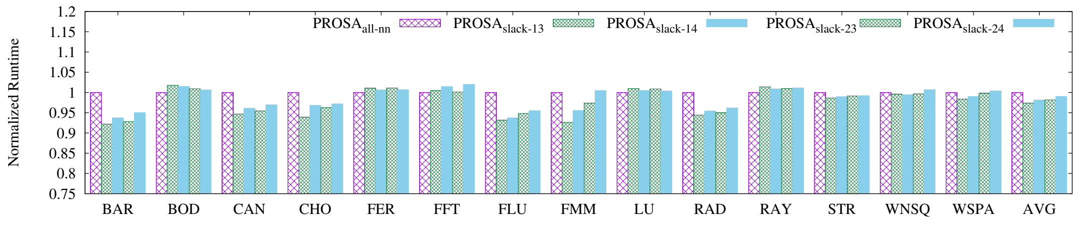
	REC.ACK	REC. NACK
MIN	(OCNC) 62.81%	(LU) 14.36%
AVG	73.25%	26.75%
MAX	(LU) 86.64%	(OCNC) 37.21%

Table 3.9: Number of ACK and NACK messages generated in PROSA$_{all-nm}$.

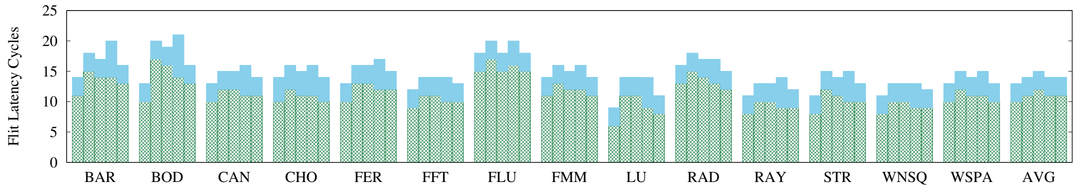
CONF. INPUT			CONF. RA		
	NACK	ACK	FB	TMP	ARB
MIN	0	(STR) 5.25%	(BOD)0.02%	(LU) 5.75%	(LU) 3.15%
AVG	0	7.84%	0.18%	12.30%	6.71%
MAX	0	(BOD) 17.15%	(FMM) 0.55%	(OCNC) 15.96%	(OCNC) 9.41%

Table 3.10: Number of conflicts generated in PROSA Controller.

Tables 3.9 and 3.10 show statistics about PROSA circuits for PROSA$_{all-nm}$. Notice that the current circuit setup success rate (73.25% on average for all applications) is much smaller than the one achieved with baseline PROSA. This is due to the higher traffic of the PROSA$_{all}$ network to establish circuits. Now, for every message a circuit setup process is launched. However, what is noticeable is that NACK messages do not introduce any conflict at the input of the PC device. Full bank conflicts at ResArb modules is also low (0.18% on average). The more prominent conflicts now are those related to the temporal conflict with already programmed circuits (12.30% on average) and due to arbiter conflicts (two concurrent requests, 6.71% on average). Notice that in this configuration each ResArb module implements a circuit table with four entries. These results are similar to the ones achieved by PROSA$_{all-nn}$. Because of the more efficient implementation, from now, we use PROSA$_{all-nn}$ for all the following experiments.



(A) Application Runtime



(B) Flit Latency

FIGURE 3.33: Performance results for different architectures (PROSA and PROSA DD in column order).

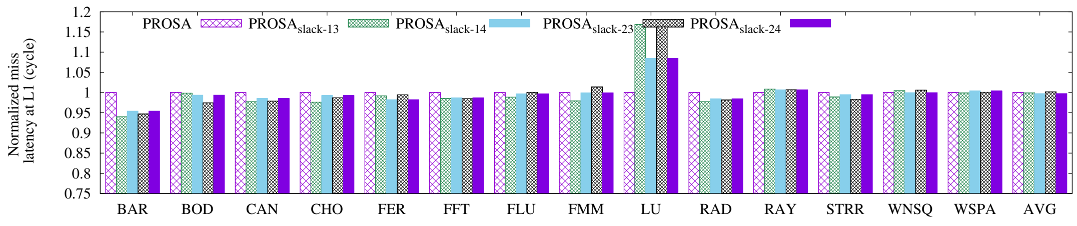


FIGURE 3.34: Memory latency results for different architectures. Normalized to baseline case

3.2.3.5 Bounded Circuits and Slack

Now we focus on the full deployment of PROSA, which includes the distance threshold and the slack for circuit setup process (PROSA$_{all-dd-slack-xy}$). We analyze two slack (x) values (1 cycle and 2 cycles) and two distance (y) thresholds (3 and 4 hops in PC, 6 and 8 routers, respectively). Thus, PROSA$_{all-dd-slack-13}$ represents PROSA with 3 PC hops distance threshold and 1 cycle for slack. All these versions do not include NACKs (similar to PROSA$_{all-nn}$).

Figure 3.33a shows the application runtime for the four new configurations and for the previous PROSA$_{all-nn}$ version. On average, all these four configurations slightly improve PROSA$_{all-nn}$. Among all, PROSA$_{all-dd-slack-13}$ is the best configuration, which outperform applications execution time up to 6% in some applications (CANNEAL, CHO, RADIX). On average performance improves by 3%.

Figure 3.33b compares flit latencies for these configurations. PROSA$_{all-nn}$ always gets the lowest flit latency for all applications. As can be seen, PROSA$_{all-dd-slack-xy}$ gets worse latency results with low threshold value. However, this does not affect L1 miss

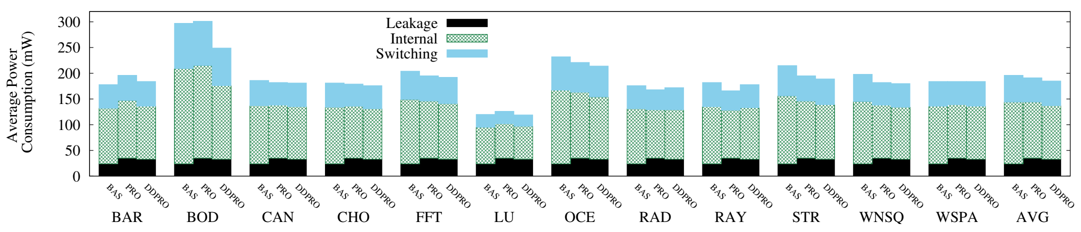


FIGURE 3.35: Power consumption results. First column represents BASELINE, sedond PROSA and third PROSA$_{all}$.

latency. Figure 3.34 shows the miss latency results which follows the same trend observed for the execution time of applications. Applications with higher miss latencies (LU) have higher execution runtimes. As it occurred with PROSA$_{all-nn}$, improvements in the network (faster messages) is not reflected in execution time of applications due to the extra delay incurred in the protocol.

	3 HOPs		4 HOPS		5 HOPS		unbounded	
	ACK	NACK	ACK	NACK	ACK	NACK	ACK	NACK
BARNES	77.52	22.48	72.78	27.22	70.51	29.49	68.25	31.75
CHO	80.31	19.69	76.90	23.10	75.09	24.91	73.03	26.97
FFT	83.38	16.62	80.10	19.90	77.61	22.39	75.41	24.59
OCEAN	84.44	15.56	80.19	19.81	77.29	22.71	75.24	24.76
LU	91.20	8.80	88.65	11.42	86.24	13.76	84.27	15.73

TABLE 3.11: Percentage of ACKs/NACKs for PROSA$_{all-dd}$ with different distances.

Table 3.11 shows PROSA statistics for different applications when varying the maximum distance of circuits inside the PC network. As we observe, the number of NACKs increases as the allowed distance of circuits increases. This is mainly motivated by the traffic increase in the PC network. As seen in the table more NACKs are generated, delaying injection of associated messages. Therefore, maximum distances of 3/4 PC hops looks like the right approach to minimize NACKs while still guaranteeing long circuits.

Finally, Figure 3.35 shows the power results for PROSA$_{all-dd-slack-xy}$. On average, the new PROSA version reduces the power consumption by 7% and 4% to BASELINE and baseline PROSA, respectively. Network traffic in the PC network in PROSA$_{all-dd-slack-xy}$ is 5 times larger than the observed in PROSA. However, the new PROSA reduces power consumption in the standard network by 35% on average.

3.2.3.6 Discussion

From the previous results we can observe different aspects worth being highlighted. The first one is the clear benefit of building an infrastructure to setup circuits for coherence protocol-oriented systems. By taking advantage of large memory access latencies the network can be configured with communication circuits configured and kept only for the transmission duration of the message. PROSA is able to achieve remarkable results in execution time and flit latencies. PROSA can set a circuit in less than 16 cycles in the worst case for an 8×8 mesh network. Although this can be enough for standard memories, for faster ones or larger networks maximum circuit distance has to be bounded.

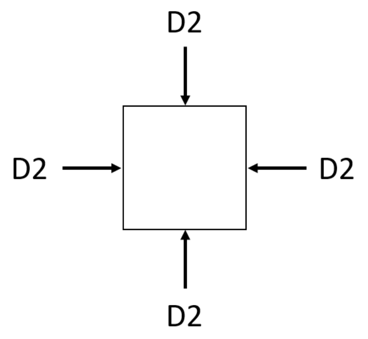
The second conclusion is the fact that tunnelling all the messages (short and long) through circuits does not necessarily lead to benefits in application performance. Execution time is barely the same and, most important, the coherence protocol must be adapted to support new race conditions triggered by messages arriving out of order. What is more interesting to note is the fact that negative acknowledgments of circuits can be discarded and it is worth relying on an automatic and silent circuit tear down process. Further savings in power and area implementation are achieved. Indeed, performance is not significantly affected.

A third conclusion we obtain is related with the circuit setup delay and its effects on performance. Indeed, by bounding the reachability of circuits performance of applications can be improved with a further 6%. This is easily achieved by adding a simple comparator on each NIC. Longer circuits tend to have a larger circuit setup time, which indeed is larger than the cache access time, incurring on a extra delay to send the message through a circuit.

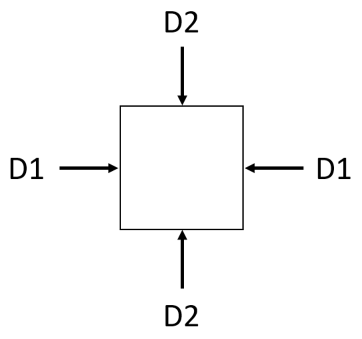
Finally, the use of a slack for circuits should be introduced with care. A large slack may lead to worse results as it induces more temporal conflicts in the network. Indeed, contention in the circuit setup network is low and, thus, with only one cycle of slack most of the circuits are successfully configured and used, leading to good performance.

One critical aspect of PROSA is its scalability. First, the current design allows only one ReqCir request to access the RT. This may create a bottleneck. However, the low/medium load traffic seen on the PC network does not compromise its performance. Anyway, if scalability problems appear, it would be solved by implementing a two-port RT module. On the other hand, for PROSA$_{all}$, small-sized messages with distant destinations could prevent long-sized messages with nearer destinations, potentially affecting performance. However, some of those small-sized messages are in the critical path of memory access by the processor, thus, having a high impact on performance. This fact makes the distinction between short and long messages less significant.

As a final conclusion, the best PROSA configuration would be the one with no NACK support, one cycle slack, and with a threshold of distance three for the 8×8 network.



(A) Requirement for router-level strong isolation.



(B) Scenario where domain interference may cause latency variations.

FIGURE 3.36: Scenarios for router-level slot allocation to domains D_i.

3.3 High Assurance Networks for CMPs

The previous contributions focused on NoC performance improvements by using mechanisms addressing both throughput and latency optimizations. Now, in this last contribution we focus our attention on high assurance property for the network. More specifically, we target the domain of systems with multiple applications running on the same chip and with strong isolation requirements mainly due to security reasons. When strong isolation is required even cycle-level variations must to be prevented in order to avoid timing channel attacks in which the attacker can get information from timing variations. For these reasons Token-Based TDM enforces non-interference between traffic logically belonging to different partitions. Therefore both types of flows, flows between sender-receiver pairs within the same partition, and flows from/to tiles of a partition to/from any MC, must be prevented from a timing variation.

3.3.1 Router-Level Strong Isolation

To achieve such property, the network relies on a token propagation scheme. Tokens contain scheduling orders to local router-level domain schedulers. For this purpose, tokens carry a domain identifier (DI), which identifies the domain whose packets can be forwarded from a specific router input upon arrival of the token. *In order to deliver strong isolation between domains, we need to synchronize the timing of token propagation throughout the network in such a way that every router gets the same DI at each input port on every cycle* (see Fig.3.36). This means that the router will arbitrate and forward messages belonging to the same domain/partition, and messages from different domains will never compete.

The goal is to create and propagate as many types of tokens as the number of running domains, each one with its own Domain Identifier (DI). All different types of tokens are triggered back-to-back in sequence from a source node in the network, and propagated through it following the CDG. The proposal triggers one new kind of token at each cycle, and repeats the domain sequence every D cycles, where D is the number of domains. As tokens are received in sequence at router input ports, they command the local scheduler to serve packets with a specific identifier from those ports.

3.3.2 Synchronized Token Propagation

Since tokens traverse router ports and links in the order of the CDG, synchronized same-ID token arrival at all router input ports depends on the CDG and nominal router and link latencies.

Assuming an horizontal Segment-based Routing algorithm [73] and single-cycle routers and links, tokens would be triggered from the bottom right corner, and would be propagated throughout the network as illustrated in Figure 3.37. With this routing algorithm, there are two token propagation phases, a scroll-up one (left) and a scroll-down one (right), which occur one after the other. Their combined effect is the traversal of all router ports and NoC links. Numbers in the figure indicate token propagation latencies since initial injection. The diagonal arrows represent routing restrictions, that is, directions that packets can NOT take as dictated by the routing algorithm at hand. They are set in order to prevent cyclic dependencies, that is, deadlock from occurring.

If we focus on a single router and a single token crossing the network as specified in the CDG, the token will reach different input ports of the same router at different timestamps t1-t2-t3 and t4, as seen in Figure 3.38.

Let us define *relative latencies* as the time periods elapsed between any two consecutive timestamps. *router-level operation with strong domain isolation requires that the result of the modulo operation between any of these relative latencies and the number of domains is always 0*:

$$\forall (y,x) \in A \rightarrow (ATy - ATx) mod D = 0, where ATy > ATx$$

where A is the set of router input ports, D is the number of running domains, and ATi is the arrival time of same-ID tokens at input port i. In fact, as the proposed architecture injects the same domain identifier token into the network every D cycles, this ensures that a token with the same identifier will reach a specified port every D cycles.

Therefore, if the previous condition is met, at regime all the router input ports will receive tokens with the same domain identifier exactly at the same time, and can thus work in strong isolation mode. The number of domains D that enables this operating condition

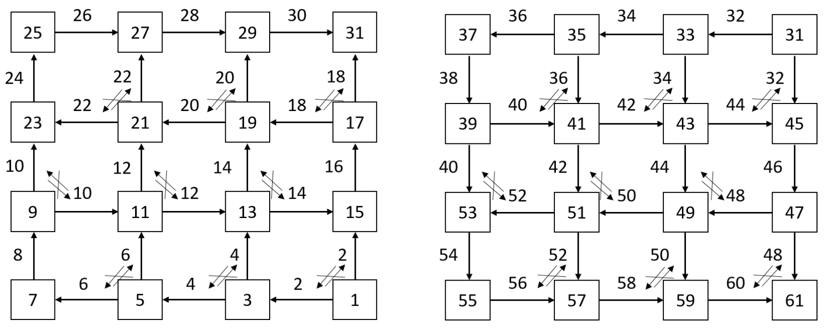


Figure 3.37: Network-level token propagation, with latency annotated, in the order of the CDG with horizontal segment-based routing and single-cycle routers and links.

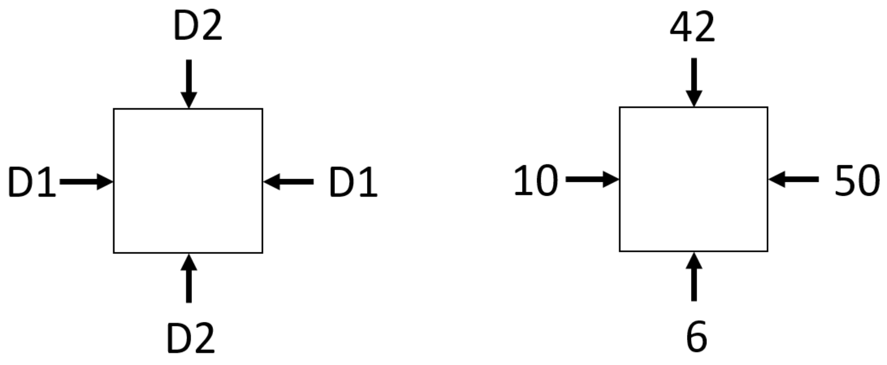


Figure 3.38: Arrival times (in clock cycles) of same-ID tokens at router input ports.

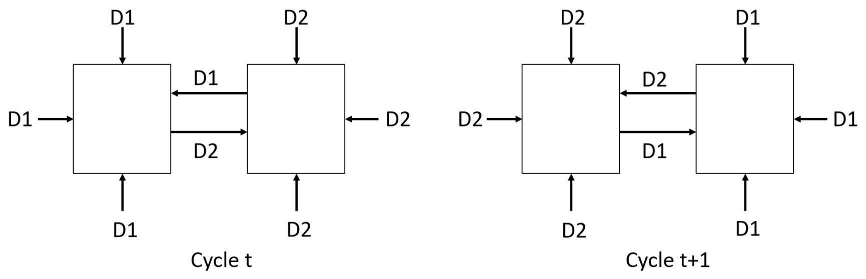


Figure 3.39: Perfect scheduling at network level for minimum latency.

is the Maximum Common Divisor (MCD) of all relative latencies between consecutive pairs of arrival times (in increasing order).

Following the example in Figure 3.38, relative latencies from initial token arrival to its presence on all input ports amount to 4, 32, and 8 cycles. Hence, the router delivers strong isolation with 4 domains.

To extend this property to the whole network, we first compute the maximum number of domains for every router as the MCD illustrated above (if any). If such an MCD can be computed for each router, then the topology, coupled with the target routing algorithm, supports strong isolation of domains. In this case, *the MCD of all router-level computed*

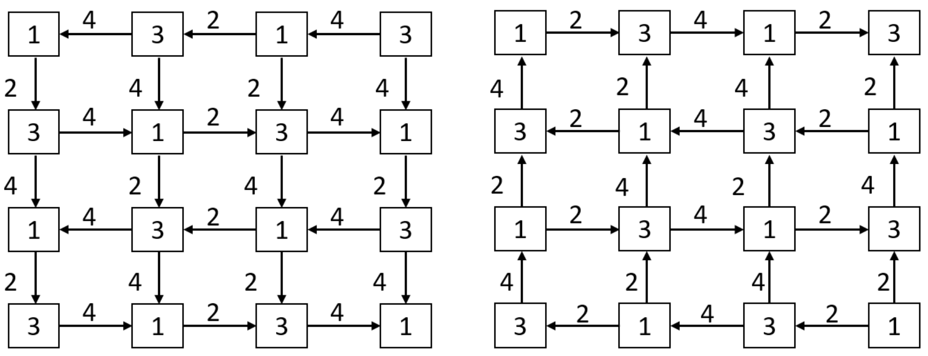


FIGURE 3.40: Token propagation flow at regime in a specific time slot. Numbers denote the token ID served on a specific NoC resource at that clock cycle.

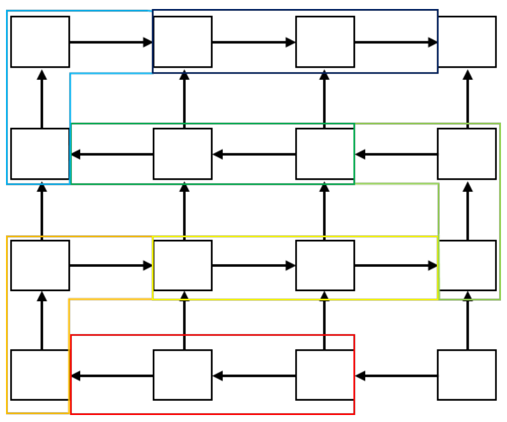
domains is the ideal number of domains which delivers strong isolation for the network as a whole. In the 2D mesh example in Figure 3.37, perfect scheduling is achieved with 4 domains.

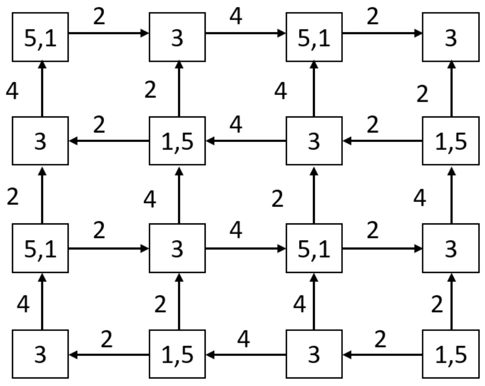
Notice that the ideal number of domains for perfect scheduling can be directly determined by the smallest relative latency inside the network. In our case, the smallest relative latency of the NoC in Figure3.37 is exactly the one reported in Figure 3.38 (which is 4 cycles), and corresponds to the latency of the smallest cyclic path spanned by a token in the network to reach two different ports of the same router. Clearly, this cyclic latency depends on router and link latency, and is equal to: $(R - 2) * (P + L)$ where R is the number of routers in the smallest cycle, P is the router latency and L is the link latency.

A relevant side effect of this theory is that the composition of strongly-isolated router-level operations at network level through the dependencies of the CDG gives rise to a *Perfect Schedule*, which consists of the onset of unstopped propagating waves of same-ID tokens throughout the network (see Figure3.39). This guarantees minimum-latency operation of the NoC.

Figure 3.40 shows token propagation flow over the network at regime, which is established once the first token comes back to the injecting router. Each router works in strong isolation mode, and globally a perfect schedule takes place.

This methodology generalizes the constraints for perfect scheduling identified in [61] from local scheduling loops, which we instead base on the observation of the CDG. This generalization is at the core of the new approach proposed, featuring enhanced flexibility as hereafter illustrated.


(A) Building segments out of the shortest token cycle.



(B) Selective stall placement to support 5 domains with strong isolation. Only scroll-up phase shown.

Figure 3.41: Extending the number of domains under strong isolation.

3.3.3 Supporting a Higher Number of Domains

In order to handle a larger number of domains than the ideal one, PhaseNoC (see a description of how it works at Chapter 2) requires deep modifications of the router architecture, thus proving unsuitable for runtime reconfigurability. In particular, the pipeline depth of all routers needs to be increased, which would preserve strong isolation and perfect scheduling. As we will see in the experimental results, for some configurations this leads to suboptimal performance. Alternatively, the NoC can be split into communicating subnetworks, similar to SurfNoC. In this case, the strong isolation property is lost (i.e., domains can affect timing of packet propagation), unless area-expensive input speedup is implemented to avoid VC contention at crossbar inputs.

Our CDG-inspired approach is less invasive and more easily reconfigurable at runtime, and aims at preserving strong isolation in router-level operation while giving up the perfect schedule globally.

As we have seen previously, in a regular 2D mesh network all relative latencies are multiples of the latency of the shortest cycle SCL (i.e., of $SCL = (R - 2) * (P + L)$). Therefore, we can split the critical path of the tokens throughout the CDG into segments of length SCL cycles (see Figure 3.41a). With the ideal number SCL of domains, all inputs to these segments will be in the same domain at a given time slot.

In order to support a higher number of domains D, our intuition is that only SCL domains should be in flight at any given point in time. The remaining $D - SCL$ domains should be stalled. This would enable to preserve the strong isolation property, while breaking the perfect scheduling assumption. As we will show in the experimental results, preserving perfect scheduling at all costs may not be the best performing solution.

In order to implement this concept, we need to place domain propagation stalls selectively within segments. The constraint to be met for correct operation is to place these stalls at the same positions within segments (e.g., either in the first router, or in the second one). Figure 3.41b shows an example of stall placement to support 5 domains. As can be observed, the stalls allow the network to receive the same domain identifier at all the input ports at every router.

Depending the target number of domains D and the ideal number of domains SCL, the number of stalls in each segment can be calculated as follows:

```
if D > SCL then
    $Stalls = D - SCL$;
else
    $Stall = 0$;
end if
```

3.3.4 Supporting a Lower Number of Domains

If at any given point in time, a lower number of domains than the ideal one needs to be enforced, then *strong isolation and perfect scheduling can be preserved provided the target number of domains is an integer divider of the ideal number.* The reason is because this way the latency of the shortest cycle spanned by tokens to bridge two consecutive ports of the same router is a multiple of the repetition period of domain identifiers. Therefore, previous conclusions are still valid. As an example, with ideally 12 domains in a perfect schedule with strong isolation guarantees, the same property can be derived with 2, 3, 4 and 6 domains.

With a different number of domains (e.g., 5 or 7), the same stall-based methodology explained before can be applied here, which preserves strong isolation but not perfect scheduling. In particular, the procedure that follows should be used to realign token domain identifiers at segment inputs with $d < SCL$ domains:

- compute an integer n such that $n \times d > SCL$;
- compute the number of stalls per segment as $stalls = n \times d - SCL$;
- enforce this number of stalls at the same position within each segment.

3.3.5 Handling the Transient to Regime

Before getting to regime, tokens start propagating in the network by following the CDG. However, not all input ports have received a token at a given point in time.

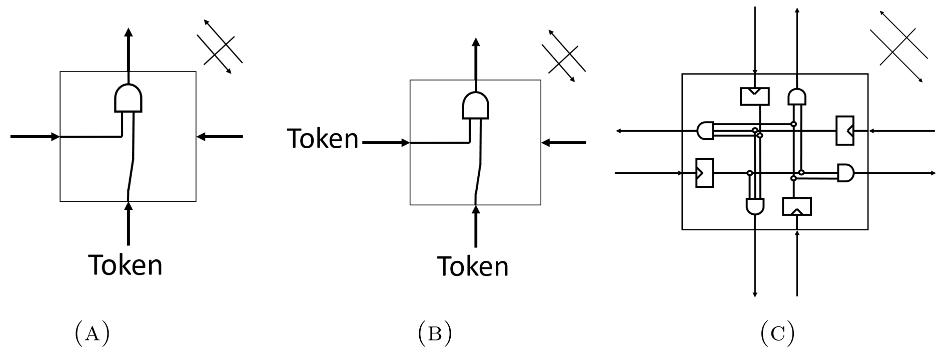


FIGURE 3.42: Token propagation mechanism.

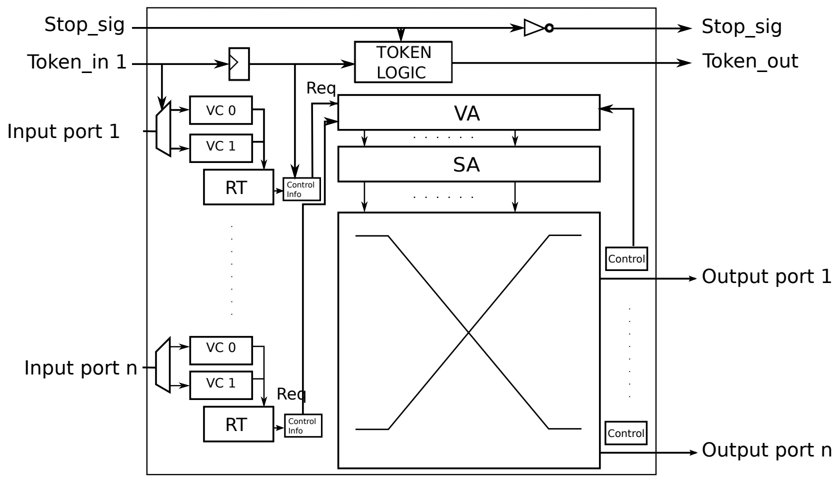


FIGURE 3.43: Proposed router architecture: the concept.

Figure 3.42 shows an example of initial token propagation through a router. For the sake of simplification, the figure only shows output port north, and input ports east, west and south. Arrows denote routing restrictions between the east and the north ports. Then, based on this CDG, only packets from the south and west input ports can be forwarded to the north output port. Figure 3.42a shows that a token has been received from south. However, this token cannot be forwarded downstream, since not all the input ports with routing dependencies with the north output port have received such a token. Therefore, the south token is dropped. When the token reaches the west input port as well (Figure 3.42b), it will appear again also in the south input port (see Section 8.6.2), then a new token can be forwarded to the north output port.

3.3.6 Router Architecture

Figure 3.43 shows a simplified version of the Token-based TDM router architecture. Only one input port is shown in order facilitate understanding, assuming one VC per domain. Single-cycle routers and links are considered.

Tokens are in charge to set the domain processed within the router. The incoming token ($TOKEN\_IN$) is used as a selector in the first demultiplexer stage to define in which domain buffer the incoming data flit will be stored. The $TOKEN\_IN$ is stored in a register for the sake of retiming, so that in the next clock cycle it can select the active domain within the router. Following this approach, only one domain can access to the VA/SA/X per cycle, then no conflicts between different domains can occur.

The token advances until the token logic, which is active only during the initial configuration transient, till the regime is reached. Until then, the logic checks whether tokens are available at all input ports with routing dependencies with the target output port. If not, tokens are dropped. At regime, all such tokens will be available, and an output token ($TOKEN\_OUT$) will be fired.

This initialization procedure is slightly more complex in case the network is initialized with a higher number of domains than the ideal one for perfect scheduling. In this case, the first domain identifier token carries the required number of domains. At the same time, the STOP signal is set to one. The token logic then computes the number of stalls to be enforced. Negation of the STOP signal enables to enforce stalls selectively once every two routers. After this, domain identifiers are injected in sequence at every cycle as usual. As an incoming domain identifier token arrives at the input port, it has to wait in a token buffer (of size $Dmax - SCL$) until it reaches the first position in the queue.

The proposed architecture can be inherently reprogrammed to support any number of running domains, since the adaptation is not in the architecture itself, but in the scheduling commands sent through the tokens. In the most complex case, the token logic has to compute, from the initialization procedure, the number of stops for tokens in transit. While in this contribution we are experimentally characterizing the offered architectural flexibility, the precise reconfiguration protocol to exploit it at runtime (i.e., to safely transition from one network configuration to another) is left for future work.

Finally, in a traditional router with M-ports, D domains and v VCs per domain, there exist $M \times D \times v$ input and outputs VCs. Then, virtual channel allocation maps $M \times D \times v$ inputs to $M \times D \times v$ outputs VCs. However, our proposal only allows one domain to access the VA arbiters at a time, then only $M \times v$ VCs perform VA at every cycle. Similarly, the $M \times D \times v$ to $M \times D \times v$ router allocator in traditional routers is reduced in our implementation to a $M \times v$ to $M \times v$ allocator.

3.3.7 Experimental Results

For the experiments, we use the VirtualSocLite virtual platform [74], targeting the full-system simulation of massively parallel heterogeneous SoCs. It is coded in SystemC

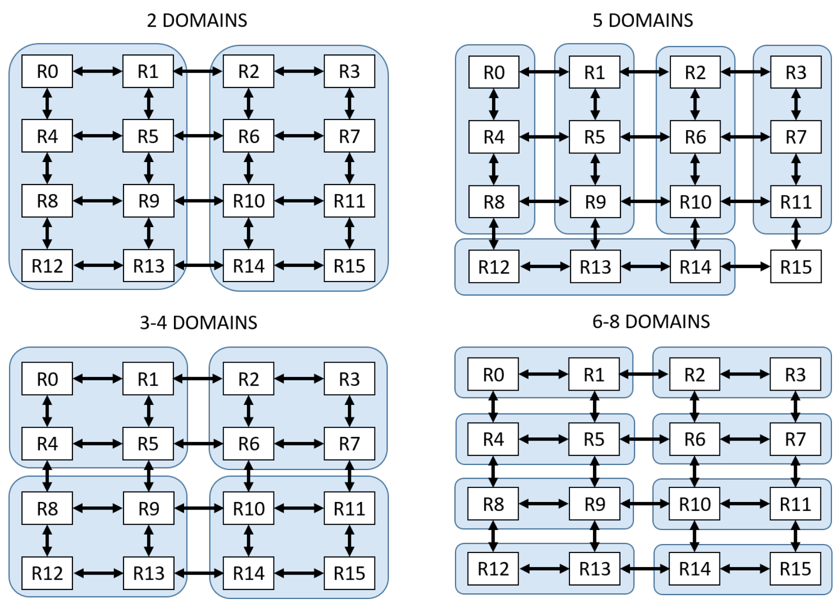


FIGURE 3.44: Different number of domains considered on a 16-tile 4x4 network.

and models operation of a 2D mesh topology of any size with RTL-equivalent accuracy. The platform has been extended to model the proposed NoC architecture, and synthetic traffic generators have been instantiated and linked to the NoC.

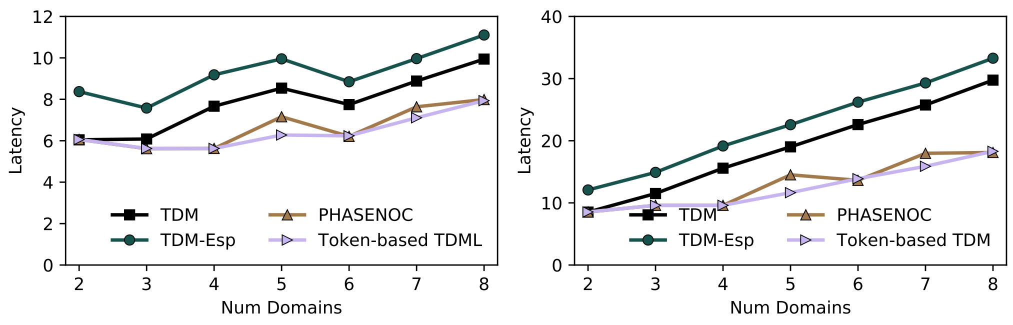
For all the results, we inject write transactions from the tiles to the distributed L2 banks of a partition (intra-domain traffic). Inter-domain traffic is obtained by injecting additional write transactions from domain tiles to memory controller nodes (MC traffic). Two topologies sizes are simulated: 4×4 and 8×8 2D-mesh topologies. In both cases, there is one core per tile.

Different configurations of domains are used. In all cases, domains are of the same size and geometry. Figure 3.44 shows the domains for the 16-tile network.

In the 8x8 2D mesh, in case of 2 domains, the network is divided into 2 domains of 32 tiles each. For 3 and 4 domains, the domain size is 16 tiles. For 5 domains, the network is split into domains of 12 tiles; if the networks requires between 6 and 8 domains, these domains are composed by 8 tiles. In case more than 8 domains are required, the domain size is set to 4 tiles.

We evaluate four mechanisms across a different number of running domains, to test the flexibility requirement:

- TDM. A baseline TDM network where at each time slot the whole network is used for a specific domain. All domains are served in consecutive order. At each time slot, both intra- and inter-domain traffic for the associated partition is served.



(A) Local traffic. (B) MC traffic.

FIGURE 3.45: Zero-load latency for the 4x4 2D-mesh.

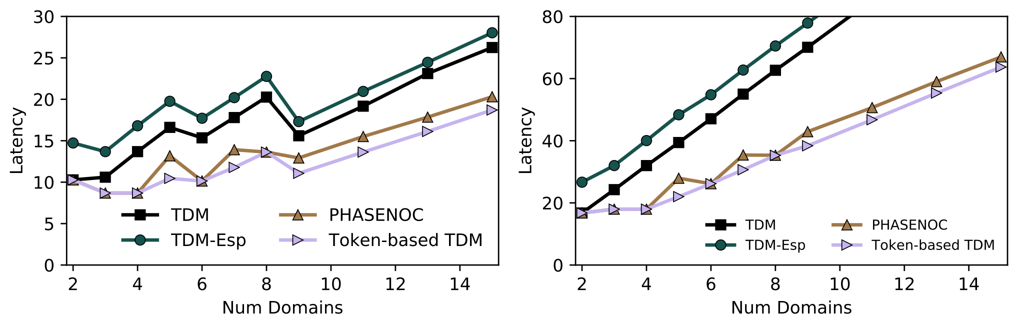
- Custom TDM, where the TDM scheme is made aware of spatial partitioning. In practice, a dedicated time slot is concurrently used for intra-partition traffic of all domains. Partition-specific local traffic is kept segregated by the regularity of partition shapes. Then, every partition has one additional reserved slot to transmit or receive its own MC traffic. This scheme will be briefly referred to as *TDM-esp* in the results.

- PhaseNoC, which delivers strong isolation and minimum communication latency under specific configuration options (no. of domains). For each tested configuration, we tune the PhaseNoC router with the proper number of pipeline stages for perfect schedule and strong isolation (although this would be problematic to apply at runtime). However, in some cases PhaseNoC ends up in a suboptimal configuration. For instance, PhaseNoC with two pipeline stages per router can support 6 fully-isolated domains in perfect schedule, but if the required number of domains is 5, one domain would go unused. In contrast, if we revert back to 1 stage per router, strong isolation cannot be provided with 5 domains. For a fair comparison, we allocate such an unused time slot to the active domains in a round robin fashion. With PhaseNoC, both intra-domain traffic and MC traffic are served when a domain is active. PhaseNoC would also enable the partitioning of the network into subdomains to handle the critical cases. However, we verified that in this case the strong isolation property is lost, unless input speedup is implemented. We do not consider this case here, since it would amplify the implementation complexity gap between PhaseNoC and our approach.

- Token-based TDM. This is our approach, which serves both inter- and intra-domain traffic when a domain is active.

3.3.7.1 Zero-Load Latency

Figure 3.45a shows the zero-load latency results for local intra-domain traffic. The x-axis represents the number of domains and the y-axis the zero-load latency. As can be



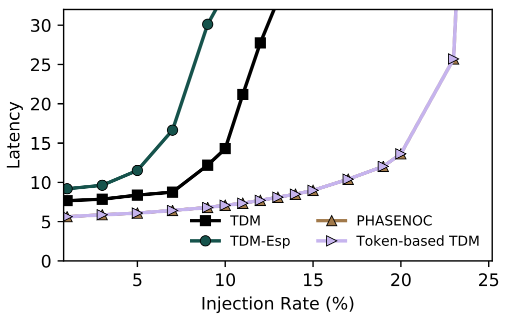
(A) Local traffic. (B) MC traffic.

Figure 3.46: Zero-load latency for the 8x8 2D-mesh.

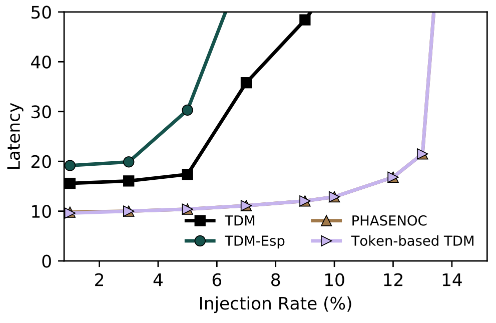
observed, both PhaseNoC and Token-based TDM improve upon the baseline TDM variants. However, our proposal improves upon PhaseNoC in scenarios where PhaseNoC's pipeline has to be oversized for strong isolation (5 and 7 domains). The improvement on the network latency is 13% and 9 % for 5 and 7 domains, respectively. In these scenarios, we claim "generalized perfect scheduling" through selective placement of stalls. In scenarios where PhaseNoC achieves perfect scheduling by increasing the router pipeline depth (with 4, 6 and 8 domains), our proposal provides the same perfect scheduling as PhaseNoC. Thus, our approach provides increased flexibility without wasting performance. In order to properly read the plot, it should be noted that the higher the number of domains the smaller the partition size (Figure 3.44). However, latency tends to increase because there is a higher waiting time to pay in the network interfaces to wait for the suitable injection time slot.

Figure 3.45b plots the zero-load latency results for MC traffic. The performance reached with this type of traffic is similar to the local traffic one. Again, we appreciate the capability of Token-based TDM to keep up with PhaseNoC whenever the latter delivers perfect scheduling, while improving the network latency to access the MCs by about 20% for 5 domains and by 12 % for a 7-domain configuration. Thus, our approach proves capable of generalizing the high performance efficiency of PhaseNoC to the whole configuration space, while exposing inherent reprogrammability of the number of domains without pipeline modifications. We instead just change the scheduling commands we give to the routers. Another architectural difference explains the above results: while our approach tends to introduce a configurable number of stops for domain identifier tokens in specific points of the NoC in order to preserve the non-interference property, PhaseNoC tends to spread the latency overhead everywhere in order to achieve the same goal.

Figure 3.46a shows the zero-load latency results for local intra-domain traffic in a 64-tile scenario. As in the 16-tile network, Token-based TDM improves upon PhaseNoC in scenarios where the pipeline is oversized. Even the reallocation of the unused slot is not able to compensate for this inefficiency. The improvement of Token-based TDM

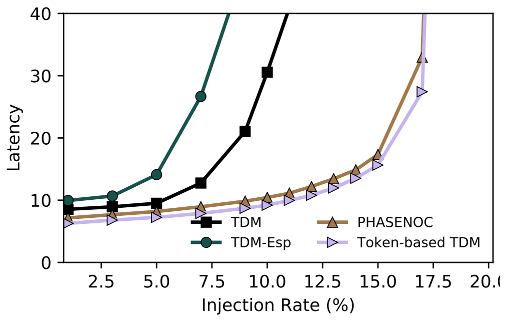


(A) Local traffic.

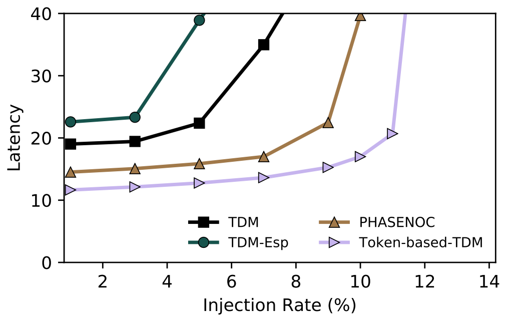


(B) MC traffic.

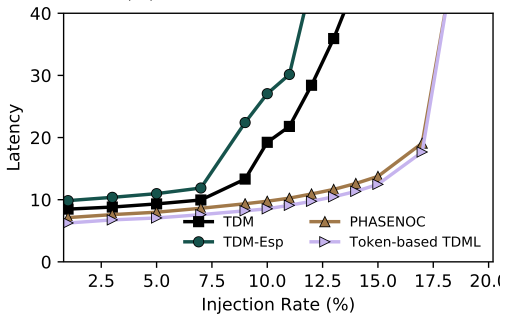
FIGURE 3.47: Performance with uniform traffic for 4 Domains



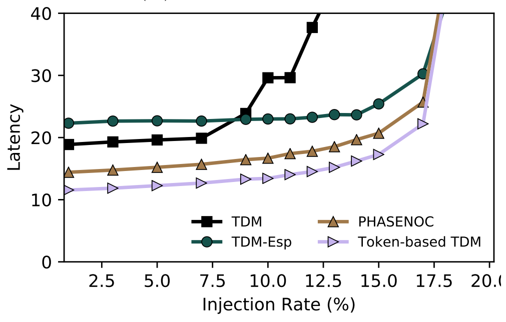
(A) 100 % Local traffic



(B) 100 %MC traffic



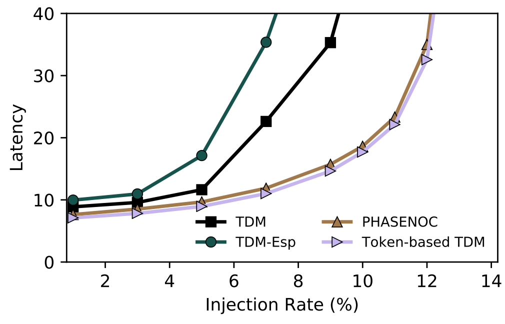
(C) 75 % Local traffic



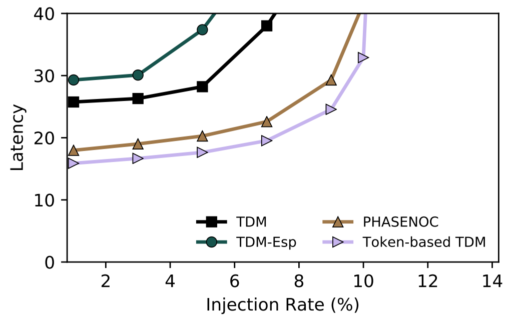
(D) 25 % MC traffic

FIGURE 3.48: Performance with uniform traffic for 5 Domains.

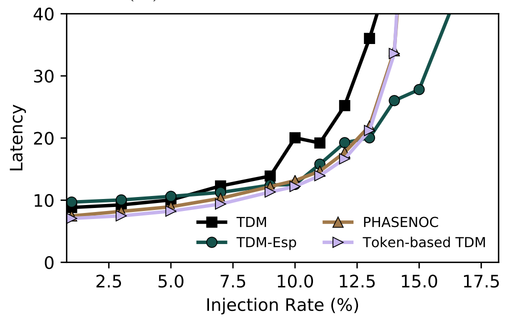
oscillates between 20 % (5 domains) and 9% (15 domains). For MC traffic (Figure 3.46b), Token-Based TDM reaches the best performance improving up to 30 % the PhaseNoC network latency. For a number of domains higher than 8, the cases where PhaseNoC gets perfect scheduling are not shown because in those cases Token-based TDM performs the same, as we have shown previously. Notice that, the higher the number of domains the smaller the benefit achieved by Token-based TDM. This is due to the fact that as the number of domains gets larger, the bandwidth underutilization by PhaseNoC decreases.



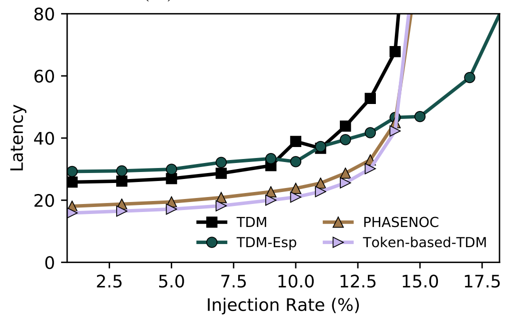
(A) 100 % Local traffic



(B) 100 %MC traffic



(C) 50 % Local traffic



(D) 25 % MC traffic

FIGURE 3.49: Performance with uniform traffic for 7 Domains.

3.3.7.2 Network Performance

Figure 3.47a shows the network saturation curve for local traffic under uniform random traffic. The characterized scenario consists of four domains. If our approach is correct, we expect both PhaseNoC and our approach to deliver perfect scheduling. As shown in the Figure, the Token-based TDM exactly matches the performance of PhaseNoC, thus validating the claim. Figure 3.47b shows the performance for MC traffic with 4 MCs, in the same characterized scenario. These plots further validate the same conclusion, hence validating the efficiency of our approach in *matching the best case for PhaseNoC, while delivering extended flexibility.*

Next we analyze the network saturation curves in those cases where our proposal can set a "generalized" perfect scheduling while PhaseNoC can not. The unused slot domain is assigned in round robin fashion to the active domains. For this study we analyze four scenarios: one where the whole traffic is local, another with only MC traffic, and two scenarios with mixed traffic, one with fifty percent of each type of traffic and another one with 75 % of local traffic and 25 % of MC traffic.

Figure 3.48a shows the results for local traffic for the 5-domain scenario. As can be seen, PhaseNoC and Token-based TDM improve the network performance of TDM. In addition, Token-based TDM reduces the network latency against PhaseNoC by 10 % along the complete network saturation curve, reaching the saturation point at similar injection rates. Figure 3.48a plots the results for MC traffic. Similar as the previous case, Token-based TDM outperforms PhaseNoC by up to 20 % before reaching the saturation point. Moreover, our approach increases the network capacity. When a mixed-flow scenario is presented, Token-based TDM continues to reach the best performance. Figures 3.48c and 3.48d show the results for local and MC traffic for a scenario with 75 % of local traffic and 25 % of MC traffic, respectively. Similarly to previous cases, Token-based TDM is able to reduce the PhaseNoC latency by roughly 10% and 20% for local and MC traffic, respectively. Moreover, in this scenario the customized TDM scheme (TDM-esp) has higher network latency compared with baseline TDM. However, for MC traffic TDM-esp is able to match the accepted traffic before reaching the saturation point. This behavior occurs because for a given domain with mixed flows, TDM-esp provides higher bandwidth than baseline TDM, because each domain has two time slots, one for intra-domain traffic and one for inter-domain traffic (for access to the MC).

Focusing on a 7-domain scenario, Figure 3.49 plots the latency results. Similarly to the 5-domain scenario, Token-based TDM reduces network latency by 8% in intra-partition traffic scenario and 15% in MC traffic scenario. Figures 3.49c and 3.49d plot the results for the mixed-flows scenario, where the traffic is divided in equal parts between both types of traffic. Token-based TDM gets the minimum network latency, improving over PhaseNoC by 7 % on local traffic and 12 % on MC traffic. However, similarly to what happens in the mixed-flows scenario for 5 domains, TDM-esp is able to accept a higher injection rate of traffic.

3.4 Proposals Digest

In order to provide a global picture, in Table 3.13 we show, for each contribution, its key characteristics. To summarize, the proposals presented in this chapter are:

- TBFC+SUR: this proposal presents a co-design of flow control and routing algorithm. It focuses the network throughput improvement on NoC with FA routing algorithms by optimizing the use of resources, achieving a balanced buffer utilization which improves the accepted traffic on the NoC and slightly reduces the network latency.
- EPC: This proposal achieves a new congestion control mechanism. It avoids spreading congestion by isolating the congested flows when using fully adaptive routing. EPC limits the adaptability of the FA routing algorithm to minimize spreading congestion.
- PROSA: This proposal presents a new circuit-packet switching NoC architecture driven by the coherence protocol. Coherence protocol messages are forwarded using a packet switching strategy, while for the data packets sets up circuits. PROSA reduces the latency of data messages using the coherence protocol access slack to set up the circuits before they are needed and only for the time period they are required.
- PROSA-DD: An extension of PROSA is shown in this proposal. This proposal sets up circuits for both type of messages, control, and data, including new functionalities such as additional slack for setting up circuits and distance-driven circuit setup strategy.
- Token-Based TDM: This proposal presents a novel TDM. It guarantees the strong isolation property required on high assurance network with a novel architecture based on the CDG. This new architecture is able to reduce network latency and it also increases the flexibility of previous TDM designs. Token-based TDM supports a different number of domains without any major change in the NoC design.

All proposals in this chapter are presented separately focusing on three different contexts. However, some combinations of these proposals can be applied to different contexts at the same time.

Table 3.12 shows the compatibility degree table between all the contributions (PROSA-DD is embedded in PROSA entry). An score of 3 means perfect compatibility, an score of 2 means high degree compatibility and an score of 1 means low degree of compatibility.

TBFC+SUR and EPC are highly compatible since they focus on the same context (fully adaptive routings algorithms) and they can be integrated easily in the router design. In

Contribution	TBFC+SUR	EPC	PROSA	TOKEN-BASED TDM
TBFC+SUR	X	3	2	1
EPC	3	X	2	1
PROSA	2	2	X	1
TOKEN-BASED TDM	1	1	1	X

Table 3.12: Contributions compatibility degree

TBFC+SUR, the packets set as unsafe are the ones that are routed adaptively, and thus, forwarding of those packets can be limited (restricted) by the EPC filter. Therefore, there is a direct relationship between the EPC filter and the set of paths that needs to be filtered out.

On the other hand, PROSA and TBFC+SUR and/or ECP are not 100 % fully compatible. The main reason is that PROSA is used in deterministic routing whereas TBFC+SUR and EPC is used in adaptive routing algorithms. However, we can still put them together in a complementary way. As an example, PROSA can be used to setup circuits and those circuits sets in the router. But for those packets that loose the circuit setup they can be injected in the network following the adaptive routing algorithm, possibly extended with TBFC+SUR and/or EPC filter.

Finally, Token-based TDM can be loosely complemented whit the other contributions. This could be achieved if the TBFC+SUR, EPC, and PROSA contributions are implemented at virtual network level, with one virtual network holding the traffic for a different application and the Token-based TDM working as top of a set of domains. This configuration seems, however, too complex and with no clear benefit.

Table 3.13: Digest of all proposals described in this thesis

	TBFC + SUR	EPC	PROSA	PROSA DD	Token-based TDM
Context	Throughput improvement with FA routings	Throughput improvement with FA routings	Memory Access Latency reduction	Memory Access Latency reduction	High Assurance network
Router Modifications	New Flow control(TBFC) new routing algorithm (SUR)	Registers of destinations, arbiters limit adaptivity using the registers	Arbiter, repeaters	Arbiter, repeaters	Arbiter domain set by Tokens
Dedicated network	-	-	PROSA network	PROSA network (without NACK notifications)	Token propagation network
Network Topology	2D Mesh 2D Torus	2D Mesh	2D Mesh	2D Mesh	2D Mesh
Traffic Type	Synthetic traffic	Synthetic traffic Hotspot	SPLASH PARSEC	SPLASH PARSEC	Synthetic traffic
Compared to	FA routing, DOR, FA buble	FA routing DOR	Baseline router, Dejavu	Baseline router, Dejavu	PhaseNoC

Chapter 4

Achieving Balanced Buffer Utilization with a Proper Co-design of Flow Control and Routing Algorithm

- **Authors:** Miguel Gorgues Alonso(Universitat Politècnica de València), José Flich (Universitat Politècnica de València)
- **Type:** Conference
- **Conference:** 2014 Eighth IEEE/ACM International Symposium on Networks-on-Chip (NoCS)
- **Location:** Ferrara, Italy
- **Year:** 2014
- **DOI:** 10.1109/NOCS.2014.7008758
- **URL:** http://ieeexplore.ieee.org/document/7008758/
- **Citation:** [75]

4.1 Abstract

Buffer resource minimization plays an important role to achieve power-efficient NoC designs. At the same time, advanced switching mechanisms like virtual cut-through (VCT) are appealing due to their inherited benefits (less network contention, higher throughput, and simpler broadcast implementations). Moreover, adaptive routing algorithms exploit the inherited bandwidth of the network providing higher throughput.

In this paper we propose a novel flow control mechanism, referred to as type-based flow control (TBFC), and a new adaptive routing algorithm for NoCs. First, the reduced flow control strategy allows using minimum buffer resources, while still allowing virtual cut-through switching. Then, on top of TBFC we implement the safe/unsafe routing algorithm (SUR). This algorithm allows higher performance than previous proposals as it achieves a proper balanced utilization of input port buffers. Results show the same performance of fully adaptive routing algorithms but using less resources. When resources are matched, SUR achieves up to 20% throughput improvement.

4.2 Introduction

Nowadays, there is no doubt that network-on-chip [64] has moved from concept to technology. NoCs are the natural way to allow efficient communication inside a chip in terms of performance, area, and power. NoCs replace complex all-to-all communication solutions, or simple solutions as busses or crossbars which do not have good scalability.

The NoC concept was inherited from the off-chip domain, where high-performance interconnects were designed to build large HPC installations or datacenter systems. This shift in the environment (from HPC/datacenters to chip) makes NoC design a challenge, since the engineer must face very limiting constraints in the chip design environment. Area constraints impose optimized designs trying to use as less resources as possible. This forces the engineer to very optimized resource designs. However, more important is to provide a power-efficient design. The chip power budget is highly limited and this imposes severe constraints in the resources used within the chip. Power-hungry components (mainly buffers) must be minimized if not removed at all. However, the engineer faces the problem of complying with those constraints while still delivering the expected performance. These two are usually conflicting requirements.

One clear example of this problem is the fact that engineers usually rely on wormhole switching where blocked packets keep in the network along their paths, keeping buffers in different routers. This allows buffers to be reduced in size, smaller than the packet size. However, this switching mechanism imposes large performance impact. Indeed, by blocking several routers, a packet may introduce severe congestion problems. In addition, wormhole switching imposes more architectural constraints. For instance, broadcast/-multicast operations can not be implemented unless more buffers are allocated. This is needed to avoid chances of inducing deadlock situations.

In addition, the way buffer components are managed may lead to power waste. Indeed, their use should be as balanced as possible in order to economize energy. Typically, routing algorithms rely on different virtual channels, specially in wormhole switching. Also, several virtual channels are implemented to cope with protocol-level deadlocks induced by dependencies between messages generated by higher-level coherence protocols. This leads to a large number of buffers and, thus, to a waste of resources if they are not equally balanced, which is the typical case.

In this paper we address the problem of balanced buffer utilization. In order to address this challenge we first propose a novel flow control strategy, referred to as Type-Based Flow-Control (TBFC). This mechanism is tailored to buffer resources with minimum capacity but still allowing virtual cut-through switching (thus enabling its benefits). In addition, TBFC is prepared for a new type of routing algorithms which, depending on the type of a packet may take different routing decisions. Indeed, we apply a novel adaptive routing algorithm on top of TBFC. The algorithm, referred to as Safe/Unsafe routing (SUR), classifies packets as safe of unsafe depending on the chances of packets to induce deadlock. Safe packets move through the network in an unrestricted manner, while unsafe packets are routed only through deadlock-free paths. When combined, TBFC and SUR achieve a perfect balanced utilization of resources thus achieving an optimal use. Performance results show a boost in performance when the algorithm is used in 2D torus networks. Also, performance is kept maximum in 2D mesh configurations while using less resources.

The rest of the paper is organized as follows. In Section 4.3 we describe the new flow control mechanism. Then, in Section 4.4 we describe the SUR routing algorithm working on top of the new flow control mechanism.In Section 4.5, we provide the performance evaluation and its analysis. The related work is described in Section 4.6, and the paper is concluded in Section 4.7.

4.3 TBFC Description

In this section we describe TBFC that enables balanced buffer utilization to the routing algorithm. But, before entering into details, we need to differentiate between two different crossbar switching strategies that may be implemented inside the router. The first one is termed *flit-level switching* and consists of improving buffer utilization by allowing the router to multiplex flits of different packets to advance through the crossbar while directed to the same output port, but mapped in different virtual channels. The second

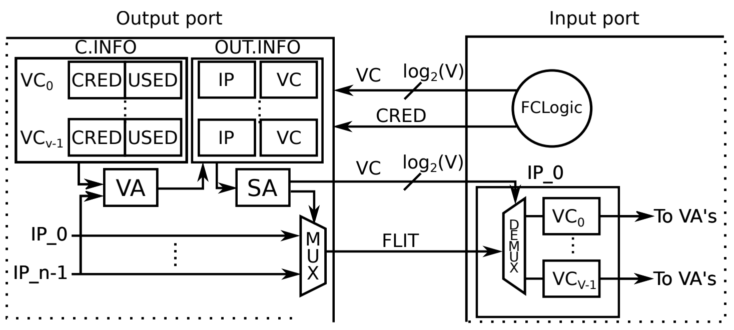


FIGURE 4.1: Traditional credit-based flow control assuming flit-level crossbar switching.

one is termed *packet-level switching* and consists in preventing the router to multiplex flits from different packets to the same output port. In this approach, when a packet header gets access to the crossbar, the remaining data of the packet will keep the crossbar connection and follow without interruption.

Flit-level switching is conceived for wormhole switching while packet-level switching is conceived for virtual cut-through. However, both approaches can be used for any switching mechanism. Nevertheless, taking flit-level or packet-level switching into account is important since it affects how the flow control can be implemented. In the next two sections we describe our flow control method for both crossbar switching strategies.

4.3.1 TBFC with Flit-Level Switching

Figure 4.1 shows a traditional credit-based flow control implementation for a pair of output-input ports. Flit-level crossbar switching is assumed. At each output port, the router needs some control information. Indeed, for each VC we need: one field for the number of credits available (*CRED*), one field to determine whether the VC is being used or not (*USED*), and the input port and VC that has this VC granted (stablishes a link between the input port and the granted VC).

When a packet header is routed, the router sends a request to the target output port. At that port, the virtual channel allocator (VA) checks whether there is any free VC that has enough credits at the next router for the whole packet (we assume virtual cut-through switching). Then, the router arbitrates (in round-robin fashion) among all the requests and assigns the VC to the winning request. It stores the winning input port and virtual channel in the control info structure associated to the VC. It also decrements the available credits in the control info associated to the VC.

At Switch Allocation (SA) stage, the arbiter selects the input port that will send a flit through the output port the next cycle. SA selects this port between the input ports assigned to this output port by the VA stage. The arbiter rotates the priorities whenever an input wins the access, thus implementing flit-level crossbar switching. The router

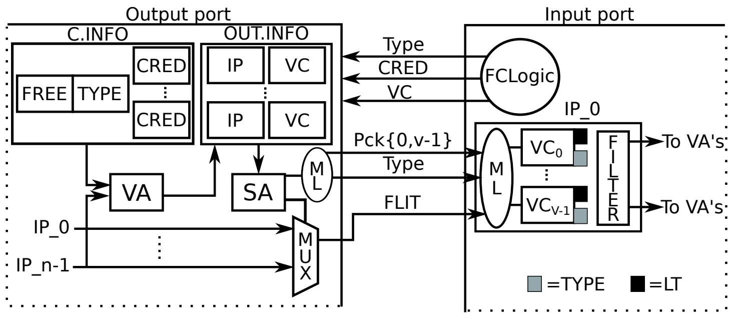


FIGURE 4.2: TBFC flow control assuming flit-level crossbar switching.

sends the flit together with the VC ID to the next router. The next router uses the VC ID to demultiplex and allocate the flit into the correct VC. When a tail flit is forwarded the VC is freed and can be assigned again to a new packet header.

At the input port, when one flit is forwarded, the Flow Control Logic (*FCLogic*) sends a credit back to the upstream router. To do this, the router needs at least $log_2\ (V)+1$ wires to indicate the VC that will receive the credit (signals *VC*), where *V* is the number of virtual channels at each input port. It also sends the control signal *CRED*. Upon reception, the credit counter associated with the VC is incremented.

Figure 4.2 shows TBFC when applied to flit-level crossbar switching. The first difference between the traditional flow control and TBFC is the flow control information structure. TBFC adds two new fields per output port: *FREE* field, which accounts for the number of available VCs, and *TYPE* field, which accounts for the number of packets stored at the input port labeled with a particular type (we will later describe the type usage in the routing algorithm). Then, for each VC, the control info keeps the *CRED* counter and the associated info for the assigned input port and VC. However, the *USED* field is removed.

Contrary to the baseline flow control, the rules (at VA stage) to assign a VC to an incoming request are different. An improvement in the VC selection was proposed in [62]. In our design, the VA stage checks only the number of free VCs and the number of labelled packets (*TYPE* field) (more details described in the next section). When one request wins the output VC, the input port of this request is assigned to the output VC. The winning input port and input VC are associated to the control info for the VC. The number of FREE VCs is decremented by one and, if the packet sent downstream is labelled, then the TYPE field is incremented by one.

At SA stage, the router selects the input port to pass through the output port and forwards the flit to the next router. At crossbar stage, the router does not send the VC selected. Instead, it performs a *packet* $\rightarrow$ *ID* mapping (*ML* block) to assign one identifier to the packet. When a head flit is sent, the router sends also the type of the packet. The identifier, the packet type and the flit are sent through the link to the next router. All the flits of the same packet will use the same identifier and only the packet header will contain the type field.

When the downstream router receives the head flit, the packet type and the identifier, a new mapping is performed (*ML* block). In this case, an $ID \rightarrow VC$ mapping is performed, thus allocating the new packet in one free VC. After the head flit, all flits that arrive with the same identifier are kept in the same VC through the mapping logic.

Each input VC has one bit associated, referred to as Last token (LT). When one head flit arrives and is allocated in one VC, this VC sets its LT bit to one and the LT bit of the other VC is reset to 0.[1] This field is used to guarantee in-order delivery of packets. Indeed, if two VCs at the same input port have a header packet with the same destination, then the oldest one (the one with LT bit set to zero) is the one to access the VA stage. Otherwise, both packets may access the VA stage. Notice that if the routing algorithm implemented on top of the flow control allows out-of-order delivery, then the LT bit and its associated logic can be removed. This will be the case of the SUR algorithm. In addition, each input VC will contain a *TYPE* bit which will indicate if the packet allocated on that VC is labelled or not. This bit is updated with the type information received when a header flit arrives.

Whenever a head flit is sent downstream, the *TYPE* bit is transmitted upstream. Upon reception, the upstream switch decreases the *TYPE* counter. In any case, the *FREE* field is increased by one. Notice also that the *CRED* field is still used in TBFC. This is needed as we are assuming flit-level crossbar switching, which may provoke different reception and transmission rates at the input ports.

4.3.2 TBFC with Packet-Level Crossbar Switching

Now, we focus on the TBFC mechanism when packet-level crossbar switching is enabled. Notice that in this case packets will not be mixed in the crossbar. This fact, together with the VCT switching we assume will guarantee that reception and transmission rates of packets at the input ports will be equal. This means that whenever a packet header wins the access to the crossbar, the whole packet can be transmitted and will not stop its transmission until reception at the downstream router. This fact simplifies greatly the TBFC mechanism, as we will see.

Figure 4.3 shows the TBFC mechanisms with packet-level crossbar switching. The first thing to notice is the simplification of the control structures. Now, we do not need credits anymore and we only need to keep which input port and VC got access to the VCs downstream through an output port. In particular, the *FREE* and *TYPE* fields

[1]If the router has more than 2 VCs, the LT field will need $log_2\ (V)$ bits and will be updated following an algorithm similar to the ones used in caches with Least Recently Used (LRU) replacement policies.

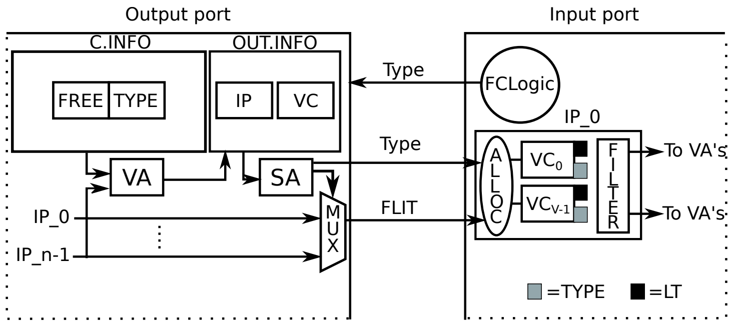


Figure 4.3: TBFC flow control assuming packet-level crossbar switching.

are still used. Also, the mapping logic blocks are removed. Indeed, when a packet gets access to the output port will be transmitted uninterruptedly.

The VA stage is not modified as it takes into account only the number of free VCs (*FREE* field) and number of packets labelled at the downstream router (*TYPE* field). The SA stage is also simplified since there is not flit multiplexing at the output. The SA stage needs only to arbitrate among competing packets but must keep the token priority fixed until the packet's tail leaves the router. This guarantees no multiplexing at the crossbar.

At the downstream input port side the logic is also simplified. There is no *ML* logic and the FCLogic only sends back upstream the type of the packet that just started to leave the input port. LT bits are still used if in-order delivery is needed to be guaranteed and the type field per VC is needed to remember whether the packet in the VC is labeled or not.

4.4 Safe/Unsafe Routing Algorithm

In this section we present the new routing algorithm adapted to the TBFC strategy. Each input port contains two VCs, while each VC is assigned a buffer to keep the whole packet. The SUR algorithm is fully adaptive and relies on an escape path to prevent deadlocks. The underline routing algorithm to implement this escape path is XY. The algorithm can work either on routers using flit-level crossbar switching or packet-level crossbar switching. SUR works on n-dimensional meshes and n-dimensional tori.

TBFC enables packet labeling and exposes this information to the routing stage. In our case, the SUR algorithm labels packets as *safe* or *unsafe*. Packets are labeled when they are sent to a downstream router as follows:

- In an n-dimensional mesh a packet is delivered and kept in the next router as a *safe* packet if the next hop conforms to the baseline routing algorithm. Otherwise, the packet is labeled as *unsafe*.

- In an n-dimensional torus a packet is delivered and labeled in the next router as *safe* if one of the following conditions is met:
  - The next hop of the packet is to traverse a wraparound link along dimension d, and the packet does not need to traverse a wraparound link with a lower dimension than d.
  - The packet does not need to traverse any wraparound link from the current router to the destination and the next hop conforms to the baseline routing.

  If any of these two conditions is not met, then, the packet is delivered and labeled as *unsafe* packet.

```
Input:
    The number of free VCs in the downstream node, $f$;
    The number of safe packets in the downstream node, $s$;
Output:
    Whether the packet can route to the downstream node;
1:  if $f > 1$ then
2:      return true;
3:  end if
4:  if $f = 1$ and $s \geq 1$ then
5:      return true;
6:  end if
7:  if $f = 1$ and $s = 0$ then
8:      and the packet will be delivered and labeled as a safe packet in the next router
9:      return true;
10: end if
11: return false;
```

Algorithm 4: check-port(f,s)

With this classification, the routing algorithm will decide which outputs ports are eligible for packets. In detail, output port along the minimal paths to destination will be eligible. *Safe* packets will be routed without any restriction and *unsafe* packets will be routed only in some particular conditions. To assist this routing algorithm we define a *check port* function suitable for meshes and tori. Algorithm 4 show the function *check-port*(f,s). This function avoids filling any input port with only *unsafe* packets. It checks, for a given input port, the number of free VCs (f) and *safe* packets (s) as follows:

- $f > 1$, the packet can be delivered because there is more than one free VC in the input port at the next router.
- $f = 1$ and $s > 0$, the packet can be delivered because there is at least one *safe* packet in the next router.

- $f = 1$ and $s = 0$, the packet can be delivered only if the packet is safe at the next router; otherwise, the packet is blocked or takes another output port.
- $f = 0$, the packet is blocked or takes another output port.

```
Input:
    coordinates of the current node $C : (c_1, c_2)$,
    coordinates of the destination $D : (d_1, d_2)$,
    free buffers: $(f_1-, f_1+, f_2-, f_2+)$,
    safe packets: $(s_1-, s_1+, s_2-, s_2+)$;
Output:
    selected output channel;
 1: S=0;ch=null;
 2: if $C == D1$ then
 3:     ch=internal; return true;
 4: end if
 5: for $i == 1$ to 2 do
 6:     if $d_i - c_i > 0$ and $check - port(f_i+, s_i+)$ then
 7:         $S =\leftarrow S \cup \{c_i+\}$;
 8:     end if
 9:     if $0 > d_i - c_i$ and check-port$(f_i-, s_i-)$ then
10:         $S \leftarrow S \cup \{c_i-\}$.
11:     end if
12: end for
13: if $S \neq \emptyset$ then
14:     $ch$ = select$(S)$;
15: end if
16: if if $S = \emptyset$ then
17:     $ch = null$;
18: end if
```

Algorithm 5: safe-unsafe-2D-meshes

The proposed fully adaptive routing algorithm in 2-D mesh is shown in Alg. 5, where f_i+, s_i+ represent the number of free VCs and *safe* packets in the input port in the neighbor router attached to the current node C along dimension i in the positive direction, respectively. Similarly, f_i- and s_i- represent the number of free VCs and *safe* packets in the input port along dimension i in the negative direction, respectively. The algorithm takes as inputs the coordinates of the current and destination nodes, number of free slot and number of safe packets of all neighboring input ports. The available channel set and the selected output channel are initialized to $\emptyset$ and *null*, respectively. If the current node is the destination, the internal channel is selected to consume the packet. Otherwise, if the offset along dimension i (dimensions 1 and 2 in Alg. 5) is greater than 0 and check-port(f_i+,s_i+) returns a true value, then the packet can be delivered along a c_i+ channel. If the offset along dimension i is less than 0 and check-port(f_i-,s_i-) returns a true value, the packet can be delivered via a c_i- channel. The check-port(f_i,s_i) function allows to add the channel c_i+ or c_i- to S if the packet can advance along dimension i.

```
Input:
    coordinates of the current node $C : (c_1, c_2)$,
    coordinates of the destination $D : (d_1, d_2)$,
    free buffers: ($f_1-$,$f_1+$, $f_2-$, $f_2+$),
    safe packets: ($s_1-$, $s_1+$, $s_2-$, $s_2+$);
Output:
    selected output channel;
 1: S=0;ch=null;
 2: if $C == D1$ then
 3:     ch=internal; return true;
 4: end if
 5: for $i == 1$ to 2 do
 6:     if $0 < d_i - c_i \leq k/2$ or $d_i - c_i < -k/2$ and $check - port(f_i+, s_i+)$ then
 7:         $S =\leftarrow S \cup \{c_i+\}$;
 8:     end if
 9:     if $d_i - c_i > k/2$ or $-k/2 \leq d_i - c_i < 0$ and check-port($f_i-$,$s_i-$) then
10:         $S \leftarrow S \cup \{c_i-\}$.
11:     end if
12: end for
13: if $S \neq \emptyset$ then
14:     $ch$ = select($S$);
15: end if
16: if if $S = \emptyset$ then
17:     $ch = null$;
18: end if
```

Algorithm 6: safe-unsafe-2D-torus

Alg. 6 presents the fully adaptive routing algorithm in 2-D torus. The difference lays in the computation of the direction to take in each dimension. Also, the *check-port* function must take into account the additional rule to define a packet as unsafe based on the crossing of wraparound links (see previous conditions).

Finally, the proposed routing algorithm randomly selects an output channel from S if it is not null. Otherwise, the packet is blocked and routed in the next cycle.

4.4.1 Deadlock-freedom property

In this subsection we demonstrate that the SUR algorithm is deadlock-free. We deduce this property using a contradiction approach. We first focus on 2-D meshes and then extended it to 2-D torus.

Let us assume we have a cycle in a 2D mesh. Such a cycle will have dependencies between $x \rightarrow y$ and $y \rightarrow x$ channels. $x \rightarrow y$ dependencies are allowed by the underlying routing algorithms but $y \rightarrow x$ dependencies are not allowed. Packets stored in an Y input port will be labeled as unsafe as they are requesting an X output port. In order to create deadlock, packets inside a cycle should not advance. This means either all the buffers are full in the cycle or the routing restrictions do not allow packets to advance. The first condition does not hold since it would mean that in the Y input port both VCs are storing unsafe packets. This can not happen since unsafe packets can be forwarded only if both VCs are available, or one VC is available and the other VC is holding a safe packet. The second condition (the routing restrictions do not allow packets to advance) does not apply neither. Indeed, if one packet is at a Y input port requesting an X output port the associated X input port will store either one, or two safe packets, or will be completely empty. In the case of storing one safe packet or being empty the unsafe packet at input Y can advance, thus no deadlock. In case of storing two safe packets, both can advance since they will always have in front of them safe packets, which potentially will move as they are using acyclic paths (conformed by safe packets using the underlying deadlock-free baseline routing algorithm). Therefore, not blocking packets in the cycle. In other words, safe packets, stored through deadlock free paths have always a reserved VC, thus always advancing. Unsafe packets can cross cycles but never filling up input buffer, thus avoiding deadlocks. Therefore, for any potential cycle, unsafe packets will never take all resources in an input buffer. They, in turn, will also advance when both VCs are available to them.

For the n-D torus case we follow a similar approach. In this case wraparound links take also an important role.

If the packet does not need to traverse any wraparound link, the packet follows the behaviour described above. So, the packet will not create a deadlock. If the packet is stored in a router connected to a wraparound link, then if the packet requests the wraparound link with the lowest dimension that the packet needs to traverse, the packet is following the baseline routing. This means that the packet will be delivered and labeled as safe. On the other hand, if the packet requests an output port connected to a wraparound link and the dimension of this wraparound link is not the lowest dimension that the packet needs to cross, then the packet will be delivered and labeled as unsafe. As we have shown before this happens only if the next input port is empty or has one free VC and the packet stored in the other is safe. So,input buffers will never fill with unsafe packets. In other words, in the case of 2D torus, all the input buffers will always allow safe packets to advance.

Let us expose an example in Figure 4.4. In this 1-D torus all routers want to send messages to a router located at two hops to their right. R0 keeps in the input port two packets from R4 with destination node R1. These two packets are safe because they arrived from to R4 crossing the wraparound link with the lowest dimension required. R1 contains two packets with destination R2 that came from R0. These two packets are safe because they do not require to cross any wraparound link and the packets follow

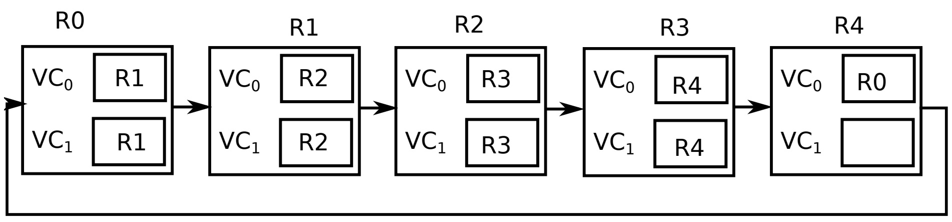


FIGURE 4.4: Example of deadlock-freedomness in a 1D ring network.

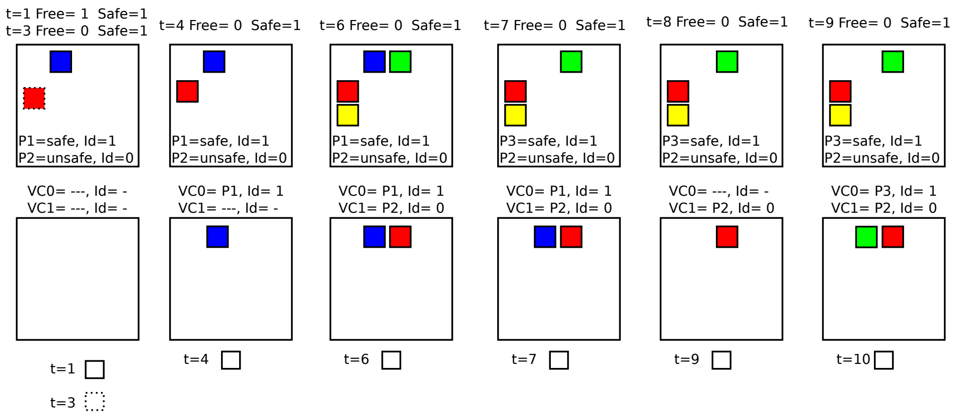


FIGURE 4.5: Logic example at the router's input port.

the baseline routing. R2 and R3 have the same situation as R1. R4 has one packet with destination to R0, this packet is unsafe because the packet comes from a router not connected to the wraparound link and needs to traverse the wraparound link to reaches its destination node. Therefore, R3 will not send another unsafe packet to R4. Then all the packets can advance because the packets at R3 can be forwarded and consumed in R4.

4.4.2 TBFC+SUR Example

Figure 4.5 shows an example of TBFC+SUR. Suppose two routers, one above the other. The north input port of the router located below is empty, so the values stored in the fields FREE and SAFE at the output port control info are 2 and 0 at router located above, respectively. Also let us assume 1 cycle of fly link. At time t=1 the header flit of the packet one (P1) with destination located at south, in the figure the blue packet, arrives to VASA stage. The arbiter selects this packet to be forwarded in the next cycle through the south output port, as this packet will be forwarded following the baseline routing this packet will sent as safe packet. The output port control is updated, FREE is decremented in 1 and SAFE is increased in 1 because the packet will be forwarded as safe packet. At t=2 the header flit is in the crossbar stage then the mapping logic calculates the ID and the router send this ID, the packet type (safe) and the flit to the next router. In this cycle the next flit of the P1 is setting to be forwarded in the next cycle. At time t=3 a new header flit arrives to VASA stage, in this case the destination of this new packet (P2),red packet, is located at south-east. The arbiter in VASA checks if this packet could win the output port, for do this the function check-port() are called. Using the south port P2 will be delivered as unsafe packet because this way do not follow the baseline routing. However, the field safe is 1, this means that in the next input port has one safe packet, so a unsafe packet can be forwarded. Then, the arbiter selects this new packet to be forwarded in the next cycle through the south output port. The output port control is updated, Free is decremented in 1, in this case the SAFE field is not incremented because the router set the packet type as unsafe.

At t=4, the header flit of P1 arrives to the input port with identifier equal to 1. Both VCs are empty, and the local variables to assign the virtual channel to an identifier are empty also. Then, the mapping logic allocates the flit to VC_0, and keeps the identifier and the packet type in the local variable. P2 is en the crossbar stage and follow same procedure as P1. At time t=5 a body flit of P1 with identifier 1 arrives. The mapping logic knows where it has to allocate the flit and puts it in VC_0. In the next cycle, t=6, packet header of P2 arrives to the input port with identifier equal to 0, then the mapping logic allocates the new message at VC_1 and keeps the identifier in the local variable. Also the packet P1 arrives to VASA stage in the second router and win the output. So the router send back the flow control information, VC FREE and the type of the packet stored in the local variable. And in the router above, two news packets arrives at the VASA stage, but the function check-port() don't allow to win this output port. Then, the tail flit of the packet P1 are selected to be forwarded in the next cycle.

At time t=7 the information of the flow control is received, so the router updates the output port control information, increasing FREE in 1 and decreasing SAFE in 1. This means that a new packet could be forwarded through of this output port. Packets P3 and P4 are waiting to be forwarded. P3 has the destination at south, and P4 at south-east. This means that the function check-port allow to P3 to be forwarded and block P4, because if P4 is forwarded both packets in the input port will be unsafe, and this could produce deadlock. Then P3 is selected to be forwarded en the next cycle as safe packet. At time t=9, the tail of the packet P1 arrives at the input port, as the header have been forwarded, then the mapping logic cleans the identifier from the local variable. If at this point a new header flit arrives, this VC can be allocated again. In the router located above, the packet P3 is at crossbar stage, and is forwarded as we explain above. At time t=10, the flit header of P3 reaches the input port and the mapping logic allocates this new message to VC_0, keeping in the local variable the identifier and the type of the message.

4.5 Performance Evaluation

Now, in this section, we perform an evaluation and analysis of our proposal. In particular, we first describe the analysis tools and simulation parameters. Then, we show the performance results for TBFC and SUR. We analyze two scenarios: a 2D mesh with 64 routers and a 2D Torus with 64 routers.

4.5.1 Analysis Tools and Parameters

The tool we use for this analysis is an event-driven cycle-accurate simulator coded in C++. The simulator allows to model any network topology and router architecture. We modeled a 4-stage pipelined router with VCs and flit-level crossbar switching. Table 4.1 shows the simulation parameters used for the 2D mesh scenario. Transient and permanent messages relate to the number of messages processed until the simulator enters the permanent state and finishes the simulation, respectively. In this scenario, we analyze three routing algorithms: deterministic routing (XY), fully adaptive routing (FA) using the typical credit-based flow control and SUR routing (SUR) with TBFC.

Parameter	FA, SUR_2VC, XY
Network topology	4x4 mesh
VCs at each input port	2
Message size, Flit size	80 bytes, 4 bytes
Queue size	20 flits
Fly link	1 cycle
Transient, Permanent msgs	10000, 10000

Table 4.1: Parameters and values used for the experiments in 2D mesh.

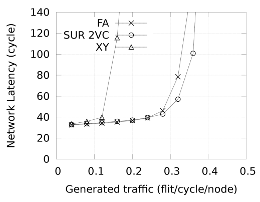
For the torus scenario, the same parameters were used except for the number of VCs and queue size at each input port. These parameters depend on the routing algorithm and flow control scheme used. Table 4.2 shows the values in the torus scenario. In torus scenario we analyze five routing algorithms: deterministic routing (XY), fully adaptive with one adaptive channel and two escape channels (FA), fully adaptive with the bubble flow control [40] (FA bubble) using one adaptive channel and one escape channel with double size (to implement the bubble). Also, we analyze SUR with two and three virtual channels (SUR 2VC and SUR 3VC). These configurations (except SUR 3VC) are the ones with minimum buffer requirements to become deadlock-free and to guarantee VCT switching.

We evaluate six traffic distributions: bit-complement, bit-reversal, transpose, perfect shuffle, uniform and hotspot. For the sake of space we only show four of them: bit-reversal, transpose, uniform, and hotspot.[2] In bit-reversal traffic, the node with binary

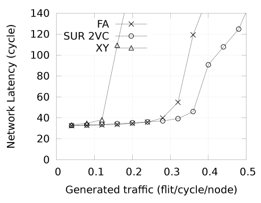
[2]We achieved similar conclusions with the not-shown traffic distributions.

Routing	VCs	Queue Size
FA_Bubble	2	20 flits adap, 40 flits esc
FA	3	20 flits
SUR_2VC	2	20 flits
SUR_3VC	3	20 flits
XY	2	20 flits

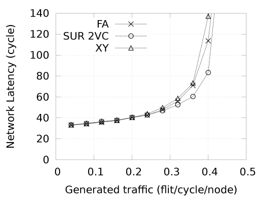
Table 4.2: Parameters and values used for the experiments in 2D torus.



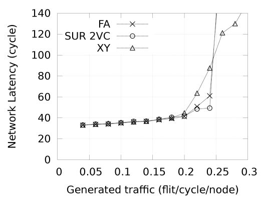
(a) latency bit reversal.



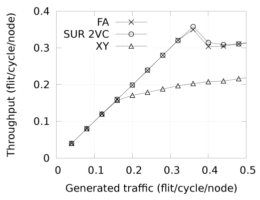
(b) latency transpose.



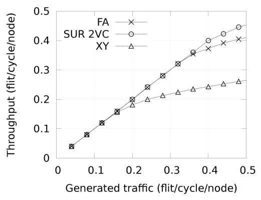
(c) latency uniform.



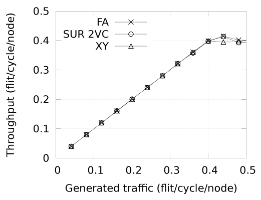
(d) latency hotspot.



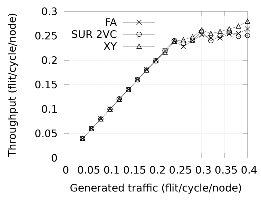
(e) throughput bit reversal.



(f) throughput transpose.



(g) throughput uniform.



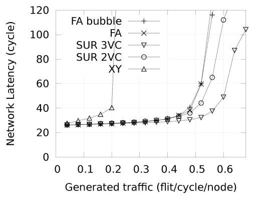
(h) throughput hotspot.

Figure 4.6: Performance evaluation in 8×8 mesh networks.

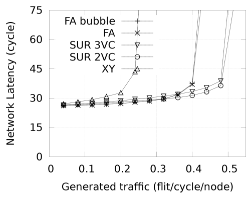
value a_{n-1}, a_{n-2},..., a_1, a_0 communicates with node a_0, a_1,..., a_{n-2}, a_{n-1}. For transpose traffic with binary value a_{n-1}, a_{n-2},..., a_1, a_0 sends packets to node $a_{n/2-1}$,... a_0, a_{n-1}, ...$a_{n/2}$. Finally, in hotspot traffic, ten randomly chosen nodes send 20% of their traffic to an specific node and the rest of traffic to any other node with equal probability. The rest of nodes keep injecting using a random uniform distribution.

4.5.2 Performance Result.

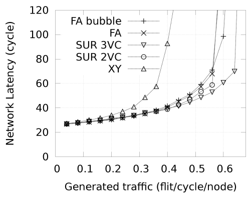
Figure 4.6 presents the results for the 2D mesh scenario. Figures 4.6a and 4.6e show the performance results for the bit-reversal traffic. In this scenario, our method reaches similar results on throughput than the ones achieved by FA. However, SUR improves latency close to saturation, when compared to FA algorithm. In any case, both adaptive algorithms (SUR and FA) outperform XY. With transpose traffic, Figures 4.6b and 4.6f, SUR outperforms FA by about 10% in network throughput. Latency is also improved by SUR when working close to saturation. For the other traffic distributions (uniform and hotspot; rest of Figure 4.6) we see similar results for the three routing algorithms. SUR, FA, and XY achieve similar network throughput. However, SUR latency is slightly improved close to network saturation.



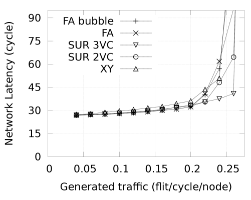
(A) latency bit reversal.



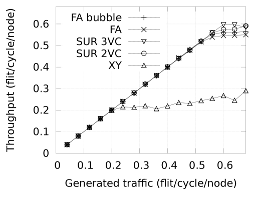
(B) latency transpose.



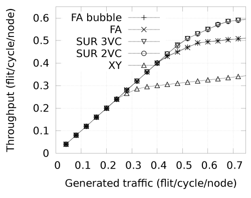
(C) latency uniform.



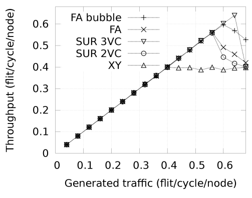
(D) latency hotspot.



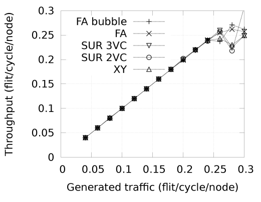
(E) throughput bit reversal.



(F) throughput transpose.



(G) throughput uniform.



(H) throughput hotspot.

FIGURE 4.7: Performance evaluation in 8 × 8 torus networks.

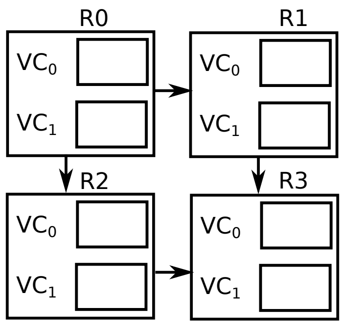
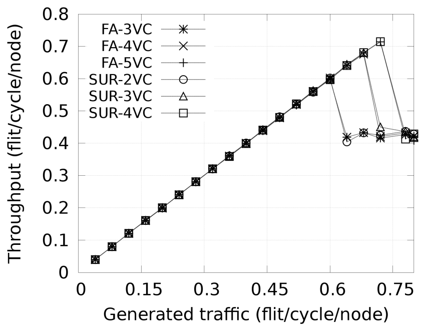


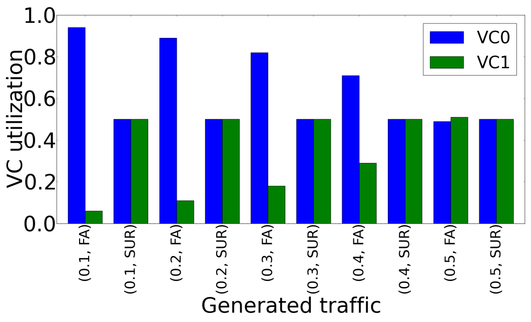
FIGURE 4.8: 2D Mesh Example.

Next, we analyze the results for the torus scenario. In this case, differences are much more significant. Figures 4.7a and 4.7e show the performance for bit-reversal traffic. SUR 2VC improves network latency achieved by FA and FA bubble. This is achieved by using less buffer resources (2VCs each with 20 slots, instead of either 3 VCs each with 20 slots or 2 VCs one with 20 slots and the other with 40 slots). Moreover, for the same number of resources, SUR 3VC works much better than FA and FA bubble on both, network latency and throughput (9% better). Also, in transpose traffic, Figures 4.7b and 4.7f, SUR routing performs much better than FA and FA bubble. In this case, both versions of SUR achieve a boost in throughput of 20% when compared to FA. Also both SUR versions perform better on network flit latency.

Finally, improvements are also achieved in uniform and hotspot. Figures 4.7c and 4.7g present the performance comparison for uniform traffic. SUR 2VC and FA perform similar on latency and throughput. SUR 3VC has the best performance on network flit latency and throughput (14% better than FA). With hotspot traffic, Figures 4.7d and 4.7h illustrate as SUR 3VC works better than FA and FA bubble on network flit latency, and all routing algorithms have similar throughput.



(A) SUR Scalability.



(B) VC utilization.

FIGURE 4.9: Scalability and VC utilization.

As we have seen in the results, SUR algorithm (together with the TBFC strategy) improve network throughput and latency over FA. In Figure 4.8 we show an example that highlights why we are achieving such improvement over FA. The Figure represents a 2x2 mesh. Assume that R0 wants to send a packet to R3. In FA, R0 can send the packet to R1 or R2. In case of R1 it can send the packet adaptively or through the escape channel (conforming to XY routing), and in case of R2 can send the packet only via the adaptive channel. Therefore, it can allocate this packet only in two VCs at R1 or in one VC in R2. The default fully routing algorithm (which promotes adaptive VCs over escape VCs) would then use only two possible VCs (one in each router). In case of an optimized FA algorithm (which gives the same priorities to adaptive and escape VCs), three VCs can be used (two in R1 and one in R2). However, in SUR algorithm safe packets can be allocated in any of the four VCs. Even unsafe packets can use any of the four VCs (taking into account there is an empty VC in the input port router). So SUR has more options to allocate the packet, allowing SUR to improve the performances obtained by FA.

Figure 4.9a shows how the benefits of TBFC+SUR scale with the number of VC. TBFC+SUR with one VC less than FA achieves the same behaviour on throughput. As can be appreciated TBFC+SUR with 3VC achieves the same maximum throughput as FA with 4 VC, and the same for TBFC+SUR with 4 VC compared with FA 5VC.

Figure 4.9b show the VC utilization at the mesh scenario, with uniform traffic. SUR with low traffic achieves a balanced use of the resources. However FA use mainly the VC0, the adaptive channel.

4.6 Related Work

In this section we describe relevant previous work about flow control and adaptive routing algorithms in NoCs. In the recent years, a lot of papers about flow control and routing for NoCs have been presented. Here, we introduce some of them.

Tang in [24] proposes a flow control in which they limit the injection rate dynamically in the network. This flow control strategy can be only used in meshes. Nousias in [25] proposed an adaptive rate control strategy in wormhole switching with virtual channels. When the contention changes, the destination node sends a signal to the source node to regulate the injection rate accordingly.

Avasare in [26] proposed a centralized end-to-end flow control for packet switching. This flow control requires two networks, the control network and the transmission data network. In [18], flit reservation control is presented. In this flow control strategy, the flit control traverses the network in advance of data flits reserving the virtual channels. After that the packet is sent to the destination node. In all the previous works, either congestion is addressed or end-to-end flow control. Also, additional structures (parallel networks) are needed. Dally and Aoki [30] described the dynamic misrouting algorithm by tagging packets based on how many misroutes they have incurred and allow any packet to request any VC as long as its not waiting for a packet with a lower dimensional reversal number. Glass and Ni in [31] proposed turn model for designing partially adaptive deadlock-free algorithms in a mesh. The west-first routing algorithm in a 2D mesh traverses the west hops first, if necessary, and then adaptively south, north and east. The negative-first routing (NFR) algorithm in a 2D mesh routes a packet first adaptively west and south, and then adaptively east and north. Chiu [32] proposed the improved partially adaptive routing algorithm odd-even turn model by constraining turns, that can introduce deadlocks, to occur in the same row or column. Wu [33] proposed a fault-tolerant odd-even turn model based routing algorithm for 2D meshes.

Dally and Seitz in [30] presented the sufficient and necessary condition for deadlock-free routing in an interconnection network. Several routing algorithms were proposed for meshes and tori [32, 34, 35]. Load-balanced, non-minimal adaptive routing algorithms for tori were proposed by Singh, *et al.* [35, 36] with three virtual channels. The method in [37] presented an adaptive minimal deadlock-free routing algorithm for 2D tori. However, the number of virtual channels required by the method was not well-controlled in [37].

Duato [38] proposed a necessary and sufficient condition for deadlock-free adaptive routing in WH-switched networks. Methodologies for design of deadlock-free adaptive routing algorithms are also presented in [39]. The adaptive bubble router [40] for VCT-switched torus is based on Duato's protocol. It requires an escape channel applied dimension-order routing (DOR) and an adaptive channel. A flow control function is added to the escape channel in order to avoid deadlocks.

In NoCs, Marculescu in [41] proposed a new routing technique (DyDA) which switches between deterministic and adaptive routing based on the network's congestion conditions. When the network is not congested DyDA router works with deterministic routing. When the network becomes congested, then DyDA router works with adaptive routing.

Ebrahimi in [42] proposed a new fully routing algorithm (DyXYZ) for 3D NoCs. In this new routing the congestion information is used as a congestion metric to select the best output port. Ma, et al. in [27] proposed a hybrid wormhole/VCT switching technique to reduce buffering while improving the performance of fully adaptive routing. Ma, et al. [28] proposed a flit bubble flow control scheme by refining the baseline bubble flo control scheme. Chen, et al. [29] proposed worm bubble flow control (WBFC), which reduces the buffer requirements and improves buffer utilization in torus networks. However, the methods in [27–29] still needs to partition virtual channels into adaptive and escape channels.

The previous flow control methods need more information control than our proposal. Some of them use circuit switching [26], so they need two different networks. Therefore, these flow control strategies use more resources than our proposal. Also, our proposal does not limit the injection rate, as happens in [24]. Finally, the SUR algorithm with only two virtual channels can provide fully adaptive routing for TBFC. Two virtual channels are not classified into escape and adaptive channels. Therefore, the buffer resources can be used more equally. Indeed, none of previous proposals focused on balancing buffer resources by co-designing the flow control and routing algorithm.

Kumar, et al. [62] present a detailed design of a network-on-chip router targeted at a 36-core shared-memory CMP system in 65nm technology with an aggressive clock frequency of 3.6GHz, thus posing tough design challenges that led to several unique circuit and microarchitectural innovations and design choices.

4.7 Conclusion

This paper presents a novel flow control Type-Based Flow-Control (TBFC) with Safe/Unsafe routing algorithm (SUR) which allows an optimized balanced buffer utilization. This is achieved because our proposal does not differentiate between VCs and does not divide the virtual channels into adaptive and escape channels. The combination of TBFC and SUR allows us to reduce the number of VCs required to implement fully adaptive routing algorithms. Sufficient simulation results were presented by comparison with the previous methods. The results showed that the proposed TBFC with SUR algorithm outperform better than the previous methods under different communication patterns.

Chapter 5

End-Point Congestion Filter for Adaptive Routing with Congestion-Insensitive Performance

- **Authors:** Miguel Gorgues (Universitat Politècnica de València), José Flich (Universitat Politècnica de València)
- **Type:** Journal
- **Journal:** IEEE Computer Architecture Letters
- **Volume:** 15
- **Issue:** 1
- **Year:** Jan.-June 2016
- **DOI:** 10.1109/LCA.2015.2429130
- **URL:** http://ieeexplore.ieee.org/document/7101264/
- **Citation:** [76]

5.1 Abstract

Interconnection networks are a critical component in most modern systems nowadays. Both off-chip networks, in HPC systems, data centers, and cloud servers, and on-chip networks, in chip multiprocessors (CMPs) and multiprocessors system-on-chip (MPSoCs), play an increasing role as their performance is vital for the performance of the whole system. One of the key components of any interconnect is the routing algorithm, which steers packets through the network. Adaptive routing algorithms have demonstrated their superior performance by maximizing network resources utilization. However, as systems increase in size (both in off-chip and on-chip), new problems emerge. One of them is congestion where traffic jams inside the network lead to low throughput and high packet latency, significantly impacting overall system performance. We propose a mechanism to eradicate this phenomena and to allow adaptive routing algorithms to achieve the expected performance even in the presence of congestion situations. End-Point Congestion Filter, EPC, detects congestion formed at the end-points of the network, and prevents the congestion from spreading through the network. Basically, EPC disables adaptivity in congested packets. Preliminary results for mid and high congestion situations show EPC is able to totally decouple congestion from routing.

5.2 Introduction

Congestion is one of the complex challenges in interconnection networks. It occurs when network resources are oversubscribed and network bandwidth is lower than the requested one. As network size increases, this effect is more apparent and problematic, both in off-chip (data centers, HPC installations, cloud servers), and in on-chip interconnects (chip multiprocessor systems; CMPs, and multiprocessor systems-on-chip; MPSoCs).

A congested situation degrades performance due to its spread over the network. Indeed, the root cause of performance degradation is the Head-of-Line (HoL) blocking caused by congested packets on non-congested ones. Packets passing through congested spots block at the head of queues, keeping resources and impeding packets not passing through those spots from advancing.

Although there are many techniques to solve the congestion problem (mainly by injection throttling or resources over provisioning), or the HoL blocking problem (resources over provisioning), they either require sophisticated implementations or exhibit reaction times dependent of network size (see Section 5.5 for an overview).

[0]This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and by FEDER funds under Grant TIN2012-38341-C04-01 and by Ayudas para Primeros Proyectos de Investigación from Universitat Politècnica de València under grant ref. 2370.

The routing algorithm is one of the key components in any interconnect. Indeed, it steers packets towards their destination and thus, can be used to avoid, or at least minimize, the congestion effects by choosing alternative congestion-free paths. Indeed, adaptive routing algorithms perform better than deterministic ones as they can adapt to the current network situations thus, avoiding congested spots. However, as shown in this paper, when congestion is severe, adaptive routing algorithms also help in spreading congestion, thus worsening the congestion problem and leading to low performance.

This paper identifies this problem and provides a compact and simple solution. A congestion filter is proposed to be used together with the adaptive routing algorithm. The filter, referred to as End-Point Congestion Filter (EPC), prevents congested packets from spreading through the network. The filter disables temporarily adaptivity for those packets that participate in a congestion situation. By doing this, congestion is prevented from spreading and taking much network resources, thus allowing the rest of non-congested packets to adapt and avoid congested ones. Results show a total decoupling of congestion from adaptive routing, thus guaranteeing no interference on network and system performance. Congested traffic can not be improved due to an excess of resources demand.

The paper is organized as follows. In Sect. 5.3 EPC is described. In Sect. 5.4 evaluation results are shown. Then, Sect. 5.5 describes previous related works. The paper concludes in Sect. 5.6 with conclusions and future work.

5.3 EPC Filter

The EPC filter works as follows. When the router has a packet (p_a) to forward, it checks whether the packet potentially contributes to a congestion situation. If the router recently forwarded a packet (p_b) with the same destination, then p_a's adaptivity is forbidden until p_b makes progress at the downstream router. Otherwise p_a is forwarded using the adaptiveness capability of the routing algorithm. p_b moving is a clear indication that packets towards that destination are not building a congestion situation. Thus, EPC enables again adaptivity for p_a. Notice that p_a, while its adaptiveness is disabled, may potentially take the same port used by p_b.

Fig. 5.1 shows the pipeline router architecture with the EPC logic. In RT, the router keeps the requested outputs following the fully adaptive routing algorithm [63]. Output port IDs (`OPs`) and destination ID (`dst`) are kept in the Control Info. In VA, the router allocates the resources (VC) to the requesting flits. In SA, the arbiter selects which input port is selected at each cycle to forward a flit through an output port. Both stages, VA and SA, are arbitrated following a round robin strategy (RR). A 3-stage arbiter (request, grant, accept) is implemented and VA and SA stages run in parallel.

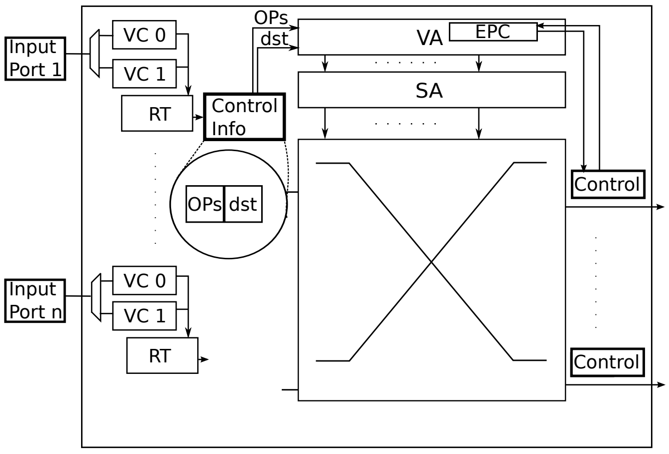
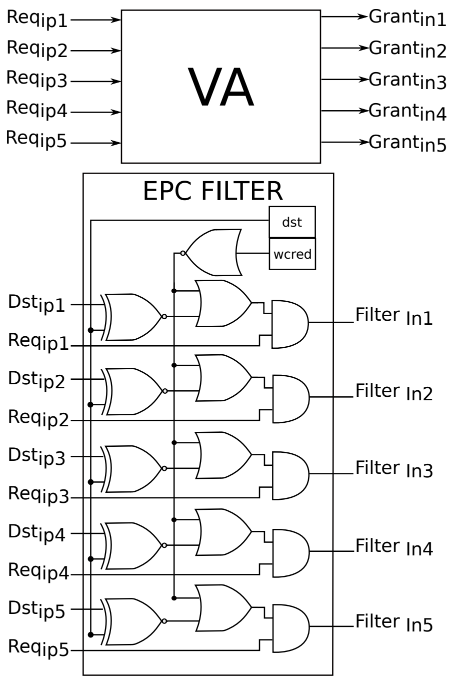


FIGURE 5.1: Baseline router architecture including EPC.

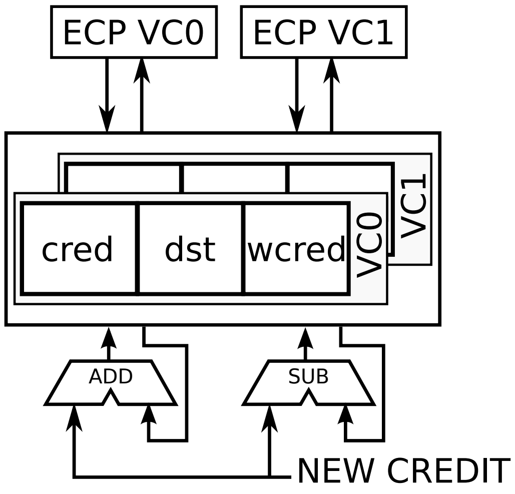
Output port selection is made on buffer occupancy. The crossbar is multiplexed (only one flit can access the crossbar at a time from each input port). Flit multiplexing is allowed (also known as wormhole flow control (WH) [64]). The router is implemented with credit flow control and Virtual Cut-Through (VCT). Notice that the mechanism can also be used in a WH network.

EPC is implemented in VA at each output port VC (Fig. 5.2a) and has to keep some info (Fig. 5.2b): the number of available credits (`cred`) located at the control information, the destination (`dst`) of last forwarded packet, and number of credits to wait in order to unlock the destination (`wcred`). When a head flit arrives to VA, it provides the `OPs` and the `dst` of the packet from routing control info. Then, EPC matches the `dst` with the one located at the output port control info with a non-zero `wcred` value. If there is no match, the flit accesses VA as usual. Otherwise, it will wait for the next cycle. To do so, the *Filter in_x* signals are sent back to the input ports and they disable the generation of accept signals in the third stage of the arbiter.

When the flit header wins one VC, the router sets the `dst` and `wcred` registers at the output port information accordingly. In `dst` the router sets the destination of the packet. The `wcred` field is updated as follows: `wcred=queue_size-cred+1`, where `queue_size` is the length of the queue in flits and `cred` is the number of credits available at the VC. This value guarantees that whenever the header leaves the downstream router the `wcred` register will reach the zero value, enabling again packet forwarding to that destination. When a credit is received, the router updates the fields, adding 1 to `cred` and subtracting 1 from `wcred`.



(A) EPC.



(B) Fully adaptive with EPC.

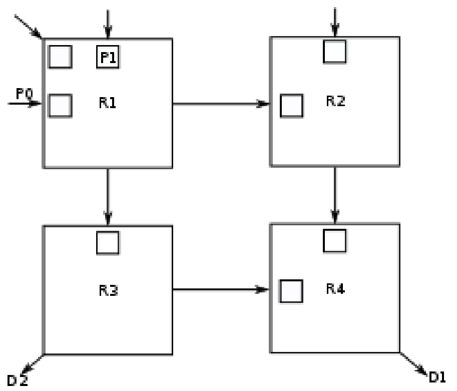
FIGURE 5.2: EPC and Output port control.

5.3.1 EPC Example

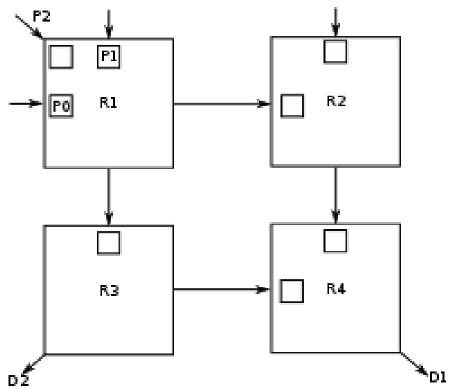
Fig. 5.3 shows an example of a 2×2 mesh assuming fully adaptive (FA) routing algorithm. At t_0, p_0 arrives to r_1 and p_1 is in the RT stage both packets have the same destination, d_1. At t_1, p_0 is in the RT stage at r_1 and p_1 competes for the outputs then p_1 is forwarded. At t_2, p_1 has been forwarded and the filter is set with $W_{cred} = 1$ and $dst = d_1$. This means that p_0 cannot get any output port until r_1 receives 1 credit from r_2. Then p_0 is blocked because the filter disable the adaptivity. At the same time, packet p_2 arrives to r_1 with destination d_2. At t_4, p_2 arrives to the VA/SA stage and wins the south output port. Finally, at t_5, p_2 is forwarded and r_1 sets the filter in the south output port with $W_{cred} = 1$ and $dst = d_2$. p_0 will be routed once the filter at the east output port is removed, meaning that p_1 is leaving router r_2 and, thus, experiencing no congestion.

5.3.2 EPC Application Scenario Examples

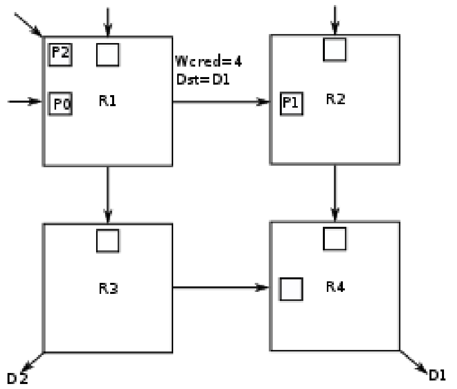
Fig. 5.4a shows an scenario in which EPC can be useful. Node D requests a cache refill using fully adaptive routing. The node sends the requests to the four memory controllers (MCs), and then, the MCs send data to D. In this case, when XY paths get congested, the routing algorithm starts to use alternative paths spreading this way the congestion over the network. With EPC, only XY paths get congested. Another useful case for



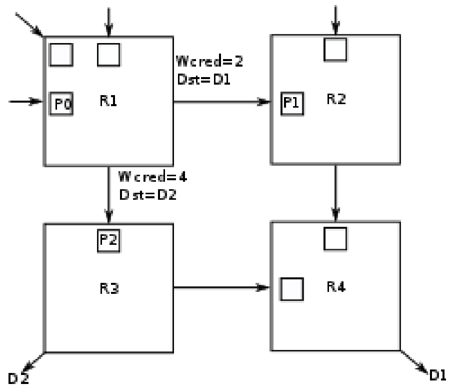
(A) t_0; p_0 arrives to r_1 and p_1 at RT stage with same destination d_1.



(B) t_1; p_0 in RT and p_1 at VASA stage in r_1.



(C) t_2; p_0 blocked and p_2 arrives.



(D) t=5; P2 advances.

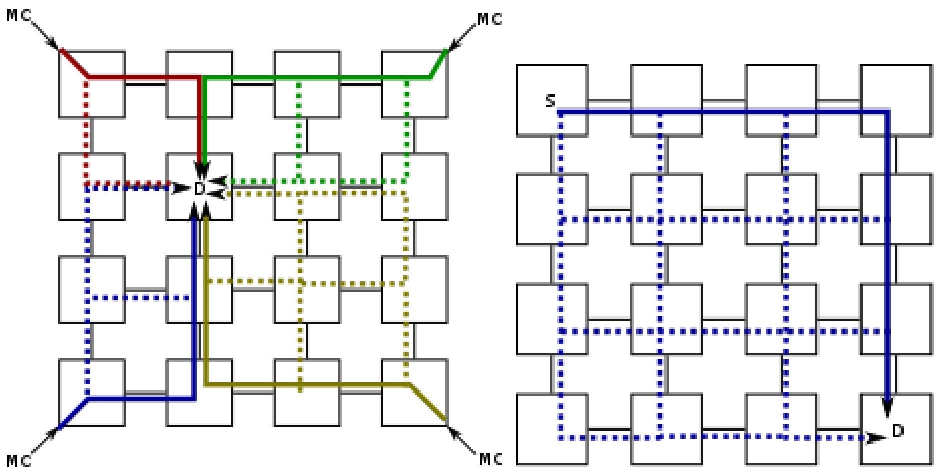
FIGURE 5.3: EPC filter walk-through example.

EPC can be seen in Fig. 5.4b where bursty traffic from S to D is injected, potentially by a multimedia application.

5.3.3 Switching, VC, and Routing Impact

The previous conditions are set for a VCT router. For a WH router those conditions would slightly differ, being the *wcred* count dependent of message size instead of queue size. Indeed, in WH what is difficult to achieve is a clear detection of the congestion situation to disable adaptiveness. However, HOL blocking will be more pronounced due to the typical less buffering exercised.

On the other hand, the more VCs implemented the less HOL blocking will be seen, thus less congestion effect (making the baseline case perform better). However, if the same amount of traffic is sent to the same destination, regardless of the number of VCs, the same degree of congestion will be seen, thus, having the same impact on the network. Moreover, VCs are typically used to classify traffic, thus, congestion can occur in isolated traffic classes, at each VC. Thus, the effectiveness of EPC will be lower as network has more buffer capacity and congestion is less severe. But for a declared congestion situation



(A) cache refill. (B) Burst traffic.

FIGURE 5.4: EPC application scenarios.

EPC will help. Related to routing, EPC does not change the routing algorithm, and is orthogonal to the number of VCs. Therefore, is orthogonal to deadlock conditions, at network and protocol level.

EPC does not isolate congested packets in separate VCs. In the presence of several hotspots, and two packets addressed to different hotspots reach one router, both will be requesting the same resource (VCs) their previous counterparts (previous packets) were using, thus, those packets will not spread congestion. Indeed, VC strategies for congested packets are orthogonal to EPC.

5.3.4 EPC Overhead Comparison

EPC is compared against two congestion control techniques, ICARO [65] and RCA [49]. Both need more resources than EPC. ICARO needs an extra Virtual Network for bursty traffic and a Dedicated Signaling Network to notify the hotspot situation to end nodes. In addition, it needs at each node two vectors with length equal to the number of nodes and some logic to manage bursty traffic. RCA needs a low bandwidth monitoring network to propagate the congestion information. RCA routers need two extra modules per port for aggregating and propagating the congestion information. RCA needs also Congestion Value Registers (CVR). Notice that the notification network is not required in EPC, thus exhibiting lower area overheads. Note that EPC just prevents congestion ramification by the adaptive routing algorithm, thus being complementary to ICARO/RCA.

5.4 Performance Evaluation

This section presents an evaluation and analysis of the EPC filter, first describing the analysis tools and simulation parameters and then, analyzing the performance results for three configurations. In the first two, we use deterministic (XY) and fully adaptive (FA) routing algorithms, in both cases without EPC. FA uses two VCs (one for adaptive and one for escape paths[63]). In the third one, we use FA with EPC (FA-EPC).

5.4.1 Analysis Tools and Parameters

We model a 4×4 mesh with 2 VCs per port, virtual cut-through switching, 4-stage pipeline routers, and 16-byte message size, 4-byte flit size, and queues with 4 flit slots. XY, FA and FA with EPC (FA-EPC) is modeled.

Three scenarios are analyzed. The first one (UNIF) refers when a uniform random distribution is used and no congestion in the network is produced. The second one (C_LOW) refers when there is a small congestion spot in the network (background traffic). In particular, eight nodes (selected randomly) send traffic to node 11 with a 30% probability (the rest of traffic is uniformly distributed). The third one (C_HIGH) refers when the hotspot probability is increased to 70%.

Results for two types of traffic are shown. The first one is for the uniform (foreground) traffic and the second one is for the hotspot (background) traffic. For this, the first 10000 packets generated after the stable network state has been reached (after initial 100000 packets reached destination) are labelled. We take into account only those packets at reception for statistics purposes. Doing this, the traffic distribution is kept the same both, at generation and at reception time, thus ensuring traffic distribution is not modified by the congestion situation.

5.4.2 Performance Results

Figs. 5.5a and 5.5b show results for foreground (uniform) traffic in C_LOW scenario. End to end latency in FA-EPC is up to four times lower than the one achieved by FA. As seen, flit latency for FA has a sharp increase around 0.3 flit/cycle/node injection rate. This is when congestion affects the uniform traffic. Also, FA-EPC slightly improves the results achieved by XY. However, in FA-EPC and XY, latency increases linearly due to the increase of foreground traffic only. This difference in latency is produced because the hotspot spreads through the network when no EPC filter is used with FA. Regarding network throughput for uniform traffic (Fig. 5.5b), FA reaches saturation at 0.25 flits/cycle/node, whereas EPC reaches 0.32 flits/cycle/node injection rate. Notice that even XY behaves better than FA. For the hotspot traffic (Figs. 5.5c and 5.5d)

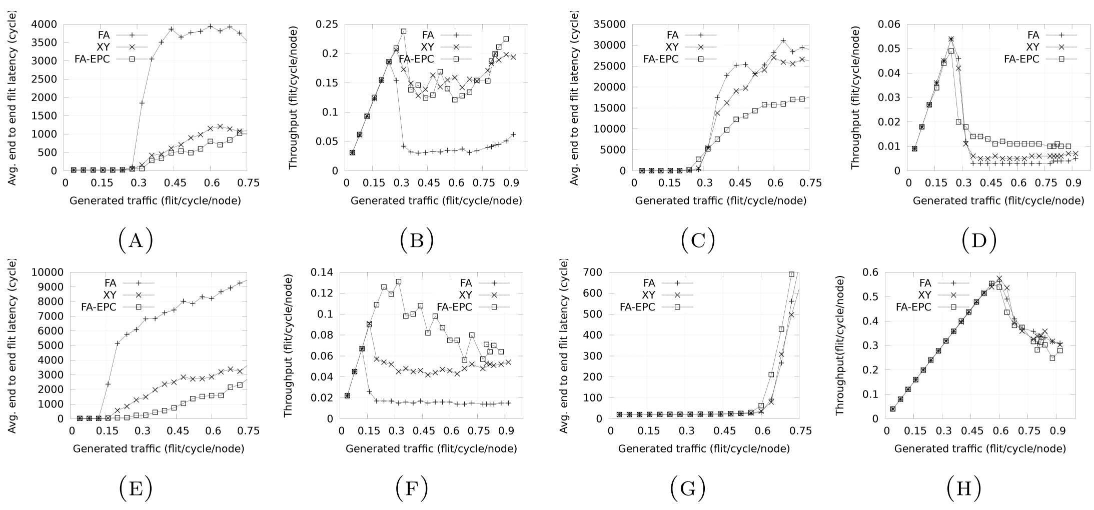


FIGURE 5.5: EPC results. C_LOW (abcd), C_HIGH (ef), UNIF (gh). Uniform traffic (abefgh), background traffic (cd).

FA-EPC keeps packet latency lower (50 percent lower) during the network congestion situation (beyond network saturation point).

The impact of EPC in C_HIGH is higher (Fig. 5.5e). Foreground end to end latency in FA becomes up to five times higher than the one achieved by FA-EPC. FA throughput (Fig. 5.5f) is doubled when using EPC. The hotspot traffic achieves a very similar behavior. Finally, in UNIF, the EPC filter has about an 8% end to end latency overhead, as the filter without congestion situation may also block non-congested packets temporarily. Network throughput, however, is roughly the same for the different routings. This small overhead in uniform traffic suggests us to use the EPC filter in very specialized scenarios, and possibly dynamically.

One interesting comparison comes from Figures 5.5e and 5.5g. Comparing the latency of background traffic for C_HIGH and the packet latency for uniform traffic (without hotspot). As shown, packet latency is very similar for both cases. This means that congestion traffic effects are decoupled from background traffic.

5.5 Related Work

Congestion can be addressed in different ways. First, congestion avoidance techniques guarantee congestion never builds in the network (e.g. ATM networks [44]). However, this leads to low network throughput and utilization. The second approach, is to detect congestion, notifying the sources, and removing the congestion by injection throttling [45]. Although it is effective in some scenarios, effectiveness of these techniques depend on the network bandwidth and network size as they rely on a closed control loop approach. The third approach is to attack directly to the side effects of congestion: HoL blocking [46]. In this case, mechanisms detect congestion and dynamically allocate new queues to isolate congested packets. One clear example of this approach is RECN [6]. In these cases, however, the implementation overhead is non-negligible. Another example of this approach is speculative reservation [47], which provides end to end flow control in order to alleviate the congestion using different VCs with different priorities, first sending the speculative packet and some flits with high priority and the rest of the packets with low priority. VOQ solutions [48] statically separate traffic, thus may alleviate congestion. However, they impede the use of adaptive routing. Finally, the last approach is to use adaptive routing algorithms to circumvent congestion spots [49]. However, the effectiveness of such methods depend on the severity of congestion since the adaptive routing algorithm may also spread congestion over the network, thus worsening the situation. As shown, the EPC filter is a good complement to those solutions.

5.6 Conclusions

The EPC filter can help to manage and prevent spreading congestion within the network when adaptive routing is used. A router with EPC significantly reduces network latency and reaches higher throughput. We plant to evolve EPC as a dynamic mechanism activated only when congestion is persistent. Also, an accurate implementation analysis will be developed.

Chapter 6

PROSA: Protocol-Driven NoC Architecture

- **Authors:** Miguel Gorgues (Universitat Politècnica de València), José Flich (Universitat Politècnica de València)
- **Type:** Conference
- **Conference:** Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)
- **Location:** Nara, Japan
- **Year:** 2016
- **DOI:** 10.1109/NOCS.2016.7579320
- **URL:** http://ieeexplore.ieee.org/document/7579320/
- **Citation:** [77]

6.1 Abstract

Nowadays chip multiprocessors tend to have an increasing number of cores, usually implementing a distributed shared last level cache. The network on chip (NoC) is in charge of interconnecting the cores, memory controller(s) and cache banks, largely impacting memory access latency. Packet switching in usually used in NoCs, but circuit switching may achieve better performance if the setup time of the circuit is shadowed (established before it is needed). In this paper we propose PROSA, a novel NoC architecture to improve memory access latency by using circuits. In PROSA, the coherence protocol steers the circuit establishment logic in order to setup circuits before needed and only for the time frame they are required. Also, a memory latency control unit (MLCU), implemented in the memory controller, assists PROSA by computing arrival time of memory blocks. A clustered router approach is followed where groups of routers are combined and attached to a PROSA circuit controller. We detail all the implementation issues of PROSA, including circuit establishment logic with acknowledgment messages, protocol modifications, and router modifications to setup circuits when required. Results from real applications demonstrate reduction of network flit latency by 34% which translates into a reduction of miss load (and store) latency of 21% (in 64-core systems). PROSA needs 9.66% more area, but reduces power by 3%.

6.2 Introduction

Chip Multiprocessor systems (CMPs) rely on networks-on-chip (NoCs) [78] to achieve fast communication between resources, mainly processors, memory caches and memory controllers. It is by no doubt that NoCs play a vital role on CMPs performance, mainly as they lay along the critical path of communication, and thus, affect process communication latencies and most important, memory access latency.

During more than one decade, NoCs have been researched (not only for CMPs) with two main goals in mind, network throughput and network latency. While network throughput is an important aspect of the NoC, in CMPs the average load of links found for accepted benchmarks (PARSEC, SPLASH, ...) is typically low. This means latency becomes the significant metric when dealing with NoCs for CMP systems. Indeed, a memory transaction is started when one processor requests a load or a write on its private L1 data cache, and finishes when the data is delivered. The NoC is used to send request commands and data between cache levels and the memory controller (MC). Therefore, the NoC plays a vital role as the memory request gets blocked until data is delivered.

[0]This work was supported by the Spanish Ministerio de Economía y Competitividad (MINECO) and by FEDER funds under Grant TIN2015-66972-C5-1-R

In this paper we address the issue of latency reduction of memory transactions. We propose PROSA, a PROtocol-oriented circuit Switch Architecture, which co-designs both the NoC and the coherence protocol and put them to play together. Our approach enhances the NoC with a new clustered component, the PROSA controller (PC), which is in charge of managing circuits and to resolve any possible conflict. The controller is in charge of four NoC PROSA routers (PR) and steers their local circuits when needed.

Results show that PROSA outperforms the baseline design by reducing network latency by about 34% on average. PROSA reduces the load and store miss latency by 21%. PROSA correctly sets up 90% of the circuits ahead of time. The PROSA cluster takes 9.66% more area but saves more than 3% in power consumption.

PROSA differs from previous circuit-switching methods by means of the feedback provided by the coherence protocol. Although some previous works also combined protocols and NoC designs [7, 10–12, 16], our proposal lets the coherence protocol to steer the network in programming circuits before they are needed, hiding the set up circuit delay. Moreover, PROSA does not need extra buffer resources at the router. Just a pair of multiplexer/demultiplexer is needed on each bidirectional router port. Section 6.6 describes related work.

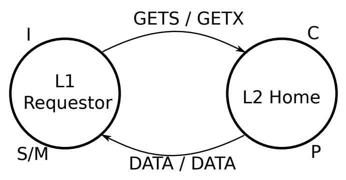
The rest of the paper is organized as follows. In Section 6.3, we describe and analyze the coherence protocol. This will center our proposal and will provide justification of its need. In Section 6.4 we describe PROSA. In Section 6.5, we provide the evaluation and its analysis. Related work is described in Section 6.6 and the paper is concluded in Section 6.7.

6.3 Coherence Protocol Analysis

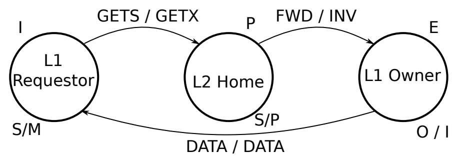
We assume the CMP uses private L1 caches for each core and shared but distributed L2 banks along a tile-based organization. Two memory controllers (MC) are placed on each top corner of the system. A 4×4 mesh NoC uses a 4-stage pipelined 7-radix router design (four ports to connect to neighbor routers and three to connect L1, L2, and MC).

The system implements the MOESI protocol [53] at L1 while at L2 blocks can be in P (private), S (shared), C (cached), or I (Invalid) state. In C mode no L1 cache has a copy of the block. Inclusive caches are assumed and a write back policy used. A static mapping policy of blocks to L2 banks is assumed.

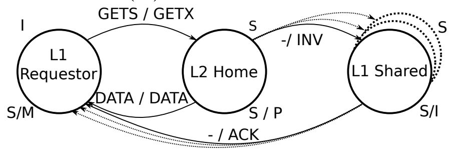
Figure 6.1 shows the simplified finite state machine (FSM) related to load and store operations. In the figure nodes are represented by circles. The current block state is represented with text placed just above the circle while the new state after the transaction is represented under the circle. Messages sent between nodes are represented by arrows. Messages for load and store operations are combined (e.g. GETS/GETX).



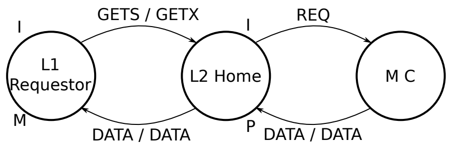
(A) L2 in state C.



(B) L2 in state P



(C) L2 in state S.



(D) L2 in state I

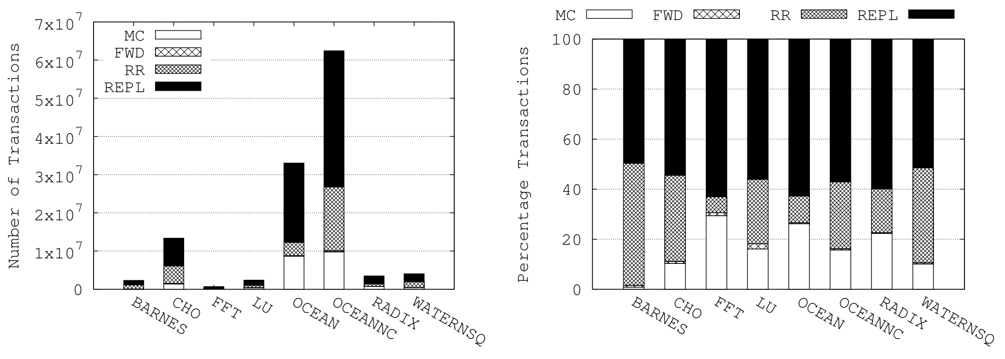
FIGURE 6.1: Coherence protocol transactions.

Whenever a L1 load miss occurs, a GETS message is sent to the L2 Home bank. Based on the block state at L2 different actions are performed. If the block is in S or C state (Figures 6.1a and 6.1c), the L2 sends the data to the L1 requestor. If the block is in P state (Figure 6.1b), the L2 sends a forward (FWD) message to the L1 with the block and the block state at L2 changes to S. When the FWD message arrives, the L1 cache sends the data to the L1 requestor cache and the block state changes O. Finally, if at the L2 the block is in I state (meaning a miss occurs, Figure 6.1d), a REQ message is forwarded to the MC. The L2 receives the data and forwards it to the L1 cache requestor. The state of the block is set to P in the L2 and to E in the L1.

Whenever a L1 store miss occurs, a GETX message is sent to the L2 Home bank. Similarly, if the block is in C state (Figure 6.1a), the L2 sends the data to the requestor and changes the block state to P. A different case occurs when the block is in P state (Figure 6.1b). The L2 sends an invalidation message (INV) to the L1 owner cache. When the INV message arrives, the data is sent to the L1 requestor and the block changes from E to I. When the block is in S state (Figure 6.1c), the L2 sends the data to the L1 requestor and sends INV messages to all the L1 sharers. When each L1 sharer receives the INV message, it sends and ACK message to the requestor and changes the block state to I. Finally, if the block is in I state (miss) (Figure 6.1d), then a REQ message is forwarded to the MC. Once the data is received, it is forwarded to the L1. The block is put in P state in L2 and in M state in L1.

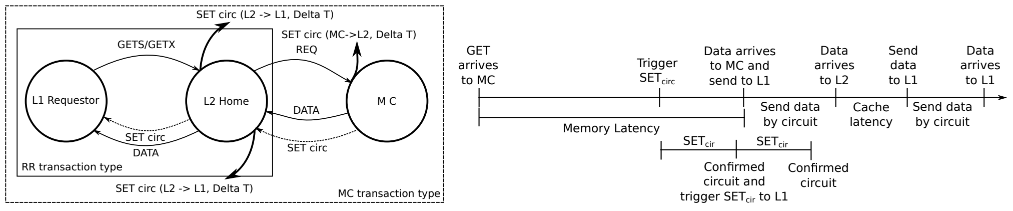
As we see, the protocol faces mainly four possible paths depending on the type of operation (load/store), L1 access type (miss/hit) and L2 access (miss/hit). Whenever a miss occurs at L1 and L2 the NoC gets involved and the memory transaction latency is increased. Factors than may affect significantly memory latency are the distance between the L1 requestor and the L2 Home bank, distance between the L2 Home bank and the MC, and the network congestion.

To better analyze these effects, Figure 6.2a shows memory transactions classified by



(A) total transac. (B) total transac (%)

FIGURE 6.2: Memory transactions types for different SPLASH-2 applications.



(A) Transaction. (B) Cronogram

FIGURE 6.3: PROSA new actions triggered by the coherence protocol and circuit establishment time line for MC transaction.

type: MC transactions (due to miss both in L1 and L2 banks), RR transactions (request-response due to an L1 miss and an L2 hit to a clean block), FWD transactions (forward transactions due to an L1 miss and an L2 forward to the owner) and REPL transactions (replacement transactions due to L2 being full). Different SPLASH-2 applications are run in a system described in Section 6.5.

First thing to note is the different amounts of transactions between applications. This depends on the complexities of the applications. However, when we normalize the numbers, Figure 6.2b, we see similarities. A high percentage of REPL transactions exist in all applications (more than 50% of the transactions). This is mainly due to the L2 cache size and the dataset of the applications and these transactions can not be predicted from the coherence protocol point of view. Noteworthy, we can observe the small percentage of FWD transactions. All applications exhibit less than 3% of this type, and for OCEANNC and RADIX lower than 1%. This indicates shared blocks mainly follow a multiple sharers pattern and not a producer-consumer pattern. Now, we focus on a more interesting transaction type. Some applications (FFT and RADIX) trigger a large percentage of MC transactions while others (BARNES and WATERNSQ) have a marginal percentage. This difference is due to the L2 hit rate and the memory footprint of the application. Finally, RR transactions are highly representative in some applications (BARNES and WATERNSQ) while minimal in others (FFT). This is due again to the hit rate in L2 and the memory footprint. Thus, applications either have large percentage of MC transactions or large percentage of RR transactions. In both cases, MC and RR, traffic needs between L2 and MC, and between L2 and L1 (requestor) can be predicted. PROSA will exploit this fact by using circuits for MC and RR transactions.

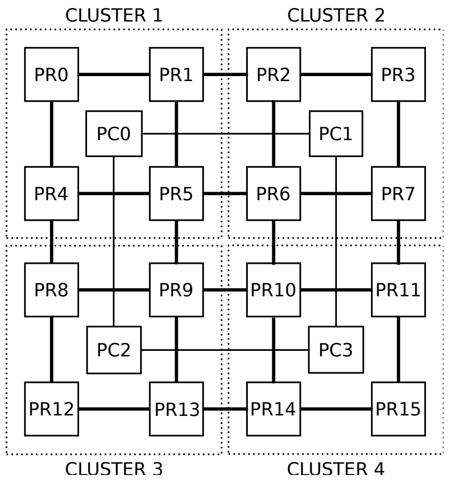
6.4 PROSA

PROSA sets up circuits between MC-L2 and L2-L1 before they are needed. First, we show the modifications performed in the coherence protocol and then, show the modifications in the NoC (the PROSA controller and PROSA router) and MC (the Memory Latency Control Unit, MLCU).

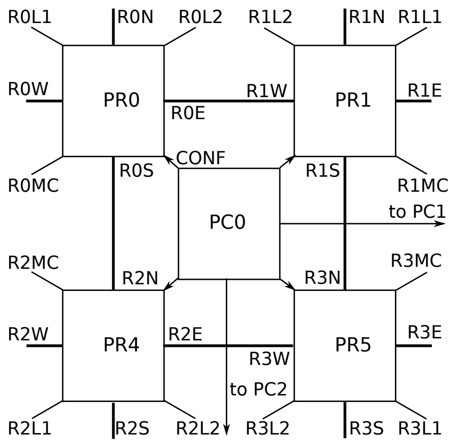
6.4.1 PROSA Coherence Protocol

In order to program circuits, we add a new action termed SET_{CIRC}. This action involves the source and destination of the circuit, and the time period the circuit will be needed (Delta T). The action is triggered by the coherence protocol in MC and RR transactions (see Figure 6.3a). In MC transactions, whenever a request is received by the MC, a Memory Latency Control Unit (MLCU; described in Section 6.4.4) predicts when the data will arrive to the MC. Based on this delay, the MC triggers a SET_{CIRC} action after a delay period (DP). DP equals to the predicted memory latency minus the circuit setup period (CSP). The SET_{CIRC} action sets the circuit. By the time the block arrives to the MC the circuit has been set and is kept during the time the data will be transmitted. When SET_{CIRC} arrives to L2, the circuit is confirmed but in parallel a new SET_{CIRC} between L2 and L1 is triggered. Thus, when the data arrives to L2, after accessing the L2, the block is sent also to L1 using a circuit. These two circuits are predictable circuits in the sense they will be necessary regardless of network status. Figure 6.3b shows the timing of both circuits.

In RR transactions (between L1 cache and L2), whenever a GETS/GETX message is received, a SET_{CIRC} action is triggered between the L2 and the L1. Again, the connection will be set only for the time period the data is available. However, contrary to the two previous triggered circuit scenarios (in MC transactions) now this circuit is speculative, in the sense the circuit may not be needed if the block is in P or I state. Therefore, for this type of circuit we will need to automatically and efficiently tear down the circuit when transmission is finished or when it is clear the circuit is not needed.This modification does not introduce out of order delivery, because for the same transaction the protocol never sends two messages to the same destination.



(A) Prosa Network.



(B) Prosa Cluster

FIGURE 6.4: PROSA controller, detailed components.

6.4.2 PROSA Circuit Network

PROSA follows a clustered approach where one PROSA Controller (PC) controls four routers (Figure 6.4a). When the SET_{CIRC} action is triggered in an L2 or MC, a request circuit (RC) message is forwarded to the local PC. If the RC wins the needed resources in the cluster, it advances to the next PC along the path. If not, a NACK response is sent to the source of the RC message. When one RC message reaches the PC that controls the destination node and wins the resources, then an ACK response is sent back to the source via the PC network. Note that routers are not affected by RC messages,nor ACKs neither NACKs. All this traffic travels via PCs.

On top of Figure 6.5, we show the RC message is composed by eight fields. P refers to the input port from which the RC arrives to the cluster. *src* and *dst* are the source and destination for the circuit. *Delta T* and *Delta T'* carry the number of cycles to wait until the circuit will be established and torn down, respectively. *Type* carries the type of the RC message (REQUEST, ACK, NACK). *ID* identifies the circuit inside the network, and finally, *GT*, Golden Token, refers to the distance between *src* and *dst*. This field is used in the PC to assign priorities between requests.

Below the RC structure, in Figure 6.5, we show the Resource Arbiter (RA) that controls whether a specific resource can be reserved for a particular period of time. A PC consists of several RA modules, one RA per output port in the cluster controlled by the PC. In total, each cluster has twenty eight RA, all shown in Figure 6.6 (e.g. R_{0S} corresponds to the south output port at the north-west router in the cluster). An RA module arbiters between some requests, the number of requests depends on the number of inputs ports that have dependencies with the resource (the output port). This depends on the routing algorithm, in our case we assume DOR routing.

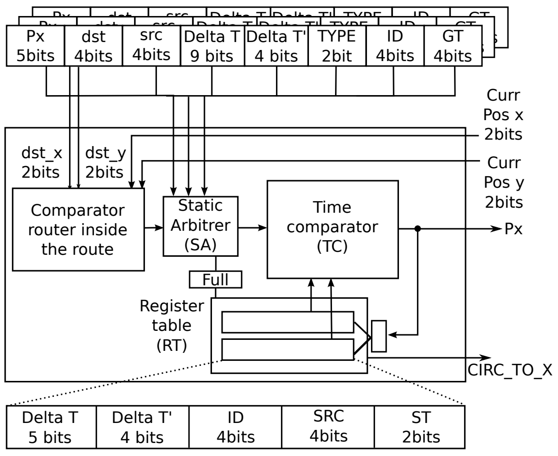


FIGURE 6.5: Resource.

When an RC message arrives to RA, the *dst* field is checked to decide whether the output port controlled by the RA is along the path between *src* and *dst.* If so, the RC goes to the Static Arbiter (SA) which arbiters between the incoming requests at this point. SA gives priority to requests with larger values in the *GT* field (longer paths have priority). Then, RC advances to the Time Comparator (TC) module where it checks that the request does not overlap in time with any previously programmed circuit. Finally, if there is no conflict with programmed circuits then the RC is stored in the Register Table and forwarded along the PC network. Notice that only one RC message gets access to the table. As we will see, this does not impact performance and reduces complexity.

The register table (RT) keeps circuits information. For each circuit the following fields are stored: *Delta T*, *Delta T'*, *ID*, *src*, and *ST*. When an ordinary request wins the resource, *Delta T*, *Delta T'*, *ID*, *src* get the values from the RC message. *ST* keeps the state of the circuit, which can be *unconfirmed*, *confirmed*, or *empty*. *Delta T* is decremented by one every cycle. When it reaches zero, the RA module activates the outputs *Cir_to_P* which control the circuit establishment in the PROSA router (described later). *Delta T'* is also decremented by one every cycle until it reaches zero. When *Delta T'* arrives to zero the signals and the register are cleared.

When an ACK or a NACK arrives to RA, it follows the same path than an ordinary request, with some small differences. First, the RC message advances to SA. ACK/NACK messages have higher priority than ordinary requests. By construction we guarantee only one ACK/NACK enters one PC controller each cycle, thus they will always win the SA access. Then, in TC the RC message checks the information stored in the table. An ACK consolidates the stored information while a NACK removes it. Then, the RC message is forwarded (to the next RA along the path inside the cluster or to the next PC).

The PC controller is shown in Figure 6.6. It contains 28 RAs, 7 per router. For the sake of simplification, we group all outputs of a router (L1, L2, MC) into a single RA (R_{xL}), thus

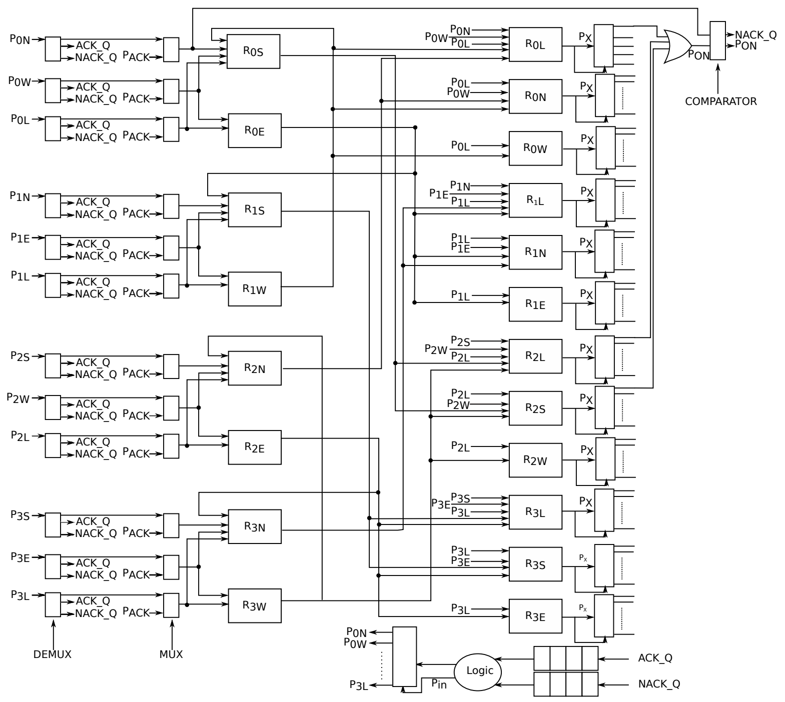


FIGURE 6.6: PROSA Controller.

showing only twenty RAs. The PC controller implements two queues to store generated and incoming ACK/NACK messages. RC requests can be dropped. A demultiplexer located at the input port (left-hand side) separates requests from ACK/NACK messages (*type* field of the RC message is used as selector). When a request arrives, it continues through the PC. However, if the incoming request is an ACK/NACK, it is sent to the corresponding ACK or NACK queue. The next stage in PC is a multiplexer, which multiplexes between the incoming RC message and queued ACK/NACKs, giving priority to ACK/NACKs. In case of conflict the request is discarded (generating a NACK message which will be queued). Finally, the selected message enters the RA tree.

RA modules are linked following the resources dependencies imposed by the routing algorithm (DOR in our case). An example of linked arbiters are $R_{0E} \rightarrow R_{1S} \rightarrow R_{3S}$ for messages coming through router PR0 and leading to a destination below router PR3. An RA module has several input and output dependencies as multiple routing combinations exist. Along this path, if a request wins all the required RA modules (from left to right) the request will be forwarded to the next PC, or will generate an ACK if the destination is controlled by the current PC. A comparator at the output of the PC module (right-hand side) checks whether one RC message won all the required resources (it was at the input side and also succeeds at the output side). In not, a NACK message is triggered and stored in the NACK queue.

Figure 6.6 shows the ACK and NACK queues and its control logic block. The logic guarantees only one ACK/NACK message will access the PC controller each cycle, ensuring that they will always win all RAs. NACK messages have higher priority than ACK

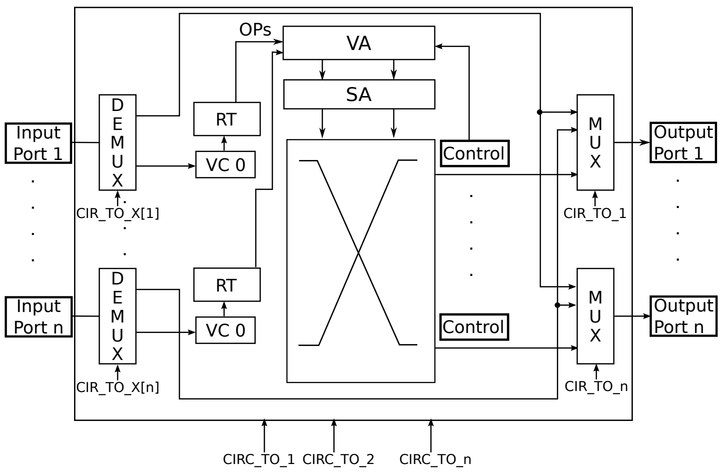


FIGURE 6.7: PROSA router.

messages. Moreover, the logic changes the input port of the ACK response, computing the input port of the ordinary request corresponding to this ACK response. Thus, ACK messages cross the same RA path used by the associated request messages. Notice that NACK generated messages locally in a PC controller need to be re-injected in the RA tree again to remove all the reserved resources, and then sent back to the previous PC controller.

6.4.3 PROSA Router

Figure 6.7 shows the modifications performed in the baseline router. We only add one demultiplexer per input port, one multiplexer per output port, connections between those elements and an asynchronous repeater per output port. The repeater allows to forward the flit very fast reducing wire delay, as is used in SMART [7]. This technology allows one flit to cross all the network between source and destination in one cycle.

The router works as usual until signals *Circ_to_X* are activated, where *X* is the output port ($L1$, $L2$, MC, N, E, W, S). In this situation, the corresponding input and output ports are switched in and the arrived flits are blindly forwarded through them. At the same time, VA and SA arbiters associated to the output port are disabled. Notice that the circuits won't use buffer resources, therefore circuits cannot introduce deadlocks.

Circ_to_X signals are generated by the PC controller, each generated by a single RA module (the one that manages the output port). *Circ_to_X* indicates the input port that has to be switched in to the 'X' output port. One wire is used for each possible input port. Thus, the demultiplexer at an input port is selected from the ORing of *Circ_to_X* signals, and the multiplexer at an output port is selected from the ORing of all the wires from the associated *Circ_to_X* signal.

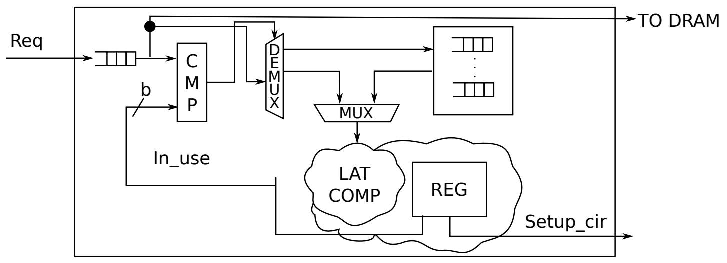


FIGURE 6.8: Memory Latency Predictor information.

Circuits are cleared in a distributed and silent mode. When T and T' values reach zero on a resource arbiter (RA), the connection is eliminated as well. This is a valid point for addressing correctly miss peculation of circuits or when the data gets delayed more time than expected, for instance from the memory bank through the MC.

6.4.4 Memory Latency Control Unit

Figure 6.8 shows the MLCU module that assists the PC controller at the MC. It computes arrival time of memory blocks. MLCU works at bank level by monitoring their status (which row is open) and memory requests (which bank/row to access). The register keeps the information required per bank to correctly compute block arrivals: ID for the request identifier, ROW for the bank row where data is located, and T_{pred} and T_{end} define the time period when data arrives to MC. When a request arrives to the MC, the MLCU computes the bank associated. Then, a comparator checks if the bank is idle. In that case MLCU proceeds to calculate the block arrival time (LAT COMP logic block) and the request is sent to main memory. Otherwise, the address and requestor ID are queued and the request is sent to main memory. Queued bank requests are dequeued when a new block arrives from the bank.

To properly compute arrival times (T_{pred}), we need some timings from the memory, mainly the activation, precharge and read latencies. The last T_{pred} and T_{end} values are stored in the register (associated to the bank). If the current row is open the next T_{pred} is computed as the maximum between the last T_{end} and current time plus the read latency. Otherwise, T_{pred} is equal to the maximum between the last T_{end} and current time plus activation, precharge and read latency.

Whenever a new T_{pred} is computed, the MC schedules a SET_{CIRC} action. This action is scheduled sixteen cycles before the data arrives to MC. When the first flit from the incoming data arrives to the MC, then the MLCU dequeues a request and computes its T_{pred} value.

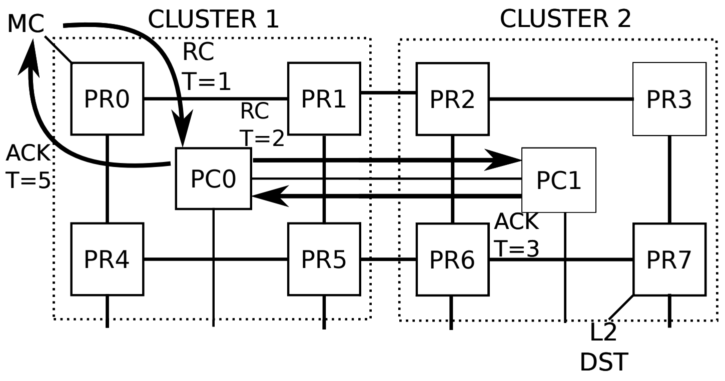


FIGURE 6.9: PROSA setup circuit example.

6.4.5 PROSA Circuit Setup Example

Figure 6.9 depicts an example of a successful PROSA setup circuit process between MC and L2, following a SET_{CIRC} action triggered by the coherence protocol. The circuit establishment process starts when the REQ message arrives to the MC. Let's assume that this event occurs at t_0 and all resources are available (no circuit conflicts will arise). Also, the MC knows the requested data will be available in 10 cycles (from memory). The MC will process the request in one cycle, thus, in t_1, will send an RC message to its PC (PC0).

At t_1, RC reaches PC_0 and attempts to get all the necessary resources in the cluster along the path between MC and L2. The resources RC will compete for are $R_{0E} \rightarrow R_{1E}$ (east port of PR_0 and PR_1). At each RA the request will be stored. Delta T will be set to 9 (Delta T at request) in the R_{0E}, R_{1E} RA modules. Delta T' will be set to the number of cycles needed for the transmission of the block.

At t_2, the RC message is forwarded from PC_0 to PC_1. Notice stored values of Delta T are decreased by one. At PC_1, the same process described for PC_0 applies, but now for resources $R_{2E} \rightarrow R_{3S} \rightarrow R_{7L}$. After winning the resources, PC_1 generates an ACK message that will be stored in the ACK queue. Delta T fields at RA modules for R_{2E}, R_{3S} and R_{7L} are set to 8.

At t_3, all the resources with one circuit established (even in *unconfirmed* state) decrease Delta T value by 1. Also, at t_3, PC_1 processes the ACK message. As said above, ACKs and NACKs messages have higher priority, and then this ACK will win the necessary RA modules, thus confirming the circuit at PC_1 and sending back the ACK message to PC_0. Finally, at t_4, the ACK message arrives to PC_0, being processed at t_5 and winning all the RA modules and confirming the circuit at PC_0. The ACK message is forwarded to the MC node, confirming the circuit at the Network Interface (NI).

In case that any of the resources were not available during the circuit setup time, the affected PC would generate a NACK message and would enqueue it into the NACK queue. The resources would be freed the next cycle and the NACK message would be transferred to the previous cluster. If a NACK is received at the NIC, the packet will be injected using packet switching.

Time advances and at time t_{11}, Delta T value at RA module associated with R_{0E} reaches 0, thus the signal *Circ_to_W* (in R_{0E}) is activated and set to point to the local port. This signal switches in input port from MC and output port E at PR_0. This circuit will last the required number of cycles for the message (Delta T'). All other programmed output ports (R_{1E}, R_{2E}, R_{3S} and R_{7L}) switch the input and output ports making the circuit just for Delta T' cycles. In this cycle the MC receives the block from memory and injects the first flit. The flit is forwarded and crosses the entire network reaching the $L2$ output port at $PR7$. Finally, when Delta T' reaches 0, the circuit is torn down in a distributed manner on each router just when the last flit of the message crossed the router.

6.5 PROSA Evaluation

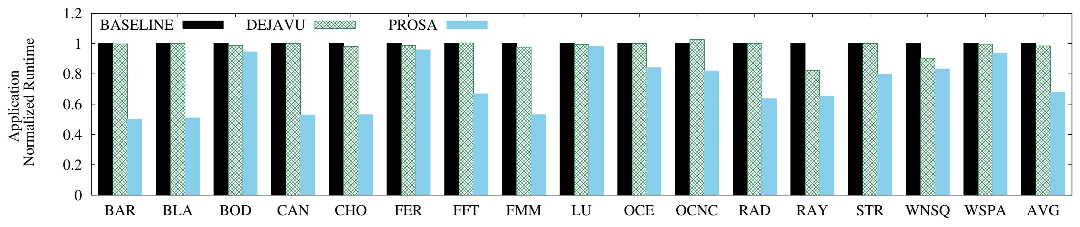
Now, we perform an evaluation of PROSA and analyze its behavior. First, we describe the analysis tools and simulation parameters. Then, we show the performance results of PROSA in a 8×8 mesh configuration. Finally, we provide implementation overheads of PROSA and an analysis about its area and power consumption requirements.

For the performance analysis we use an event-driven cycle-accurate simulator that models any network topology and router architecture. We model a 2-stage pipelined router (Input Buffer (IB), Routing (R), VC allocator and Switch allocator (VASA), that three in only one stage, and Crossbar (X)) with VCs and flit-level crossbar switching, as used in Garnet [67]. Table 6.1 shows the simulation parameters for the router and the cache hierarchy. L1 caches are private to the core and L2 cache is shared but distributed among all the tiles.

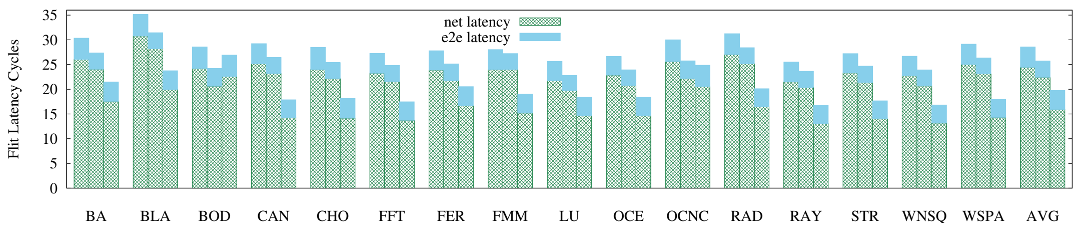
Parameter	Network	L1	L2 per tile	MC
Topology	8x8 mesh			
VCs/fly link	4/1 cycle			
Message sizes	8/72 bytes			
Flit/Queue size	8/72 bytes			
sets/way/line size (B)		32/4/64	128/16/64	
cache/tag latency		2/1	4/2	
Num. MCs				2
Topen/Tact/Tread				16/16/16

Table 6.1: Parameters and values used for routers and caches.

We evaluate three mechanisms: the baseline router, the Dejavu solution [14] and PROSA. In PROSA data sent through circuits take one cycle (using SMART). We analyze applications from SPLASH [68] and PARSEC [69]. Deja Vu pre-allocates the circuit between nodes in order to hide the setup latency by dividing the NoC in two planes: control and



(A) Application Runtime



(B) Flit Latency

FIGURE 6.10: Performance results for different architectures (BASELINE, DEJAVU, PROSA in column order).

data plane. The control plane is in charge of configuring the circuits. This plane has higher voltage and frequency, so is faster than the data plane. In this NoC, the request packet pre-allocates the path in backward direction as it approaches destination. The destination node can forward the response to the requestor node whenever the data is ready with no circuit setup. This approach can produce conflicts. Deja Vu configures the circuits in the order they are reserved.

6.5.1 Results

Figure 6.10a shows the application runtime. PROSA reduces, on average, BASELINE application runtime by 33.11%. PROSA reaches lower gains in applications with a low number of L2 misses (BODYTRACK), or with short application runtimes (LU) as it improves those application runtimes by 5.45% and 4.0% respectively. However, with balanced applications (OCEANNC), or applications with a high number of MC requests (FFT, CANNEAL, FMM, BLACKSCHOLES) PROSA reaches an improvement up to 50%. Deja Vu achieves negligible benefits in performance, as already seen in [14].

Figure 6.10b shows latency results. DEJAVU outperforms on average BASELINE by 10%. PROSA, in applications with a low number of MC requests and small datasets (e.g. BODYTRACK), achieves a 7% improvement on end-to-end latency. However, PROSA outperforms BASELINE up to 39% on applications with a high number of L2 misses (e.g. CANNEAL). On average, PROSA outperforms BASELINE end-to-end latency by 31.14%. For network latency (solid color), on average, PROSA improves by 34.16 %.

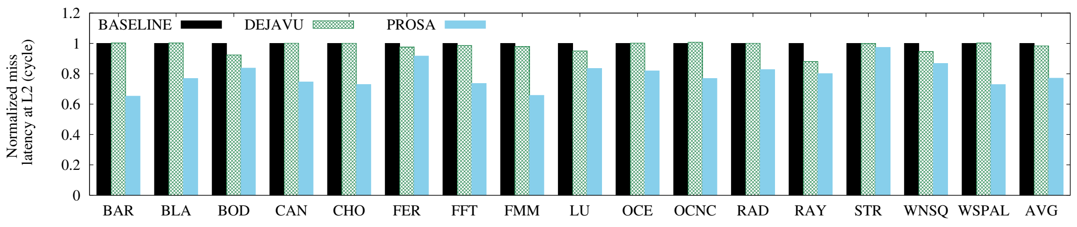


Figure 6.11: Memory latency results for different architectures. Normalized to baseline case.

Now, we analyze results for L1 miss latency normalized to BASELINE (Figure 6.11). In this case, as it occurs with runtime, DEJAVU slightly improves BASELINE and PROSA achieves better performance. On average, our proposal improves BASELINE by 23 %.

	REC.ACK	REC. NACK
MIN	(OCNC) 85.98%	(LU) 4.28%
AVG	90,62%	9,38%
MAX	(LU) 95.72%	(OCNC)14.02%

Table 6.2: Number of ACK and NACK messages generated in PROSA in PC module.

	CONF. INPUT		CONF. RA		
	NACK	ACK	FB	TMP	ARB
MIN	(LU) 0.01%	(BOD) 0.63%	0%	(WSPA) 1.92%	(WSPA) 0.31%
AVG	0,04%	3,94%	0,00%	4,61%	0,79%
MAX	(WSPA) 0.18%	(WSPA) 9.27%	0%	(OCNC) 9.04%	(OCNC) 1.70%

Table 6.3: Number of conflicts generated in PC module.

Table 6.2 shows statistics about PROSA circuits. The first column shows the received ACKs (the number of circuits established successfully). In all cases this is higher than 85.58% and the average is 90.62%. The second column shows the received NACKs and is the sum of the five columns of Table 6.3, which list the different types of conflicts in the PC controller. The first and second column show the number of NACKs generated at the input of the PC controller as a normal request and an ACK/NACK conflicted at the same time. The last columns show conflicts generated at the RA arbiters due to (1) the register table is full (FB), (2) the required period of time of one request overlaps with one established circuit (TMP), and (3) two requests collided in the same RA module (ARB) due to the static arbiter. As we can see, most conflicts are generated because of temporal conflicts in generating circuits or because of concurrent requests enter with conflict with an ACK is the same resource. However, the current table size at RA modules (4 entries) seems to be properly sized (even could be reduced thus saving more area) as the number of conflicts due to the table being full are negligible.

6.5.2 PROSA Implementation

We have implemented all the PROSA infrastructure for a 4×4 CMP system. Each PC module has been implemented in Verilog and tested. We use a canonical router design with 64-bit flits, four 9-flit depth VCs, and with seven ports to attach 2D-mesh ports and L1, L2, and MC (including the MLCU). We use Design Vision tool from Synopsys with 45nm Nangate open cell library [70]. Power results are obtained from Orion-3 power library [71].

Table 6.4 shows the area overheads of PROSA for different configurations. In all of them, and for the sake of comparison, we also account for the components to build a cluster of four routers. In the case of PROSA we consider all the components (including the PC and the four PRs).

Configuration	area (μm^2)	overhead
Baseline router	243784	-
PROSA Router (PR)	248200	1.8%
PROSA Controller (PC)	76547	-
Baseline cluster	975136	-
PROSA cluster	1069347	9.66%
Flattened Butterfly cluster	1380109	41%
2xBaseline cluster	1658560	70%

Table 6.4: Area overheads for different router and NoC organizations.

As we can see, the PROSA router takes only 1.8% more area than the baseline router. The PROSA controller (PC) takes less area than a baseline router (22% of baseline router area). However, this component is new and needs to be considered as an additional overhead. To make this comparison fair, the table shows area of cluster regions. In this case, the PROSA cluster takes 9.66% more area. The NIC's overhead can be considered as negligible.

PROSA overheads can be seen as high. However, we should consider the performance gains that PROSA circuits enable. In order, however, to better assess the overheads, the table shows two additional configurations worth being analyzed. The first one, $flattenedbutterfly$, is the overhead for a flattened butterfly topology which has more connectivity along each dimension and direction. In this case, because of the larger number of input ports, the area overhead is increased by 41%. The second one is for the case where the baseline cluster is enhanced with double flit size. This can lead to faster transfer times between the nodes. However, as we can see, overhead skyrockets to 70% additional area. This is mainly due to the larger buffer requirements.

Figure 6.12 shows the energy results. Although the leakage energy increases by 42% the total energy consumption is reduced by 3%. This reduction is due to switching and internal energy, which are reduced by 10%. As described in [14] Deja Vu achieves a 30%

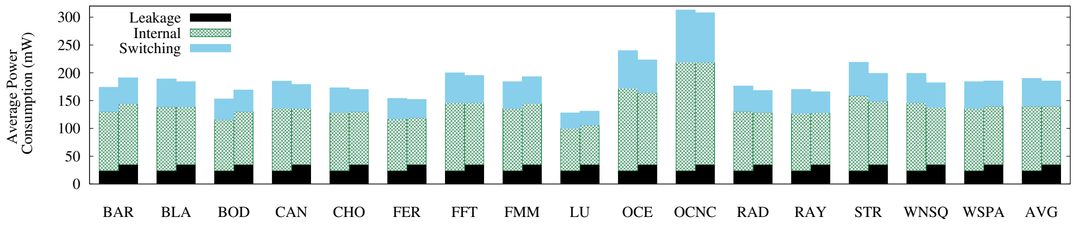


Figure 6.12: Power consumption results. First column represents BASELINE and second PROSA.

energy reduction, which is similar to our results (but without the latency improvements). Notice that the 30% execution time reduction will translate in major power savings.

6.6 Related Work

Circuit-switching [64] has been used in a number of previous works in NoC architectures in order to reduce on-chip communication latency. Once a circuit is set, data does not travel through the routing and arbitration stages on each router. However, setup time usually causes low resources utilization and performance degradation. On the other hand, packet switching improves resource utilization and network performance, splitting the entire message in smaller blocks and forwarding them along the network.

Some works try to get benefit from both mechanisms by implementing a hybrid circuit-packet switching strategy. Kumar [9] proposes Express Virtual Channels (EVC) allowing packets to bypass intermediate routers along their path. EVCs only allow to connect nodes along the same dimension, so circuits cannot turn from one dimension to another. PROSA, on the other hand, allows to connect nodes regardless their location, thus offering more flexibility.

Jerger [10] proposes circuit switched coherence, setting permanent circuits between pairs of frequent data sharers instead of tearing them down. It allows to quickly send data between the same nodes. However, if another circuit requires the resource, the data is switched to packet switching until it reaches destination. Yin [11] proposes a hybrid circuit-packet switched network in which the packet is forwarded along the packet network while the circuits can be set in parallel, using TDM. Yim's proposal also expends time in the setup latency. Mazloumi [12] proposes another hybrid packet-circuit switched router. This mechanism setups the circuit along the network while the request message is being forwarded between the requestor and the destination. When the request reaches the destination and the data is ready, the mechanism sends a probe message activating the reserved circuit, after that the data is sent. All these mechanisms require a setup period. However, PROSA hides the setup circuit latency based on the coherence protocol and when the circuit usage finishes, resources are freed allowing the packet switching to forward packets without having to tear down any circuit, the data is sent without delay.

Abousamra [14] proposes Deja Vu (briefly described in Section 6.5). In Deja Vu, the selected order schema can produce underutilization of network resources. In [15] authors alleviate the problem by using a different order. However, it still requires the high frequency and voltage control plane. PROSA, as Deja Vu, preallocates the circuit in order to hide the setup circuit delay, However, as per our evaluation, PROSA achieves better performance results. Van Lear [16] proposes a coherence-based message predictor for optical interconnection networks. In the proposal a global predictor establishes the circuits between nodes. All the traffic in the network has to cross the predictor, thus potentially causing a bottleneck in the network. This proposal is also for optical interconnects where a full optical crossbar is assumed. This makes scalability a major issue. Krishna in [7] presents SMART, a multihop network with single-cycle data-path all the way from source to destination. Setup circuit is required one cycle before the data is sent and partial circuits can be established. An extra network is required to send SMART-hop Setup Request, SSR. This mechanism is not based on coherence cache as PROSA and our mechanism relies on SMART circuits. Peh [18] presents flit-reservation flow control. In this proposal the circuit is setup hop by hop. However PROSA uses clusters to setup circuits, improving the time required to establish the circuit. PROSA does not require to buffer messages, contrary to flit-reservation. Our proposal anticipates the circuit setup process.

As a summary, PROSA allows to establish circuits between any pair of nodes hiding the setup latency. PROSA uses the coherence protocol information to establish these circuits. In addition, PROSA programs circuits for their exact period of time they will be needed, thus not conflicting with other circuits using the same resources (in other time periods). Indeed, PROSA circuits can be steered by coherence protocols or even for other higher-level applications where traffic bursts can be predicted and requested ahead of time. The previous strategies do not rely on coherence protocols or they rely on more expensive architectures and technologies.

6.7 Conclusions

In this paper we introduce PROSA, a circuit-switched enabled NoC architecture which allows the coherence protocol to steer circuit connections for future predictable and speculative connections between memory controllers (MC), L2 cache banks and L1 caches. Connection establishment is performed as single-packet based and established for the required time period for L2 to L1 an MC to L2. On a miss predict the circuit is silently removed. PROSA builds a separate infrastructure to manage circuits, thus the main NoC network is not affected by the additional control traffic.

Results show network latency is reduced up to 35% by correctly predicting and using circuits with PROSA. Also, PROSA outperforms the baseline designs by reducing application runtime by 33% and PROSA improves the DEJAVU performance results achieving similar energy results. PROSA takes an overhead of 9.66% more than the baseline proposal, however, it saves more than 3% in power consumption. Overhead of PROSA is reasonable given the benefits in performance. In a future work, we plan to extend our proposal with partial circuits, increasing the number of MCs and studying the table size scalability.

Chapter 7

PROSA: Protocol-Driven Network on Chip Architecture

- **Authors:** Miguel Gorgues (Universitat Politècnica de València), Mario R. Casu (Politecnico di Torino) and José Flich (Universitat Politècnica de València)
- **Type:** Journal
- **Conference:** IEEE Transactions on Parallel and Distributed Systems (Volume: PP, Issue: 99)
- **Year:** 2017
- **DOI:** 10.1109/TPDS.2017.2784422
- **URL:** http://ieeexplore.ieee.org/document/8219740/
- **Citation:** [79]

7.1 Abstract

Nowadays chip multiprocessors (CMPs) tend to increase the number of cores, usually implementing a distributed shared last level cache (LLC). The network on chip (NoC) is in charge of interconnecting the cores, memory controller(s) and cache banks, largely impacting memory access latency. Packet switching (PS) is typically used in NoCs but circuit switching (CS) may complement PS achieving higher performance if the circuit is established before its need. In this paper we propose PROSA, an architecture to improve memory access latency by using CS. In PROSA, the coherence protocol steers the circuit setup logic in order to configure circuits before they are needed and only for the time they are required. PROSA uses a clustered router approach where groups of four routers are clustered and their circuit control logic is combined. Based on key design decisions, we present different PROSA versions, analyzing their impact on applications and NoC performance. PROSA is able to reduce applications' execution time by 35% while it significantly reduces average network flit latency by 54%, leading to a reduction of miss load (and store) latency of 21% (in CMP systems with 64 processors). PROSA needs 8.4% more area, but reduces power consumption by 7%.

7.2 Introduction

Chip multiprocessors (CMPs) rely on networks-on-chip (NoCs) [78] to achieve fast communication between processor cores, memory caches and memory controllers (MC). It is by no doubt that NoCs play a vital role on CMPs performance, mainly as they lay along the critical path of communication, and thus, affect process communication latencies and most important, memory access latency.

During more than one decade, NoCs have been researched (not only for CMPs) with two main goals in mind, network throughput and network latency. While throughput is an important aspect, in CMPs the average load of links found for accepted benchmarks (PARSEC [69], SPLASH [68], ...) is typically low. This means latency becomes the significant metric when dealing with NoCs for CMP systems. Indeed, a memory transaction is started when one processor requests a load or a write on its private L1 data cache, and finishes when the data is delivered. The NoC is used to send request commands and data between cache levels and the MC. Therefore, the NoC plays a vital role as the memory request gets blocked until data is delivered.

In this paper we address the issue of latency reduction of memory transactions. We propose PROSA, a PROtocol-oriented circuit Switch Architecture, which co-designs both the NoC and the coherence protocol and put them to play together. Our approach enhances the NoC with a new clustered component, the PROSA controller (PC), which is in charge of managing circuits and to resolve any possible conflict. The PC is in charge of four NoC PROSA routers (PR) and steers their local circuits when needed. For the transmission of data through circuits, PROSA relies on the SMART [7] asynchronous repeater technique where a multihop network with single-cycle data-path can be achived.

Circuit Switching (CS) has been explored and used in the past in many different network-related fields. PROSA differs from previous circuit-switching methods by means of the feedback provided by the coherence protocol. Although some previous works also combined protocols and NoC designs [7, 10–12, 16], our proposal lets the coherence protocol to steer the network in programming circuits before they are needed, hiding the set up circuit delay. Moreover, PROSA does not need extra buffer resources at the router. Just a pair of multiplexer/demultiplexer is needed on each bidirectional router port. Section 7.7 describes related work.

The effectiveness of the CS approach resides on the type of traffic that will be using circuits. Long messages or messages sent to distant destinations are good candidates to configure circuits for them, as they will pay off the time spent in setting up the circuit. Moreover, bursty traffic is the best candidate for CS. However, in a CMP scenario, bursty traffic is not common. Instead, short single-flit messages (carrying protocol commands) and long multi-flit messages (carrying memory blocks) define the network traffic.

As expected, there exist multiple key design decisions when building a CS-based system for CMPs. Different design alternatives are explored in PROSA. One alternative is to setup circuits only for long multi-flit protocol messages injected by the coherence protocol. Circuits for those messages may be setup before they are indeed injected, taking advantage of the cache latency when accessing the L1 and L2 caches and for the incurred processing delay in the network interface (NIC). Thus, hiding the circuit setup time. Another alternative is to setup circuits for all messages, either short or long taking as a premise that even with an small delay at injection (because we need to setup the circuit) the transmission latency of the message will still be smaller than the transmission time in packet switching (PS) mode (as we use SMART technique). Another alternative relies on message distance to destinations, using circuits only for messages with closer destinations, thus circuit setup time is shorter. Finally, few additional cycles (slack) can be provided to the circuit setup process, increasing the success rate of circuits established and used by the application. All these alternatives will be explored in PROSA.

Evaluation results show different benefits when using PROSA. When PROSA is used for long predicted messages, the base line design is improved in average network latency by about 34%. Indeed, PROSA reduces the load and store miss latency by 21%. This is achieved because PROSA correctly sets up 90% of the circuits ahead of time. For implementation overheads, the PROSA cluster takes 9.66% more area but saves more than 3% in power consumption.

Although different PROSA versions improve network latency by up to 35%, all the different PROSA versions proposed in this paper obtain similar runtime performance

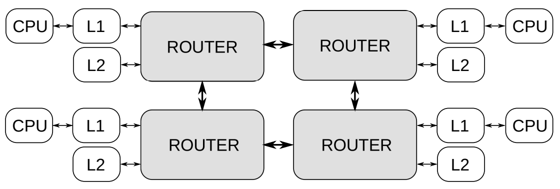


Figure 7.1: Chip Multi Processor Architecture.

for the different tested applications. This is mainly due to the delay introduced in the coherence protocol and the setup circuit. However, the best PROSA configuration achieves and additional 7% of power saving and reduces area by 2% when compared to the original PROSA version.

The paper is organized as follows. In Section 7.3, we describe and analyze the coherence protocol. This will center our proposal and provide justification of its need. In Section 7.4 we describe PROSA and the different alternatives we will explore. In Section 7.5, we provide the evaluation and analysis. Results are discussed in Section 7.6. Related work is described in Section 7.7 and the paper concluded in Section 7.8.

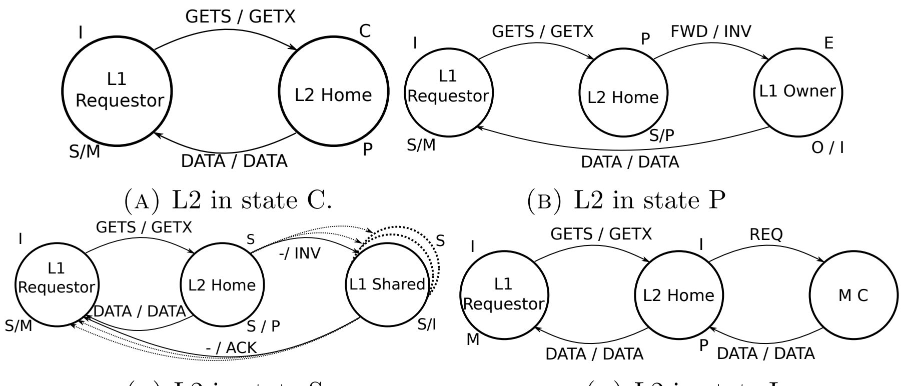
7.3 Coherence Protocol Analysis

Figure 7.1 shows the CMP architecture assumed for this work. The CMP uses private L1 caches for each core and shared but distributed L2 banks along a tile-based organization. Two memory controllers (MC) are placed on each top corner. A 4 × 4 mesh NoC uses a 2-stage pipelined 7-radix router design (four ports to connect to neighbor routers and three to connect L1, L2, and MC).

The system implements the MOESI protocol [53] at L1. L2 blocks can be in P (private), S (shared), C (cached), or I (Invalid) state. In C mode no L1 cache has a copy of the block. Inclusive caches and a write back policy is used. A static mapping policy of blocks to L2 banks is used.

Figure 7.2 shows the transaction diagrams related to load and store operations. In the figure nodes are represented by boxes. The current block state is represented with text placed just above the boxes while the new state after the transaction is represented under the boxes. Messages sent between nodes are represented by arrows. Messages for load and store operations are combined (e.g. GETS/GETX).

Whenever an L1 load miss occurs, a GETS message is sent to the L2 Home bank. Based on the block state at L2 different actions are performed. If the block is in S or C state (Figures 7.2a and 7.2c), the L2 sends the data to the L1 requestor. If the block is in P state (Figure 7.2b), the L2 sends a forward (FWD) message to the L1 with the block. The block state at L2 changes to S. When the FWD message arrives, the L1 cache sends



(A) L2 in state C. (B) L2 in state P

(C) L2 in state S. (D) L2 in state I

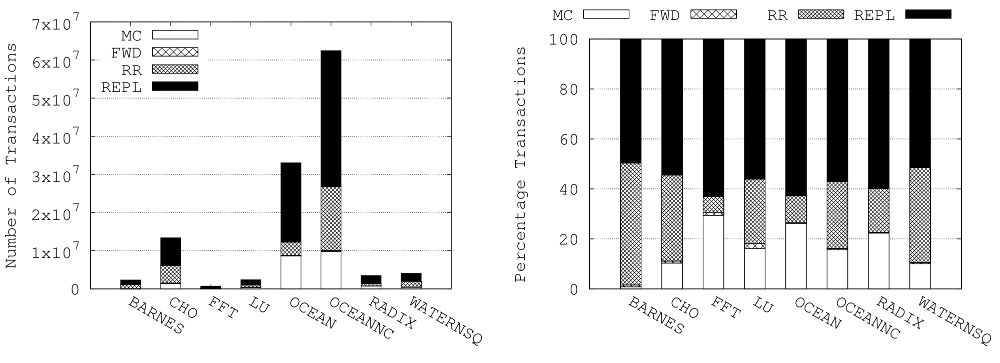
FIGURE 7.2: Coherence protocol transactions.

the data to the L1 requestor cache and the block state changes to O. Finally, if the block at L2 is in I state (meaning a miss occurs, Figure 7.2d), a REQ message is forwarded to the MC. The L2 receives the data and forwards it to the L1 cache requestor. The state of the block is set to P in L2 and to E in L1.

Whenever a L1 store miss occurs, a GETX message is sent to the L2 Home bank. Similarly, if the block is in C state (Figure 7.2a), the L2 sends the data to the requestor and changes the block state to P. A different case occurs when the block is in P state (Figure 7.2b). The L2 sends an invalidation message (INV) to the L1 owner cache. When the INV message arrives, the data is sent to the L1 requestor and the block changes from E to I. When the block is in S state (Figure 7.2c), the L2 sends the data to the L1 requestor and sends INV messages to all the L1 sharers. When each L1 sharer receives the INV message, it sends an ACK message to the requestor and changes the block state to I. Finally, if the block is in I state (miss) (Fig. 7.2d), then a REQ message is forwarded to the MC. Once data is received, it is forwarded to L1. The block is put in P in L2 and in M in L1.

As we see, the protocol faces mainly four possible paths depending on the type of operation (load/store), L1 access type (miss/hit) and L2 access type (miss/hit). Whenever a miss occurs at L1 and L2 the NoC gets involved several times and the memory transaction latency is significantly increased. Factors that may affect significantly memory latency are the distance between the L1 requestor and the L2 Home bank, distance between the L2 Home bank and the MC, and the network congestion.

To better analyze these effects, Figure 7.3a shows memory transactions classified by type: MC transactions (due to miss both in L1 and L2 banks), ReqResp transactions (request-response due to an L1 miss and an L2 hit to a clean block), FWD transactions (forward transactions due to an L1 miss and an L2 forward to the owner) and REPL transactions (replacement transactions due to L2 being full). Different SPLASH-2 applications are run in a system described in Section 7.5.



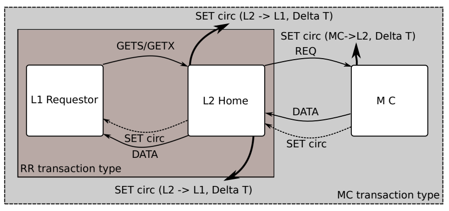
(A) total transac. (B) total transac (%)

FIGURE 7.3: Memory transactions types for different SPLASH-2 applications.

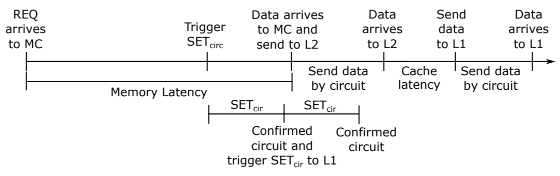
First thing to note is the different amounts of transactions between applications. This depends on the complexities of the applications. However, when we normalize the numbers, Figure 7.3b, we see similarities. A high percentage of REPL transactions exist in all applications (more than 50% of the transactions). This is mainly due to the L2 cache size and the dataset of the applications. These transactions can not be predicted from the coherence protocol point of view. Noteworthy, we can observe the small percentage of FWD transactions. All applications exhibit less than 3% of this type, and for OCEANNC and RADIX lower than 1%. This indicates shared blocks mainly follow a multiple sharers pattern and not a producer-consumer pattern. Now, we focus on a more interesting transaction type. Some applications (FFT and RADIX) trigger a large percentage of MC transactions while others (BARNES and WATERNSQ) have a marginal percentage. This difference is due to the L2 hit rate and the memory footprint of the application. Finally, ReqResp transactions are highly representative in some applications (BARNES and WATERNSQ) while minimal in others (FFT). This is due again to the hit rate in L2 and the memory footprint. Thus, applications either have large percentage of MC transactions or large percentage of ReqResp transactions. In both cases, MC and ReqResp, traffic needs between L2 and MC, and between L2 and L1 (requestor) can be predicted. PROSA will exploit this fact by using circuits for MC and ReqResp transactions.

7.4 PROSA

PROSA will be developed in phases. At each phase more functionality will be provided, in order to achieve more efficiency and performance. Also, we will extend PROSA with complementary designs such as the memory latency estimator device or the inclusion of slack time to circuit setup logic. For the sake of understanding, we will start with the base design that focuses only on predictable coherence actions without slack time



(A) Transaction.



(B) Cronogram

FIGURE 7.4: PROSA new actions triggered by the coherence protocol and circuit establishment time line for MC transaction.

for setup circuits. We refer to this method as standard PROSA. Then, we will extend PROSA.

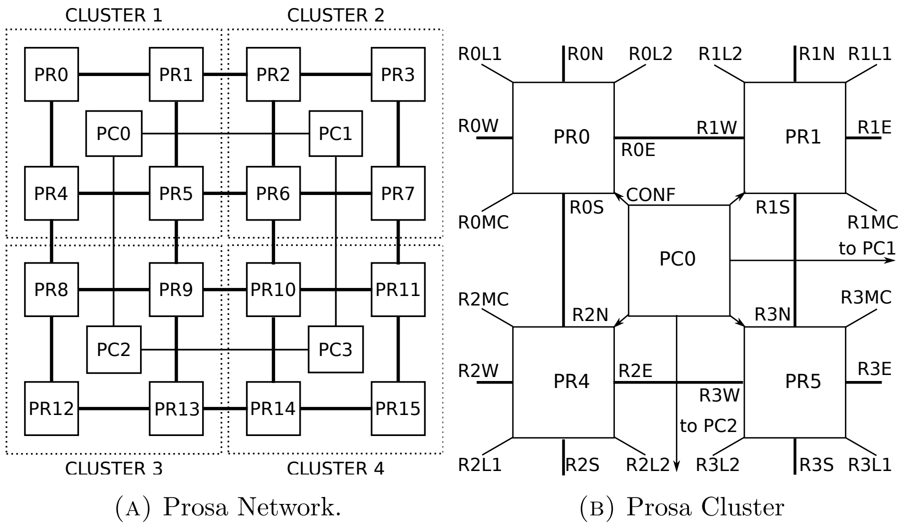
7.4.1 Standard PROSA

PROSA sets up circuits between MC-L2 and L2-L1 before they are needed. First, we show modifications performed in the coherence protocol and then describe modifications needed in the NoC (the PROSA controller and PROSA router) and MC (the Memory Latency Control Unit, MLCU).

7.4.1.1 PROSA Coherence Protocol

In order to program circuits, we add a new protocol action termed SET_{CIRC}. This action involves the source and destination of the circuit, and the time period the circuit will be needed (Delta T). The action is triggered by the coherence protocol in MC and ReqResp transactions (see Figure 7.4a). In MC transactions, whenever a request is received by the MC, a Memory Latency Control Unit (MLCU; described in Section 7.4.1.4) predicts when the data will arrive from main memory. Based on this delay, the MC triggers a SET_{CIRC} action that will setup a circuit after a delay period. The delay period equals to the predicted memory latency minus the circuit setup period,(CSP), required time to setup a circuit between the MC and the destination node. The SET_{CIRC} action sets the circuit. By the time the block arrives to the MC the circuit has been set and is kept during the time the data will be transmitted. When SET_{CIRC} arrives to L2, the circuit is confirmed (acknowledged) but in parallel a new SET_{CIRC} command between L2 and L1 is triggered. Thus, when the data arrives to L2, after accessing the L2, the block is sent also to L1 using a circuit. These two circuits are predictable one as they will be necessary regardless of network status. Figure 7.4b shows timings of both circuits.

In ReqResp transactions (between L1 cache and L2), whenever a GETS/GETX message is received, a SET_{CIRC} action is triggered between the L2 and the L1. Again, the connection will be set only for the time period the data is available. However, contrary to the two previous triggered circuit scenarios (in MC transactions) now this circuit is



(A) Prosa Network. (B) Prosa Cluster

FIGURE 7.5: PROSA controller, detailed components.

speculative, in the sense the circuit may not be needed if the block is in P or I state. Therefore, for this type of circuit we will need to automatically and efficiently tear down the circuit when transmission is finished or when it is clear the circuit is not needed. Notice that this modification does not introduce out of order delivery or duplicates, because for the same transaction the protocol never sends two messages to the same destination. Either the circuit is used for transmission of the message or the message uses the regular PS network for transmission.

7.4.1.2 PROSA Circuit Network

PROSA follows a clustered approach where one PROSA Controller (PC) controls four routers (Figure 7.5a). When the SET_{CIRC} action is triggered in an L2 or MC, a request circuit (ReqCir) message is forwarded to the local PC. If the ReqCir wins the needed resources in the cluster, it advances to the next PC along the path. If not, a NACK response is sent to the source of the ReqCir message. When ReqCir reaches the PC that controls the destination and wins the resources, an ACK response is sent back to the source via the PC network. Routers are not affected by ReqCir messages, neither ACKs nor NACKs. All them travel via PCs.

On top of Figure 7.6, we show the ReqCir message is composed by eight fields. *P* refers to the input port from which the ReqCir arrives to the cluster. *src* and *dst* are the source and destination for the circuit. *Delta T* and *Delta T'* carry the number of cycles to wait until the circuit will be established and torn down, respectively. *Type* carries the type of the ReqCir message (REQUEST, ACK, NACK). *ID* identifies the circuit inside the network, and finally, *Golden Token*, refers to the distance between *src* and *dst*. This field is used in the PC to assign priorities between requests.

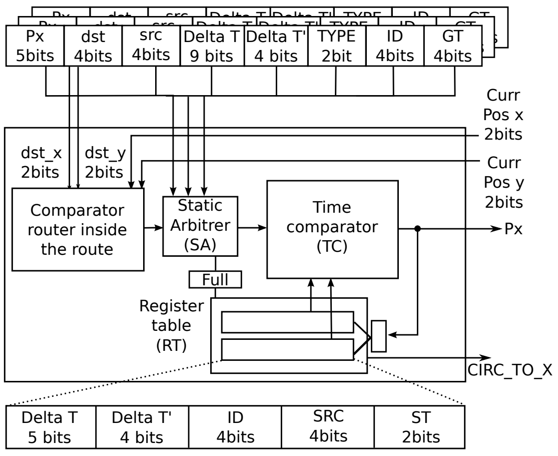


FIGURE 7.6: Resource.

Below the ReqCir structure, in Figure 7.6, we show the Resource Arbiter (ResArb) that controls whether a specific resource can be reserved for a particular period of time. A PC consists of several ResArb modules, one ResArb per output port in the cluster controlled by the PC. In total, each cluster has twenty eight ResArb, all shown in Figure 7.7 (e.g. R_{0S} corresponds to the south output port at the north-west router in the cluster). Each ResArb module arbiters between some requests, the number of requests depends on the number of inputs ports that have dependencies with the resource (the output port). This depends on the routing algorithm, in our case XY routing. In order to support other routing algorithms or topologies, the path comparator logic would need to be adapted together with the wiring connections between resource modules.

When an ReqCir message arrives to ResArb, the *dst* field is checked to decide whether the output port controlled by the ResArb is along the path between *src* and *dst.* If so, the ReqCir goes to the Static Arbiter which arbiters between the incoming requests at this point. Static Arbiter gives priority to requests with larger values in the *Golden Token* field (longer paths have priority). Then, ReqCir advances to the Time Comparator (TC) module where it checks that the request does not overlap in time with any previously programmed circuit. Finally, if there is no conflict with programmed circuits then the ReqCir is stored in the Register Table and forwarded along the PC network. Notice that only one ReqCir message gets access to the table. As we will see, this does not impact performance and reduces complexity.

The register table keeps circuits information. For each circuit the following fields are stored: *Delta T*, *Delta T'*, *ID*, *src*, and *ST*. When an ordinary request wins the resource, *Delta T*, *Delta T'*, *ID*, *src* get the values from the ReqCir message. *ST* keeps the state of the circuit, which can be *unconfirmed*, *confirmed*, or *empty*. *Delta T* is decremented by one every cycle. When it reaches zero, the ResArb module activates the outputs *Cir_to_P* which control the circuit establishment in the PROSA router (described later).

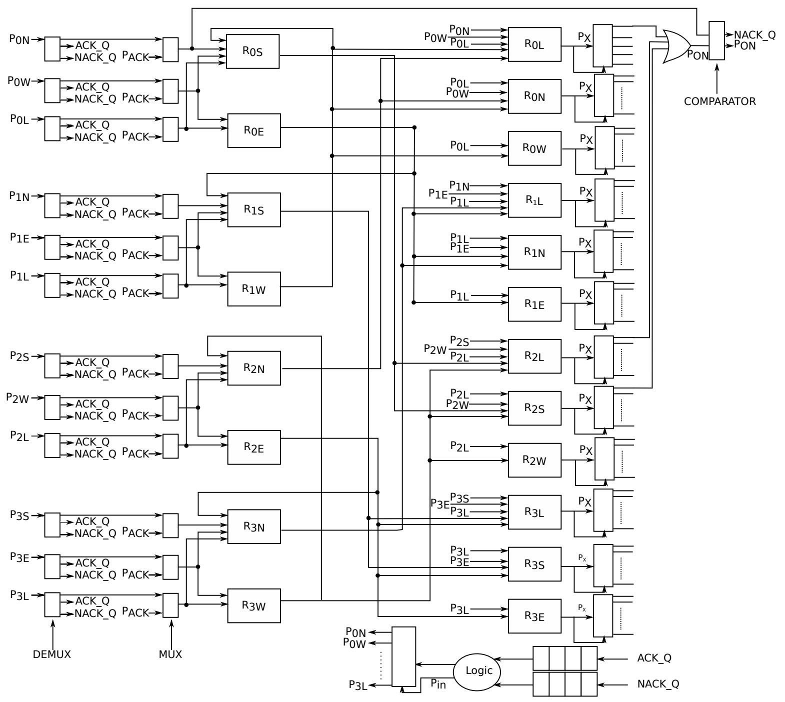


Figure 7.7: PROSA Controller.

Delta T' is also decremented by one every cycle. When *Delta T'* arrives to zero the signals and the register are cleared.

When an ACK or a NACK arrives to ResArb, it follows the same path than an ordinary request, with some small differences. First, the ReqCir message advances to Static Arbiter. ACK/NACK messages have higher priority than ordinary requests. By construction we guarantee only one ACK/NACK enters one PC each cycle, thus they will always win the Static Arbiter access. Then, in Time Comparator the ReqCir message checks the information stored in the table. An ACK consolidates the stored information while a NACK removes it. Then, the ReqCir is forwarded (to next ResArb along the path inside the cluster or to the next PC).

The PC is shown in Fig. 7.7. It contains 28 RAs, 7 per router. For the sake of simplification, we group all outputs of a router (L1, L2, MC) into a single ResArb (R_{xL}), showing only twenty RAs. The PC implements two queues to store generated and incoming ACK/NACKs. ReqCir requests can be dropped. A demultiplexer located at the input port (left-hand side) divides requests by ACK/NACK messages (*type* field of the ReqCir message is used as selector). When a request arrives, it continues through the PC. However, if the incoming request is an ACK/NACK, it is sent to the corresponding ACK or NACK queue. The next stage in PC is a multiplexer, which multiplexes between incoming ReqCir message and queued ACK/NACKs, giving priority to ACK/NACKs. In case of conflict the request is discarded (generating a NACK which will be queued). Finally, the selected message enters the ResArb tree.

ResArb modules are linked following the resources dependencies imposed by the routing algorithm (XY in our case). An example of linked arbiters are $R_{0E} \rightarrow R_{1S} \rightarrow R_{3S}$ for messages coming through router PR0 and leading to a destination below router PR3. An ResArb module has several input and output dependencies as multiple routing combinations exist. Along this path, if a request wins all the required ResArb modules (from left to right) the request will be forwarded to the next PC, or will generate an ACK if the destination is controlled by the current PC. A comparator at the output of the PC module (right-hand side) checks whether one ReqCir message won all the required resources (it was at the input side and also succeeds at the output side). If not, a NACK message is triggered and stored in the NACK queue. When a ReqCir is sent to the next PC, the value of Delta T is decremented by 1. When this value reaches zero, the ReqCir is discarded by the PC.

Figure 7.7 shows the ACK and NACK queues and its control logic block. The logic guarantees only one ACK/NACK message will access the PC each cycle, ensuring that they will always win all RAs. NACK messages have higher priority than ACK messages. Moreover, the logic changes the input port of the ACK response, computing the input port of the ordinary request corresponding to this ACK response. Thus, ACK messages cross the same ResArb path used by the associated request messages. Notice that NACK generated messages locally in a PC need to be re-injected in the ResArb tree again to remove all the reserved resources, and then sent back to the previous PC.

7.4.1.3 PROSA Router

Figure 7.8 shows the modifications performed in the baseline router. We only add one demultiplexer per input port, one multiplexer per output port, connections between those elements and an asynchronous repeater per output port. The repeater allows to forward the flit very fast reducing wire delay, as is used in SMART [7]. This technology allows one flit to cross all the network in one cycle.

The router works as usual until signals *Circ_to_X* are activated, where X is the output port ($L1$, $L2$, MC, N, E, W, S). In this situation, the corresponding input and output ports are switched in and the arrived flits are blindly forwarded through them. At the same time, VA and Static Arbiter arbiters associated to the output port are disabled. Notice that the circuits will not use buffer resources, therefore circuits cannot introduce deadlocks.

Circ_to_X signals are generated by the PC , each generated by a single ResArb module (the one that manages the output port). *Circ_to_X* indicates the input port that has to be switched in to the 'X' output port. One wire is used for each possible input port. Thus, the demultiplexer at an input port is selected from the ORing of all inputs bits on *Circ_to_X* signals associated to the input port (e.g Circ_to_1[1] or ... or Circ_to_n[1]),

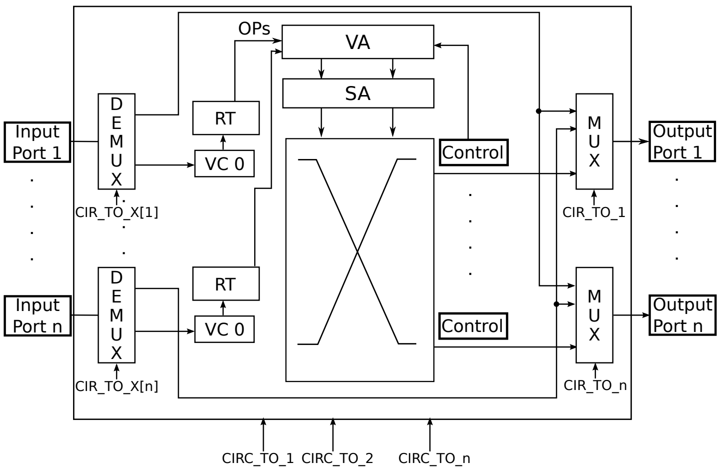


FIGURE 7.8: PROSA router.

and the multiplexer at an output port is selected from the ORing of all the wires from the associated *Circ_to_X* signal. During the setup process, PROSA sets at each router the exact cycle where the ports need to be switched in. Programming those values is the most critical part to guarantee correct operation. Robustness is guaranteed by the correct implementation of the circuit setup process.

Circuits are cleared in a distributed and silent mode. When T and T' values reach zero on a resource arbiter (ResArb), the connection is eliminated as well. This is a valid point for addressing correctly miss speculation of circuits or when the data gets delayed more time than expected, for instance from the memory bank through the MC.

7.4.1.4 Memory Latency Control Unit

Figure 7.9 shows the MLCU module that assists the PC at the MC. It computes arrival time of memory blocks. MLCU works at bank level by monitoring its status (which row is open) and memory requests (which bank/row to access). The REG register keeps the information required per bank to correctly compute block arrivals: *ID* for the request identifier, *ROW* for the bank row where data is located, and T_{pred} and T_{end} define the time period when data arrives to MC. Each bank has also a buffer to store pending requests.

When a request arrives, the MLCU unit computes the bank associated. A comparator checks if the bank is idle (memory controller is not waiting for data from this bank). If idle, MLCU computes the block arrival time (LAT COMP logic block) and the request is sent to DRAM. The request ID is stored in REG register associated to the bank. Otherwise, the address, requestor ID, and required information to compute the memory latency are queued in the buffer associated to the bank. The request is sent to DRAM.

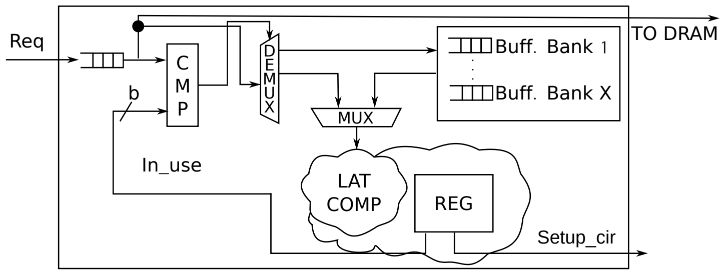


FIGURE 7.9: Memory Latency Predictor information.

A circuit setup process is triggered whenever the expected arrival time for a requested stored in REG is 16 cycles. When data arrives from DRAM the data is injected through the circuit (if set) or using packet switching. At the same time, if the buffer associated to the bank has a stored request, the MLCU dequeues the request and computes the new data arrival time from DRAM and stores the request in the REG register associated to the bank.

To properly compute arrival times (T_{pred}), we need some timings from the memory, mainly the activation, precharge and read latencies. The last T_{pred} and T_{end} values are stored in the register (associated to the bank). If the current row is open the next T_{pred} is set as the maximum between the last T_{end} and current time plus the read latency. Otherwise, T_{pred} is set to the maximum between the last T_{end} and current time plus activation, precharge and read latency.

Whenever a new T_{pred} is computed, the MC schedules a SET_{CIRC} action. This action is scheduled sixteen cycles before the data arrives to MC, this is the maximun number of cycles that the PC needs to setup a new circuit with the farthest node. When the first flit from the incoming data arrives to the MC, then the MLCU dequeues a request and computes its T_{pred} value.

7.4.1.5 PROSA Circuit Setup Example

Figure 7.10 depicts an example of a successful PROSA setup circuit process between MC and L2, following a SET_{CIRC} action triggered by the coherence protocol. The circuit establishment process starts when the REQ message arrives to the MC. Let's assume that this event occurs at t_0 and all resources are available (no circuit conflicts will arise). Also, the MC knows the requested data will be available in 10 cycles (from memory). MC will process the request in one cycle, then, in t_1, will send an ReqCir message to PC0.

At t_1, ReqCir reaches PC_0 and attempts to get all the necessary resources in the cluster along the MC-L2 path. The resources ReqCir will compete for are $R_{0E} \rightarrow R_{1E}$ (east port of PR_0 and PR_1). At each ResArb the request will be stored. Delta T will be set

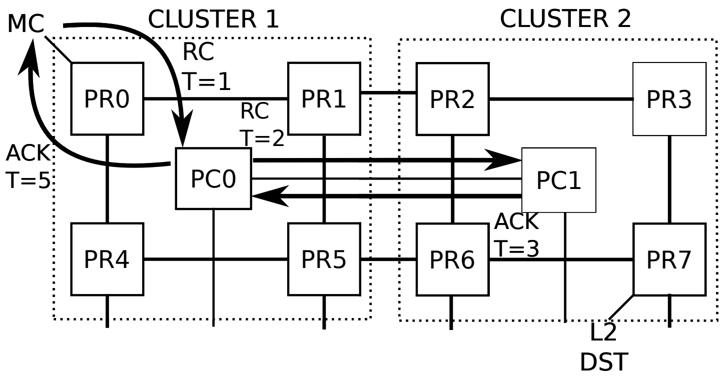


FIGURE 7.10: PROSA setup circuit example.

to 9 (Delta T at request) in the R_{0E}, R_{1E} ResArb modules. Delta T' will be set to the number of cycles needed for the transmission of the block.

At t_2, the ReqCir message is forwarded from PC_0 to PC_1. Notice stored values of Delta T are decreased by one. At PC_1, the same process described for PC_0 applies, but now for resources $R_{2E} \rightarrow R_{3S} \rightarrow R_{7L}$. After winning the resources, PC_1 generates an ACK message that will be stored in the ACK queue. Delta T fields at ResArb modules for R_{2E}, R_{3S} and R_{7L} are set to 8.

At t_3, all the resources with one circuit established (even in *unconfirmed* state) decrease Delta T value by 1. Also, at t_3, PC_1 processes the ACK message. As said above, ACKs and NACKs messages have higher priority, and then this ACK will win the necessary ResArb modules, thus confirming the circuit at PC_1 and sending back the ACK message to PC_0. Finally, at t_4, the ACK message arrives to PC_0, being processed at t_5 and winning all the ResArb modules and confirming the circuit at PC_0. The ACK message is forwarded to the MC, confirming the circuit at the NIC.

If one of the resources is not available during the circuit setup time, the affected PC will generate a NACK and will enqueue it into the NACK queue. Resources will be freed the next cycle and the NACK message will be transferred to the previous cluster. If a NACK is received at the NIC, the packet will be injected using packet switching (PS).

At time t_{11}, Delta T at ResArb module associated with R_{0E} reaches 0, thus the signal *Circ_to_W* (in R_{0E}) is activated and set to point to the local port. This signal switches in input port from MC and output port E at PR_0. This circuit will last the required number of cycles for the message (Delta T'). All other programmed output ports (R_{1E}, R_{2E}, R_{3S} and R_{7L}) switch the input and output ports making the circuit just for Delta T' cycles. In this cycle MC receives the block from memory and injects the first flit. The flit is forwarded and crosses the entire network reaching $L2$ output port at $PR7$. When Delta T' reaches 0, the circuit is torn down in a distributed manner on each router just when the last flit of the message crossed the router.

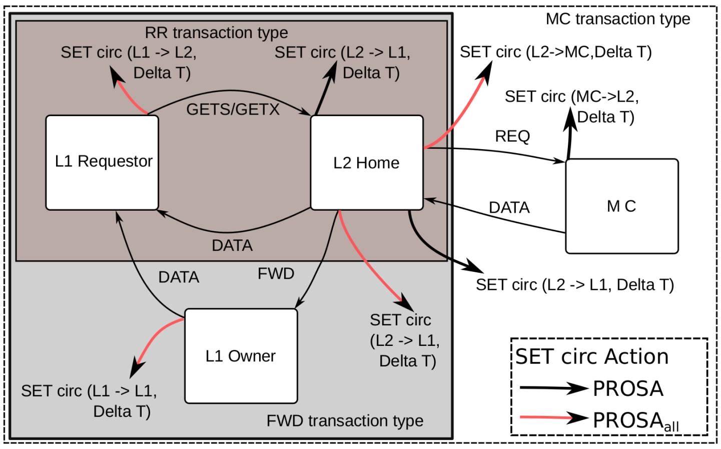

MC transaction type
RR transaction type
SET circ (L1 -> L2, Delta T)
SET circ (L2 -> L1, Delta T)
SET circ (L2->MC,Delta T)
SET circ (MC->L2, Delta T)
GETS/GETX
L1 Requestor
L2 Home
REQ
DATA
M C
DATA
DATA
FWD
SET circ (L2 -> L1, Delta T)
SET circ (L2 -> L1, Delta T)
L1 Owner
SET circ (L1 -> L1, Delta T)
FWD transaction type
SET circ Action
PROSA
PROSA$_{all}$

FIGURE 7.11: Transaction diagram for PROSA$_{all}$ coherence protocol.

7.4.2 PROSA Enhancements

As mentioned above, different design alternatives exist when dealing with CS. To analyze them, we extend the baseline PROSA design with four enhancements. First, all the protocol messages (including single-flit messages) will be considered for the setup and usage of circuits (we name this method as PROSA$_{all}$). The fact that circuits, when used, are much faster than the standard PS network suggests that a small penalty can be paid in the circuit setup process. In the second enhancement, we will allow a small slack to the circuit setup process (we name this method as PROSA$_{slack}$). As ACK messages may contend in the PC some delays may be incurred. A single cycle delay will ruin the circuit as the message will not use it. Therefore, by adding a small slack more circuits will be effectively used. The third enhancement refers to the use of circuits only for messages with destinations located closer than a given threshold. The larger the circuit the larger the penalty in setting up the circuit. Different thresholds will be analyzed (we name this method as PROSA$_{dd}$; distance driven). Finally, we will enhance baseline PROSA with different priority schemes in the PCs when dealing with ReqCir, ACK and NACK messages. This will affect the success rate of circuits being used. This method will be named PROSA$_{priorities}$.

7.4.2.1 Coherence Protocol Extension

First enhancement is the use of circuits for all protocol messages. Figure 7.11 shows the new diagrams for the PROSA$_{all}$ protocol. All events (control and data messages) request a setup circuit process by triggering the SET_{CIRC} action. New requests are plotted in red. In ReqResp transactions, the L1 cache requests a circuit between L1 and L2 to send a GETS/GETX message. If the circuit is confirmed then the GETS/GETX message is sent using CS. Otherwise, the message is sent using PS. When the GETS/GETX

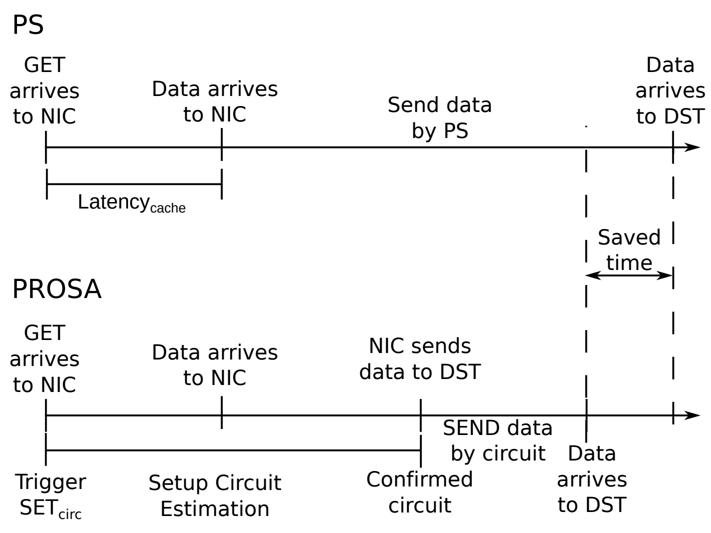


FIGURE 7.12: PROSA$_{all}$ and PS timings.

message arrives to the L2 cache it follows the baseline PROSA behavior. A speculative circuit is setup between L2 and L1.

In MC transactions the L2 Home cache requests a circuit between L2 and MC to send the REQ message. If the circuit is confirmed the message is sent through the circuit, otherwise is sent using PS. Upon arrival, the MC sets a circuit for the incoming data to be delivered to the L2.

Finally, in FWD transactions, when a load or store misses in L1, a new circuit is requested between L1 and L2 trying to setup a new circuit for the GETS/GETX message (as is the case for ReqResp transactions). When the request reaches the L2 and the block is in P state, a new circuit is requested between the L2 and the L1 owner to send the FWD message. When the message reaches the L1 Owner, a final circuit is set to send the block to the original L1 requestor.

Every SET_{CIRC} action is classified either as speculative or predictable. Predictable actions occur between the L1 and the L2 cache, between the L1 owner and the L1 requestor, between the L2 cache and the MC controller and between the L2 and the L1 owner (FWD transaction). Speculative actions occur when a GETX/GETS reaches the L2 cache, triggering a request from L2 to L1 requestor.

7.4.2.2 Slack on Circuit Setup Process

Figure 7.12 shows the timings of a message injected through PS and through PROSA. The case refers to an speculative circuit in ReqResp transactions. As we can see, when using PROSA, upon arrival of a message to the network interface a new circuit setup process is triggered. During this process the message is processed and the cache is accessed. Notice, however, that the confirmation message (ACK) may be delayed some cycles as it may encounter contention within the PCs. Therefore, before the circuit is confirmed the data is ready to be injected through the network. With standard PROSA, the circuit would fail and the message would then be injected through the network in PS mode. However, now in PROSA$_{slack}$, a slack of time is allowed to avoid this problem. Thus, the NIC will wait more time for the confirmation of the circuit. Once the confirmation arrives the message is injected through the network in CS mode. Notice that depending on the slack and on the relative speed of the circuit compared to the network in PS mode, PROSA$_{slack}$ may still deliver the message faster.

We define a slack constant ($Slack_{cir}$) which is assumed by the PC and by the NIC injector. Notice that by using this slack the number of circuits successfully used increase, potentially increasing performance. Also, notice that the confirmation of a circuit may be received before the slack expires (if no contention is encountered by the ACK message). In that case, the message is injected using the circuit as early as the confirmation arrives and the slack does not expire. Notice that the circuit will be programmed in the network for the transmission time plus the slack constant.

The latency on a circuit setup process is determined by the distance to destination and the slack constant:

$$Setup_{latency} = MAX(cache_{latency}, d * 2 + Slack_{cir})$$

Where $cache_{latency}$ is the cache access latency, d is the distance in hops in PROSA (number of PCs crossed) and $Slack_{cir}$ is a small delay due to the contention in the PROSA network. Notice that this formula can get $Setup_{latency}$ smaller than $cache_{latency}$. As an example, if d equals 0 when source and destination are in the same PC and $Slack$ is smaller than $cache_{latency}$, in this case, the $Setup_{latency}$ is set equal to $cache_{latency}$.

When a successful circuit is configured, PROSA$_{slack}$ guarantees the circuit transmission time ($Setup_{latency}$ plus $CS_{latency}$) is lower than the message transmission time ($cache_{latency}$ plus $PS_{latency}$).

7.4.2.3 Distance Driven Setup Circuit

As a further enhancement, PROSA$_{dd}$ allows selective configuration of the circuits based on the distance to destination. Thus, when a circuit is requested to be configured, PROSA$_{dd}$ analyzes the $Setup_{latency}$ which is mainly determined by the number of PC hops. A $MaxHops$ parameter can be configured in PROSA$_{dd}$. If the distance is larger than $MaxHops$, then the circuit is cancelled and the data is forwarded through the network using PS. Later we analyze the behavior of PROSA$_{dd}$ with different thresholds.

7.4.2.4 PROSA$_{dd}$ Messages Priorities

As a final enhancement, PROSA$_{priorities}$ enables the use of different priorities between ReqCir, ACK and NACK messages within the PC network. In the standard PROSA, NACK messages get higher priority than ACK messages, and ACK messages get higher priority than ReqCir messages ($NACK > ACK > ReqCir$). This priority scheme guarantees all NACK messages get delivered to end points. However, many circuits cannot be set due to a conflict with NACK or ACK messages. Remind that ReqCir messages convert to NACK messages if they conflict with a NACK or ACK message.

Now, in PROSA$_{priorities}$, the endpoints will inject the message through the network in PS mode when the $Setup_{latency}$ expires. Also, PCs will remove circuits automatically. Thus, there is no need to guarantee all NACK messages are delivered to the end points. Indeed, NACK messages could be simply removed from the network. The only benefit they produce is that a message can be injected earlier into the network once it receives the NACK message (before the $Setup_{latency}$ expires).

Thus, in PROSA$_{priorities}$ we can use two different priority schemes. In the first one, ACK messages get now highest priority, so to speed up injection of messages through circuits, then ReqCir messages get higher priority than NACKs, which have the lowest priority ($ACK > ReqCir > NACKS$). The second one is more radical as NACK messages will simply be removed from the system. They will not be generated by PCs. In this scheme, ACK messages get higher priority than ReqCir messages ($ACK > ReqCir$).

Notice that removing the NACK support we ease the design of the PC. Figure 7.13 shows the new PC structure. In particular, the comparator column and the NACK queue structure and associated logic have been removed. Also, the TYPE field is reduced to only two types (one bit encoding). Also the logic at RCs is greatly simplified as partial circuit elimination is not longer needed when NACKs do not exist. This means the area and power overheads of PROSA will be reduced (will be presented later).

7.5 PROSA Evaluation

For the performance analysis we use gNoCsim [66], an event-driven cycle-accurate simulator that models any network topology and router architecture. We model a 2-stage pipelined router with VCs and flit-level crossbar switching, as used in Garnet [67]. Table 7.1 shows the simulation parameters for the router and the cache hierarchy. L1 is private to the core and L2 is shared but distributed among all tiles.

In a first analysis, we evaluate three mechanisms: the baseline router (BASELINE), the Dejavu (DEJAVU) solution [14] and PROSA (no optimizations involved). In PROSA data sent through circuits take one cycle (using SMART). We analyze applications from

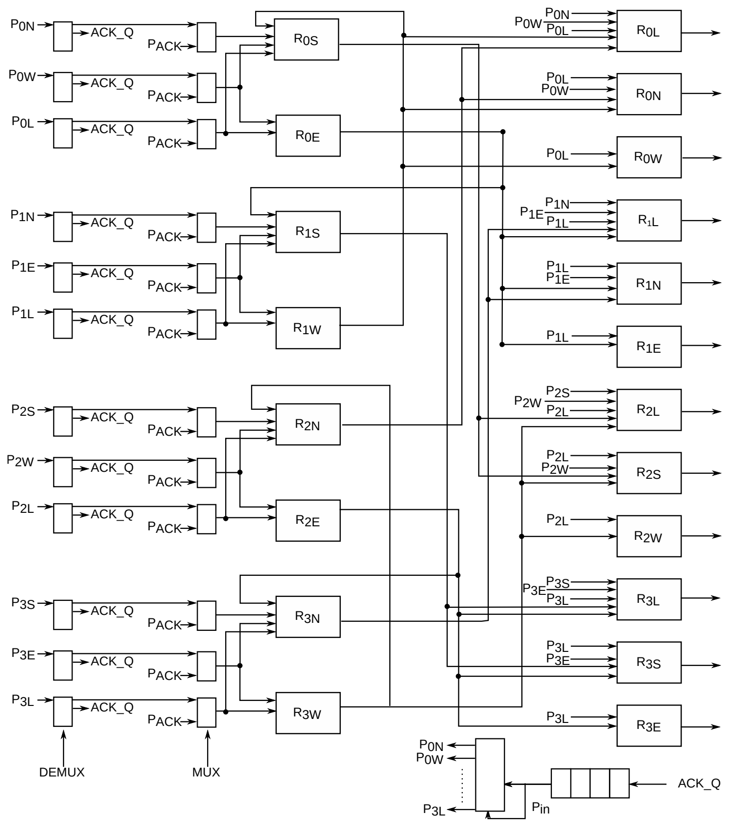


FIGURE 7.13: PROSA Controller without NACKs.

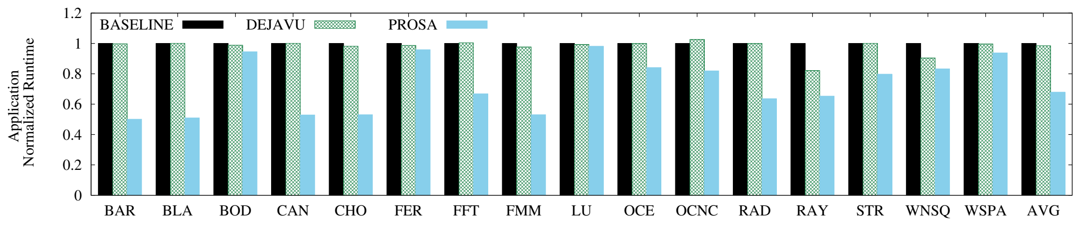
Parameter	Network	L1	L2 per tile	MC
Topology	8x8 mesh			
# VCs/fly link	4/1 cycle			
Message sizes	8/72 bytes			
Flit/Queue size	64 bits/9 flits			
sets/way/line size (B)		32/4/64	128/16/64	
cache/tag latency		2/1	4/2	
# MCs				2
Memory controller delays (open/activate/read rows)				16/16/16

TABLE 7.1: Parameters and values used for routers and caches.

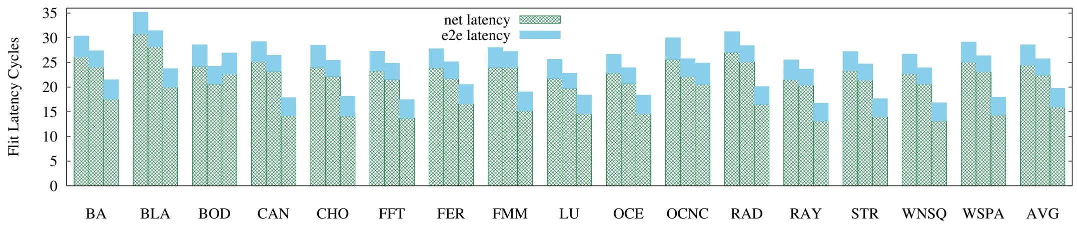
SPLASH [68] and PARSEC [69]. Table 7.2 shows the applications with their observed loads. Deja Vu pre-allocates the circuit between nodes in order to hide the setup latency by dividing the NoC in two planes: control and data plane. The control plane is in charge of configuring the circuits. This plane has higher voltage and frequency, so being faster. In this NoC, the request packet pre-allocates the path in backward direction as it approaches destination. The destination node can forward the response to the requestor whenever data is ready with no circuit setup. This approach can produce conflicts. Deja Vu configures the circuits in the order they are reserved.

Application	Benchmark	Abbr.	Runtime	L1 miss	L2 miss
BARNES	SPLASH	BAR	High	Low	Med
BLACKSCHOLES	PARSEC	BLA	High	High	High
BODYTRACK	PARSEC	BOD	Med	Low	Low
CANNEAL	PARSEC	CAN	High	High	High
CHOLESKY	SPLASH	CHO	Med	Low	High
FERRET	PARSEC	FER	Low	Low	High
FFT	SPLASH	FFT	High	Low	Med
FMM	SPLASH	FMM	Med	Med	High
LU	SPLASH	CHO	Low	Low	Low
OCEAN	SPLASH	OCE	Low	Med	High
OCEANNC	SPLASH	OCNC	Low	High	Low
RADIX	SPLASH	RAD	Med	High	High
RAYTRACE	SPLASH	RAY	Med	Med	Med
STREAMCLUSTER	PARSEC	STR	High	Low	High
WATERNSQ	SPLASH	WNSQ	Med	Med	High
WATERSPACIAL	SPLASH	WSPA	Med	Med	High

Table 7.2: Applications tested with observed runtime and L1/L2 miss.



(a) Application Runtime



(b) Flit Latency

Figure 7.14: Performance results for different architectures (BASELINE, DEJAVU, PROSA in column order).

7.5.1 Results

Figure 7.14a shows application runtime. PROSA reduces, on average, BASELINE application runtime by 33.11%, reaching lower gains in applications with a low number of L2 misses (BODYTRACK), or with short application runtimes (LU) where the reduction in runtime is 5.45% and 4.0%, respectively. However, with balanced applications (OCEANNC), or applications with a high number of MC requests (FFT, CANNEAL, FMM, BLACKSCHOLES) PROSA reaches an improvement up to 50%. Deja

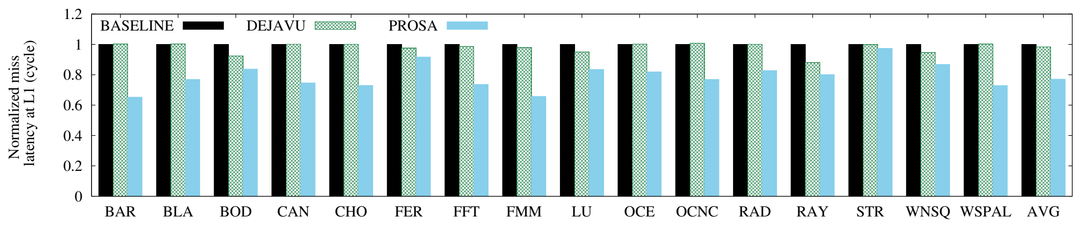


Figure 7.15: Memory latency results for different architectures. Normalized to baseline case.

Vu achieves negligible benefits in performance, as seen in [14].

Figure 7.14b shows latency results. DEJAVU outperforms BASELINE by 10% on average. PROSA, in applications with a low number of MC requests and small datasets (e.g. BODYTRACK), achieves a 7% improvement on end-to-end latency. However, PROSA outperforms BASELINE up to 39% on applications with a high number of L2 misses (e.g. CANNEAL). On average, PROSA outperforms BASELINE end-to-end latency by 31.14%. For network latency, blue solid color, on average, PROSA outperforms BASELINE by 34.16%. Notice PROSA outperforms significantly DEJAVU.

For L1 miss latency normalized to BASELINE (Figure 7.15), as it occurs with runtime, DEJAVU slightly improves BASELINE and PROSA achieves better performance. On average, our proposal outperform BASELINE by 23%.

	REC.ACK	REC. NACK
MIN	(OCNC) 85.98%	(LU) 4.28%
AVG	90,62%	9,38%
MAX	(LU) 95.72%	(OCNC)14.02%

Table 7.3: Number of ACK and NACK messages generated in PROSA in PC module.

	CONF. INPUT		CONF. RA		
	NACK	ACK	FB	TMP	ARB
MIN	(LU) 0.01%	(BOD) 0.63%	0%	(WSPA) 1.92%	(WSPA) 0.31%
AVG	0,04%	3,94%	0,00%	4,61%	0,79%
MAX	(WSPA) 0.18%	(WSPA) 9.27%	0%	(OCNC) 9.04%	(OCNC) 1.70%

Table 7.4: Number of conflicts generated in PC module.

Table 7.3 shows statistics about PROSA circuits. The first column shows the received ACKs (the number of circuits established successfully). In all cases this is higher than 85.58% and the average is 90.62%. The second column shows the received NACKs and is the sum of the next five columns of Table 7.4, which list the different types of conflicts in the PC. The first and second column show the number of NACKs generated at the input of the PC as a normal request and an ACK/NACK conflicted at the same time. The last columns show conflicts generated at the ResArb arbiters due to (1) the register table being full (FB), (2) the required period of time of one request overlaps with one

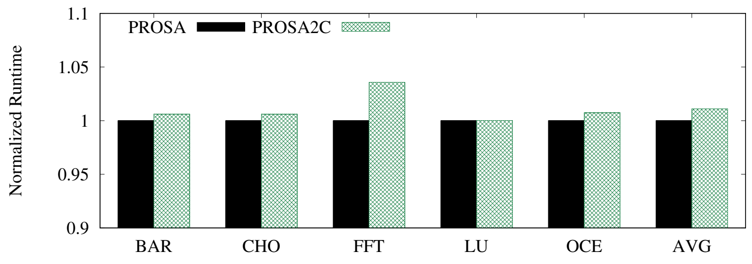


FIGURE 7.16: Results for PROSA with delay circuits equal to 2.

established circuit (TMP), and (3) two requests collide in the same ResArb module (ARB) due to the static arbiter. As we can see, most conflicts are generated because of temporal conflicts in generating circuits or because of concurrent requests conflict with an ACK in the same resource. However, the current table size at ResArb modules (2 entries) seems to be properly sized (even could be reduced thus saving more area) as the number of conflicts due to the table being full are negligible.

One critical aspect of PROSA is the technological ability to transmit via circuit one flit from one network corner to the opposite. To analyze the impact of a more relaxed circuit design, 2-cycle circuits have been tested (PROSA2C). Figure 7.16 shows, for some applications, the runtime application results assuming two 2-cycle circuits (for every source-destination pair). As can be seen, PROSA2C reaches similar results as PROSA, on average worsening performance by 1.2%. As both configurations reach similar results, one cycle delay will be assumed for the rest of the paper.

7.5.2 PROSA Implementation

We have implemented all the PROSA infrastructure for a 4×4 CMP system. Each PC module has been implemented in Verilog and tested. We use a canonical router design with 64-bit flits, four 9-flit depth VCs, and with seven ports to attach 2D-mesh ports and L1, L2, and MC (including the MLCU). We use Design Vision tool from Synopsys with 45nm Nangate open cell library [70]. Power results are obtained from Orion-3 power library [71].

Table 7.5 shows the area overheads of PROSA for different configurations. In all of them, and for the sake of comparison, we also account for the components to build a cluster of four routers. In the case of PROSA we consider all the components (including the PC and the four PRs) and the logic and resources used for NACK messages.

As we can see, the PROSA router takes only 1.8% more area than the baseline router. The PROSA controller (PC) takes less area than a baseline router (22% of baseline router area). However, this component is new and needs to be considered as an additional overhead. To make this comparison fair, the table shows area of cluster regions. In this

Configuration	area (μm^2)	overhead
Baseline router	243784	-
PROSA Router (PR)	248200	1.8%
PROSA Controller (PC)	76547	-
Baseline cluster	975136	-
PROSA cluster	1069347	9.66%
Flattened Butterfly cluster	1380109	41%
2xBaseline cluster	1658560	70%

TABLE 7.5: Area overheads for different router and NoC organizations.

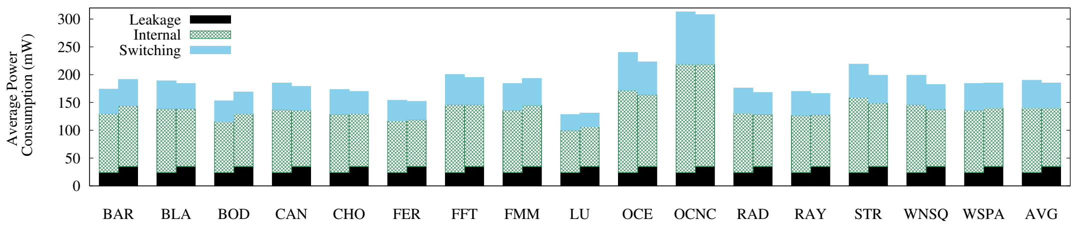


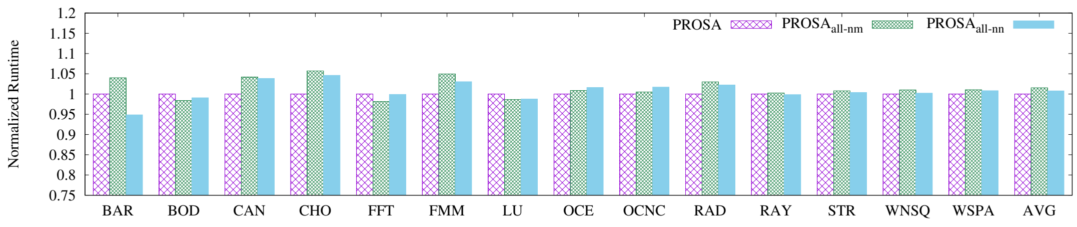
FIGURE 7.17: Power consumption results. First column represents BASELINE and second PROSA.

case, the PROSA cluster takes 9.66% more area. The NIC's overhead can be considered as negligible.

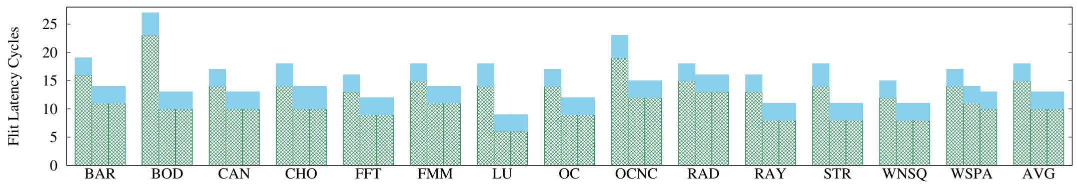
PROSA overheads can be seen as high. However, we should consider the performance gains that PROSA circuits enable. In order, however, to better assess the overheads, the table shows two additional configurations worth being analyzed. The first one, $flattenedbutterfly$, is the overhead for a flattened butterfly topology which has more connectivity along each dimension and direction. In this case, because of the larger number of input ports, the area overhead is increased by 41%. The second one is for the case where the baseline cluster is enhanced with double flit size. This can lead to faster transfer times between the nodes. However, as we can see, overhead skyrockets to 70% additional area. This is mainly due to the larger buffer requirements.

Figure 7.17 shows the power consumption results for the entire network, PROSA and PS network. Although the leakage power increases by 42% the total power consumption is reduced by 3%. This reduction is due to switching and internal power, which are reduced by 10%. As described in [14] Deja Vu achieves a 30% power reduction, which is similar to our results (but without the latency improvements). Notice that the 30% execution time reduction will translate in major power savings as shown in [72][1].

[1]In [72] authors show that in modern computing systems, DVFS gives much more limited power savings with relatively high performance overhead as compared to running workloads at high speed and then transitioning into low power state.



(A) Application Runtime



(B) Flit Latency

FIGURE 7.18: Performance results for different architectures (PROSA, PROSA$_{all-nn}$, PROSA$_{all-nm}$ in column order).

Notice that in some applications (BAR, FMM, LU, WSPA) overall power consumption is slightly higher with PROSA. In these applications, internal and switching power consumption is roughly the same with PROSA and BASELINE (due to the higher number of delayed circuits that could not be used). This fact, combined with PROSA incurring in higher leakage (due to its additional network), makes the overall power consumption to be slightly higher . Notice, however, PROSA targets energy savings by reducing application runtime.

7.5.3 Enhanced PROSA

In this section we analyze the performance achieved by PROSA with the different added functionalities presented before. First, we analyze PROSA$_{all}$ when circuits are configured for all messages. In this version, no slack is provided nor distance threshold. Different priorities between PROSA messages will be explored. Then, we focus on PROSA version with distance thresholds and slack for circuit setup process. This method is called PROSA$_{all-dd-slack-xy}$ where x and y will identify threshold and slack values.

7.5.3.1 Circuits for all Messages and Different Priorities

Figure 7.18 shows the comparison between baseline PROSA and PROSA$_{all}$ with two different versions. The first one (PROSA$_{all-nn}$) with no NACKs on the PROSA circuit network, and the second one (PROSA$_{all-nm}$) with NACKs with lowest priority ($ACK > ReqCir > NACK$). Figure 7.18a shows the normalized execution time for different applications. Notice the Y axis which does not start from zero value (for the

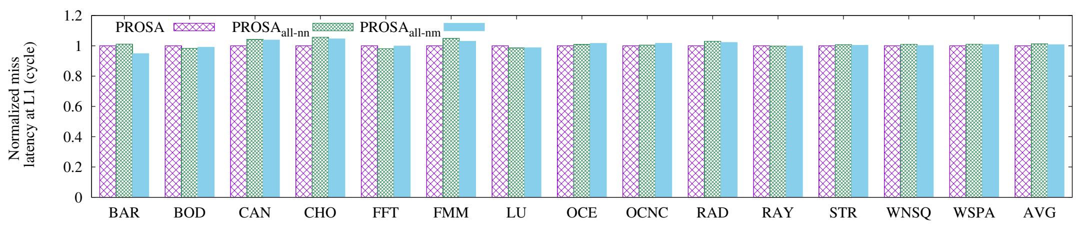


FIGURE 7.19: Memory latency results for different architectures. Normalized to baseline case.

sake of clearly showing differences). As we can see, differences in execution time are very small. However, when comparing flit latency (Figure 7.18b) differences are much more noticeable. Both PROSA configurations (PROSA$_{all-nn}$ and PROSA$_{all-nm}$) reduce baseline PROSA flit latency by 35%. In flit latency PROSA$_{all}$ benefits range from 67% (in applications where PROSA achieves a small improvement in terms of flit latency (see Fig. 7.14), e.g. BODYTRACK) to 18% or 32 % (in applications in which PROSA gets higher improvement in that metric (see Fig. 7.14), e.g. RAD or CANNEAL and FFT respectively.

It has to be noted that although flit latency is significantly improved, the execution time of applications barely changes. This effect is due to the extra delay paid by PROSA$_{all-nm}$ and PROSA$_{all-nn}$ at the protocol level. Indeed, by tunneling all messages through circuits some protocol messages are delivered out of order, triggering race conditions in the coherence protocol. Those races have been fixed by adding additional protocol states or by recycling some protocol requests in order to enforce the strict ordering of messages. Indeed, Figure 7.19 shows the average L1 miss latency results. As we can observe, the different PROSA versions achieve almost identical miss latency values.

More interesting is the fact that both new versions achieve almost the same flit latency values. Indeed, lowering the priority of NACK messages has the same effect as of removing them completely. This result suggests that NACK messages can be removed as they will hardly affect performance. Indeed, some area and power savings will be achieved by removing NACK support. Table 7.6 shows the overhead implementation of the PC without NACKs. As can be seen PC and cluster area is reduced by 16% and 2% respectively. Thus, PROSA$_{all-nn}$ reaches similar performance results as PROSA$_{all-nm}$ but reducing the area.

Configuration	area (μm^2)	overhead
PROSA Router (PR)	248200	
PROSA Controller (PC)	76547	
PROSA$_{all,nn}$ Controller (PC)	64918	-16%
PROSA cluster	1069347	
PROSA$_{all,nn}$ cluster	1057718	-2%

TABLE 7.6: Area overheads for PROSA controller without NACK .

We concluded PROSA$_{all-nn}$ and PROSA$_{all-nm}$ being almost identical in performance was in average terms. However, for every application results are slightly different and significant in BAR. The trend is that not using NACKs increases performance. In BAR, the pollution created by NACK messages leads to critical circuits to get delayed and thus miss shorter message latencies. This is, on average, cancelled due to the large amount of traffic each application injects. However, delaying a process punctually will lead to a higher execution time because of barriers and synchronization events between processes. This is the root cause of higher runtime but similar average latencies.

			CONF. INPUT		CONF. RA		
	REC.ACK	REC. NACK	NACK	ACK	FB	TMP	ARB
MIN	(OCNC) 62.81%	(LU) 14.36%	0	(STR) 5.25%	(BOD)0.02%	(LU) 5.75%	(LU) 3.15%
AVG	73.25%	26.75%	0	7.84%	0.18%	12.30%	6.71%
MAX	(LU) 86.64%	(OCNC) 37.21%	0	(BOD) 17.15%	(FMM) 0.55%	(OCNC) 15.96%	(OCNC) 9.41%

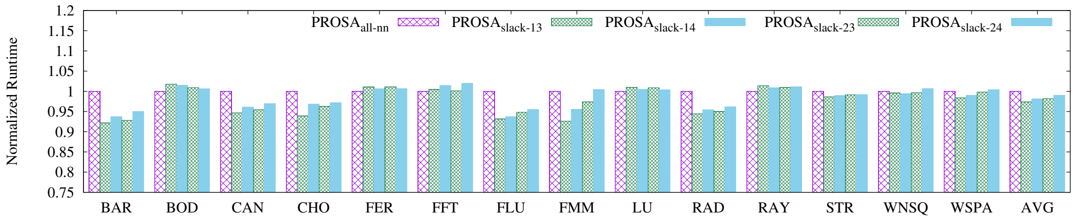
Table 7.7: Number of ACK and NACK messages generated in PROSA$_{all-nm}$ and number of conflicts generated in PROSA Controller.

Table 7.7 shows statistics about PROSA circuits for PROSA$_{all-nm}$. Notice that the current circuit setup success rate (73.25% on average for all applications) is much smaller than the one achieved with baseline PROSA. This is due to the higher traffic of the PROSA$_{all}$ network to establish circuits. Now, for every message a circuit setup process is launched. However, what is noticeable is that NACK messages do not introduce any conflict at the input of the PC device. Full bank conflicts at ResArb modules is also low (0.18% on average). The more prominent conflicts now are those related to the temporal conflict with already programmed circuits (12.30% on average) and due to arbiter conflicts (two concurrent requests, 6.71% on average). Notice that in this configuration each ResArb module implements a circuit table with four entries. These results are similar to the ones achieved by PROSA$_{all-nn}$. Because of the more efficient implementation, from now, we use PROSA$_{all-nn}$ for all the following experiments.

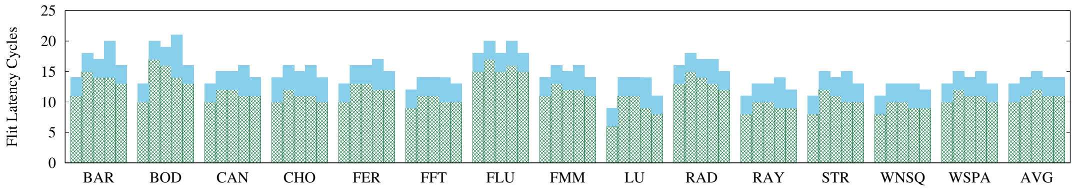
7.5.3.2 Bounded Circuits and Slack

Now we focus on the full deployment of PROSA, which includes the distance threshold and the slack for circuit setup process (PROSA$_{all-dd-slack-xy}$). We analyze two slack (x) values (1 cycle and 2 cycles) and two distance (y) thresholds (3 and 4 hops in PC, 6 and 8 routers, respectively). Thus, PROSA$_{all-dd-slack-13}$ represents PROSA with 3 PC hops distance threshold and 1 cycle for slack. All these versions do not include NACKs (similar to PROSA$_{all-nn}$).

Figure 7.20a shows the application runtime for the four new configurations and for the previous PROSA$_{all-nn}$ version. On average, all these four configurations slightly improve PROSA$_{all-nn}$. Among all, PROSA$_{all-dd-slack-13}$ is the best configuration, which outperform applications execution time up to 6% in some applications (CANNEAL, CHO, RADIX). On average performance improves by 3%.



(A) Application Runtime



(B) Flit Latency

Figure 7.20: Performance results for different architectures (PROSA and PROSA DD in column order).

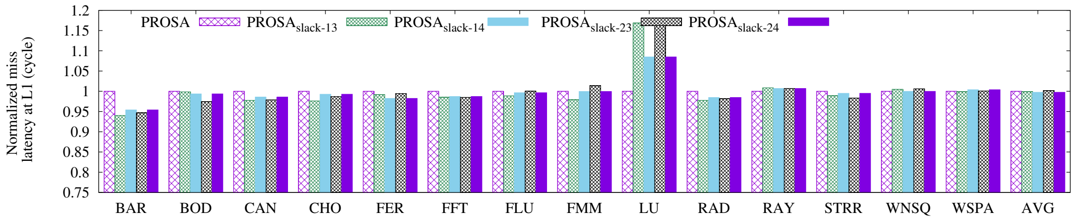


Figure 7.21: Memory latency results for different architectures. Normalized to baseline case

Figure 7.20b compares flit latencies for these configurations. PROSA$_{all-nn}$ always gets the lowest flit latency for all applications. As can be seen, PROSA$_{all-dd-slack-xy}$ gets worse latency results with low threshold value. However, this does not affect L1 miss latency . Figure 7.21 shows the miss latency results which follows the same trend observed for the execution time of applications. Applications with higher miss latencies (LU) have higher execution runtimes. As it occurred with PROSA$_{all-nn}$, improvements in the network (faster messages) is not reflected in execution time of applications due to the extra delay incurred in the protocol.

	3 HOPs		4 HOPS		5 HOPS		unbounded	
	ACK	NACK	ACK	NACK	ACK	NACK	ACK	NACK
BARNES	77.52	22.48	72.78	27.22	70.51	29.49	68.25	31.75
CHO	80.31	19.69	76.90	23.10	75.09	24.91	73.03	26.97
FFT	83.38	16.62	80.10	19.90	77.61	22.39	75.41	24.59
OCEAN	84.44	15.56	80.19	19.81	77.29	22.71	75.24	24.76
LU	91.20	8.80	88.65	11.42	86.24	13.76	84.27	15. 73

Table 7.8: Percentage of ACKs/NACKs for PROSA$_{all-dd}$ with different distances.

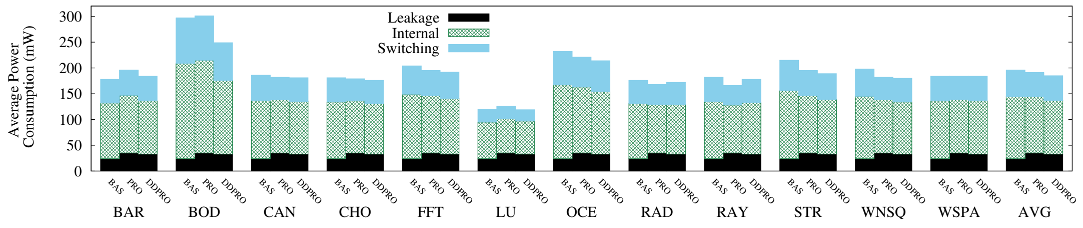


FIGURE 7.22: Power consumption results. First column represents BASELINE, sedond PROSA and third PROSA$_{all}$.

Table 7.8 shows PROSA statistics for different applications when varying the maximum distance of circuits inside the PC network. As we observe, the number of NACKs increases as the allowed distance of circuits increases. This is mainly motivated by the traffic increase in the PC network. As seen in the table more NACKs are generated, delaying injection of associated messages. Therefore, maximum distances of 3/4 PC hops looks like the right approach to minimize NACKs while still guaranteeing long circuits.

Finally, Figure 7.22 shows the power results for PROSA$_{all-dd-slack-xy}$. On average, the new PROSA version reduces the power consumption by 7% and 4% to BASELINE and baseline PROSA, respectively. Network traffic in the PC network in PROSA$_{all-dd-slack-xy}$ is 5 times larger than the observed in PROSA. However, the new PROSA reduces power consumption in the standard network is reduced by 35% on average.

7.6 Discussion

From the previous results we can observe different aspects worth being highlighted. The first one is the clear benefit of building an infrastructure to setup circuits for coherence protocol-oriented systems. By taking advantage of large memory access latencies the network can be configured with communication circuits configured and kept only for the transmission duration of the message. PROSA is able to achieve remarkable results in execution time and flit latencies. PROSA can set a circuit in less than 16 cycles in the worst case for an 8×8 mesh network. Although this can be enough for standard memories, for faster ones or larger networks maximum circuit distance has to be bounded.

The second conclusion is the fact that tunneling all the messages (short and long) through circuits does not necessarily lead to benefits in application performance. Execution time is barely the same and, most important, the coherence protocol must be adapted to support new race conditions triggered by messages arriving out of order. What is more interesting to note is the fact that negative acknowledgments of circuits can be discarded and it is worth relying on an automatic and silent circuit tear down process. Further savings in power and area implementation are achieved. Indeed, performance is not significantly affected.

A third conclusion we obtain is related with the circuit setup delay and its effects on performance. Indeed, by bounding the reachability of circuits performance of applications can be improved with a further 6%. This is easily achieved by adding a simple comparator on each NIC. Longer circuits tend to have a larger circuit setup time, which indeed is larger than the cache access time, incurring on a extra delay to send the message through a circuit.

Finally, the use of a slack for circuits should be introduced with care. A large slack may lead to worse results as it induces more temporal conflicts in the network. Indeed, contention in the circuit setup network is low and, thus, with only one cycle of slack most of the circuits are successfully configured and used, leading to good performance.

One critical aspect of PROSA is its scalability. First, the current design allows only one ReqCir request to access the RT. This may create a bottleneck. However, the low/medium load traffic seen on the PC network does not compromise its performance. Anyway, if scalability problems appear, it would be solved by implementing a two-port RT module. On the other hand, for PROSA$_{all}$, small-sized messages with distant destinations could prevent long-sized messages with nearer destinations, potentially affecting performance. However, some of those small-sized messages are in the critical path of memory access by the processor, thus, having a high impact on performance. This fact makes the distinction between short and long messages less significant.

As a final conclusion, the best PROSA configuration would be the one with no NACK support, one cycle slack, and with a threshold of distance three for the 8×8 network.

7.7 Related Work

Circuit-switching [64] has been used in a number of previous works in NoC architectures in order to reduce on-chip communication latency. Once a circuit is set, data does not travel through the routing and arbitration stages on each router. However, setup time usually causes low resources utilization and performance degradation. On the other hand, packet switching improves resource utilization and network performance, splitting the entire message in smaller blocks and forwarding them along the network.

Some works try to get benefit from both mechanisms by implementing a hybrid circuit-packet switching strategy. Kumar [9] proposes Express Virtual Channels (EVC) allowing packets to bypass intermediate routers along their path. EVCs only allow to connect nodes along the same dimension, so circuits cannot turn from one dimension to another. PROSA, on the other hand, allows to connect nodes regardless their location, thus offering more flexibility.

Jerger [10] proposes circuit switched coherence, setting permanent circuits between pairs of frequent data sharers instead of tearing them down. It allows to quickly send data between the same nodes. However, if another circuit requires the resource, the data is switched to packet switching until it reaches destination. Yin [11] proposes a hybrid circuit-packet switched network in which the packet is forwarded along the packet network while the circuits can be set in parallel, using TDM. Yim's proposal also expends time in the setup latency. Mazloumi [12] proposes another hybrid packet-circuit switched router. This mechanism setups the circuit along the network while the request message is being forwarded between the requestor and the destination. When the request reaches the destination and the data is ready, the mechanism sends a probe message activating the reserved circuit, after that the data is sent. Chen [13] proposes an implementation of a hybrid circuit-packet switching. All these mechanisms require a setup period. However, PROSA hides the setup circuit latency based on the coherence protocol and when the circuit usage finishes, resources are freed allowing the packet switching to forward packets without having to tear down any circuit, data is sent without delay.

Abousamra [14] proposes Deja Vu (briefly described in Section 7.5). In Deja Vu, the selected order schema can produce underutilization of network resources. In [15] authors alleviate the problem by using a different order. However, it still requires the high frequency and voltage control plane. PROSA, as Deja Vu, preallocates the circuit in order to hide the setup circuit delay, However, as per our evaluation, PROSA achieves better performance results. Van Lear [16] proposes a coherence-based message predictor for optical interconnection networks. In the proposal a global predictor establishes the circuits between nodes. All the traffic in the network has to cross the predictor, thus potentialy causing a bottleneck in the network. This proposal is also for optical interconnects where a full optical crossbar is assumed. This makes scalability a major issue. Shacham [17] proposes an hybrid optical-electrical network. In this proposal the electrical network is used to establish the optical circuit. This mechanism is not based on coherence protocols and uses photonic networks. Krishna in [7] presents SMART, a multihop network with single-cycle data-path all the way from source to destination. Setup circuit is required one cycle before the data is sent and partial circuits can be established. An extra network is required to send SMART-hop Setup Request, SSR. This mechanism is not based on coherence cache as PROSA and our mechanism relies on SMART circuits. Peh [18] presents flit-reservation flow control. In this proposal the circuit is setup hop by hop. However PROSA uses clusters to setup circuits, improving the time required to establish the circuit. PROSA does not require to buffer messages, contrary to flit-reservation. Our proposal anticipates the circuit setup process.

Liu et al. [19] propose an effective setup circuit procedure, in which the setup procedure is guaranteed to terminate in 3D+6 cycles, where D is the distance. Xue et al. [20] propose a general mathematical framework for reconfigurable networks in order to optimize the network by configuring subnetworks to transmit data. Hollis et al. [21]

propose a reconfigurable NoC using the Skip-link. This proposal configures the Skip-link to allow packets to bypass the channel, avoiding to cross the router. PROSA implements a cluster approach instead. Moreover, those proposals do not get any benefit from the coherence protocol delay.

A cluster approach has been proposed previously by some authors. Xue et al. [22] propose a cluster approach to send multicast messages, encoding these messages and decoding them at destination. They support dropping parts of the message. Qian et al. [23] propose a cluster approach by connecting clusters through Express Virtual Channels (EVC) and a Hub router. They use an adaptive routing algorithm to choose between regular network, the EVC or the Hub router. Both approaches implement a packet switching approach and do not use the coherence protocol information to improve network performance.

Pimpalkhute et al. [80] propose a packet classification scheme for CMPs. The NoC routers maintain the Used Back Row table. This table is used to forward incoming request to main memory. If the incoming request has the same bank and row destination, the request is forwarded and bypasses all the stored requests. In our proposal MLCU follows the same behavior as Used Bank Row table. However, MLCU is used only for latency estimation.

As a summary, PROSA allows to establish circuits between any pair of nodes hiding the setup latency. PROSA uses the coherence protocol information to establish these circuits. In addition, PROSA programs circuits for their exact period of time they will be needed, thus not conflicting with other circuits using the same resources (in other time periods). Indeed, PROSA circuits can be steered by coherence protocols or even for other higher-level applications where traffic bursts can be predicted and requested ahead of time. The previous works do not rely on coherence protocols or they rely on more expensive architectures and technologies.

7.8 Conclusions

In this paper we introduce PROSA, a circuit-switched enabled NoC architecture which allows the coherence protocol to steer circuit connections for future predictable and speculative connections between memory controllers (MC), L2 cache banks and L1 caches. The baseline PROSA system establishes circuits for the required time period between L2 and L1 caches and between MC and L2 banks. On a misspredict the circuit is silently removed. PROSA builds a separate infrastructure to manage circuits, thus the NoC network is not affected by the additional circuit setup traffic.

Network latency is reduced up to 35% by correctly predicting and using circuits with PROSA, outperforming baseline designs by reducing application runtime by 33% and improving DEJAVU with similar power results. PROSA takes an area overhead of 9.66% compared to baseline but it saves more than 3% in power consumption.

Also, further enhancements of PROSA are presented: 1) the coherence protocol allows to send all messages via circuits, 2) modifications at arbiters to set priorities, 3) a new threshold mechanism to limit distance of circuits, and 4) removal of NACK messages with implicit teardown circuits logic implemented. Compared with baseline PROSA, the new method results in a reduction of network latency up to 35% by correctly predicting and using circuits. The new PROSA optimizations further reduce baseline PROSA overhead by 2% and reduce power consumption by 4%, mainly by removing NACKs. Compared to baseline, enhanced PROSA outperforms results up to 50% for better network latency, 35% better runtime application and 7% power consumption reduction at the cost of using 7.6% more area. Overhead of PROSA is reasonable given the benefits in performance. As a future work, we plan to extend our proposal with partial circuits.

Chapter 8

A Low-Latency Reconfigurable Time-Division Multiplexed Network-on-chip for High-Assurance Systems

:

- **Authors:** Miguel Gorgues (Universitat Politècnica de València), José Flich (Universitat Politècnica de València) and Davide Bertozzi (Università di Ferrara)
- **Type:** Pending
- **Conference:** Pending
- **Location:** Pending
- **Year:** Pending
- **DOI:** Pending
- **URL:** Pending
- **Citation:** Pending

8.1 Abstract

High-assurance systems are typically organized as a set of domains that should be kept separate. The strong isolation requirement is motivated by the need to avoid security threats, to facilitate the verification and certification process, to effectively contain faults, and to enable composability and performance predictability. The on-chip interconnection network (NoC) is key to delivering strong domain isolation, since many of its internal resources are shared between packets from different domains. In this paper, we tackle the challenge of eliminating any form of interference in the NoC in the strict sense: injection of packets from one domain cannot affect the timing of packet delivery from other domains. This way, even timing channel protection is provided. The distinctive feature of our approach is that it is based on the observation of the channel dependency graph, thereby achieving the best isolation-performance trade-off with respect to state-of-the-art solutions. Another major step forward is that our NoC implementation is inherently reconfigurable, that is, a variable number of domains can be supported without the need to change the switch architecture, because of a token-based propagation mechanism of domain identifiers.

8.2 Introduction

Nowadays, Chip Multiprocessors (CMPs) have become pervasive across a computing continuum spanning from embedded systems to high-performance computing. However, when applied to mission-critical applications, CMPs need to be adapted in order to deliver security and reliability guarantees.

Indeed, in high-assurance systems it is a common practice to break the system into a set of domains, which are to be kept separate and should have no effect (i.e., interference) on one another. In CMPs, however, such controlled domain partitioning is not straightforward since the on-chip interconnection network (NoC) is a shared resource by all domains. Even assuming to spatially isolate domains inside physical compute and memory partitions, memory controller (MC) reachability becomes an issue, due to the need to cross intermediate partitions. As a result, the NoC resources are necessarily shared between communication flows from different domains, and the proper course of action should be taken to avoid domain interference.

Several degrees of non-interference can be enforced on the NoC. A first approach to loosening interdependencies among communication flows consists of delivering quality of service (QoS) guarantees. In fact, most QoS techniques aim at limiting flow rates, while restoring nominal rates in the absence of contention. While QoS-augmented NoCs can typically protect from denial-of-service (DoS) and bandwidth depletion attacks between domains, they cannot easily avoid an information leak associated with latency and throughput variations of communication flows as a function of network state. In fact, they can be used as timing channels by an attacker either to infer confidential information from a protected high-security program (side channel attacks) or to have a malicious program deliberately leak information covertly when direct communication channels are protected (covert channel attacks) [54].

When the protection against such timing channel attacks is required, even cycle-level variations of communication performance should be prevented, a scenario that we hereafter denote as *strong isolation* of domains. Interestingly, implementing strongly isolated domains makes it also easier to contain the propagation of faults, and does not require to account for all possible system-level interactions for the sake of certification.

In order to deliver strong isolation to networked domains, the NoC must be designed to guarantee the non-interference property in its strictest sense: injection of packets from one domain cannot affect the timing of packet delivery from other domains. One straightforward solution to this problem is to statically schedule domains on the network over time with some form of time-division multiplexing (TDM) [55]. However, this approach typically comes with relevant performance overheads. On the one hand, strict non-interference requires that resource allocation decisions are independent from application demands. On the other hand, performance of packets in time-multiplexed NoCs is highly sensitive to the scheduling methodology of time slots.

Also, while existing solutions support a generic global schedule, its reconfiguration at runtime is not at all obvious. In some cases, this would imply to solve a slot allocation problem in software or through hardware acceleration, and large programming tables at network interfaces [55] as well. In other cases, invasive hardware modifications would be required to optimally support different system configurations (e.g., number of domains) [61]. Therefore, these solutions are well suited for design time customizations only.

The main goal of our proposal is to deliver the strong isolation property for a runtime-configurable number of domains, while significantly reducing the latency incurred by temporal partitioning across the entire configuration space. Recently, there has been a surge of activity to tackle this very same challenge. SurfNoC [60] schedules the network into strictly non-interfering waves that flow across the interconnect. However, SurfNoC requires input speedup, which leads to significant area overhead. PhaseNoC [61] applies TDM scheduling at virtual channel (VC) level at the granularity of individual router pipeline stages. The phases are coordinated into optimally scheduled interlocked propagating waves, which ensure that in-flight packets of all domains experience the minimum possible latency. However, PhaseNoC is not able to deliver high-performance strong isolation across all possible system configurations in an efficient way.

The solution proposed by this paper provides a flexible architecture from the ground up, and extends performance and implementation efficiency to the whole configuration space. It builds on the concept of *Token-based TDM.* The approach is based on the observation of the Channel Dependency Graph (CDG) defined by the topology and the routing algorithm. From it, we derive the generic requirements to have all input ports of all routers serving packets from the same domain at each time slot. This would allow contention between packets from the same domain, while at the same time delivering secure-grade isolation between domains. The approach is generalized to an arbitrary number of domains by selectively and deterministically placing propagation stops at specific points in the NoC, which preserves the strong isolation property and delivers more latency-efficient NoC communications than state-of-the-art of solutions. Finally, we provide support for runtime modifications of the system configuration through a flexible architecture that does not require substantial changes to support the new configuration, but rather modifies the scheduling commands distributed to the network switches through a token-based notification mechanism.

With respect to state-of-the-art, our work pursues two innovative directions:

a) We understand the approaches to strong isolation and the performance pitfalls of previous solutions by means of a unified abstraction, which we identify with the CDG of the on-chip interconnection network. Then, we derive from its observation a latency-optimized and generic solution to the allocation problem of time slots to domains in the network.

b) We extend the level of flexibility of NoC architectures for strong isolation in high-assurance systems beyond design-time customizations. Our architecture runs unmodified with high performance regardless the static (switch and link latency) and dynamic (number of running domains) NoC settings.

The rest of the paper is organized as follows. In Section 8.3, we describe related work. In section 8.4, we provide a motivation of our work and introduce the target scenario. In Section 8.5, we describe the target architecture. Our proposal is then described in detail in section 8.6. In Section 8.7, we provide the experimental results, while conclusions are drawn in Section 8.8.

8.3 Related Work

Prior approaches to TDM-based scheduling in NoCs lose relevance when they are viewed from the viewpoint of the concurrently conflicting requirements of latency optimization, area efficiency and architectural flexibility. Numerous designs perform TDM scheduling at the time-slot level [55][56][57]. When using such architectures, the scheduling is typically performed offline (and assumes perfect a priori knowledge of the applications expected to be running on the system), and then statically applied to the entire NoC [58] [7]. However, in this type of approach the latency overhead can be quite substantial.

AEthereal [55] employs pipelined TDM (at the time-slot level) and circuit-switching to guarantee performance services. Traffic is separated into two main classes: 1) guaranteed service (GS) and 2) best effort (BE). Excess bandwidth not used by GS flows is given to BE flows. Packets on a single connection are always ordered, but ordering cannot be enforced between connections. This approach incurs a substantial programming overhead of time slots at network interfaces.

The SuperGT NoC [59] is an evolution of AEthereal providing three QoS classes. Aelite[56] simplifies the router architecture by providing only GS, and AElite moves one step further by including multicast traffic and fast virtual-circuit setup. Argo [57] allows schedules to evolve at the granularity of a single cycle, even when the routers have more than one pipeline stage. The resulting hardware cost is quite low, but the latency overhead can be substantial. In [54], static network partitioning in space and time is employed to provide multi-way isolation among the supported domains. This multi-way isolation property comes at a high performance cost, which is alleviated by the introduced reversed priority with static limits (RPSL) mechanism. It uses priority-based arbitration and static limits to guarantee one-way isolation between high-security and low-security flows.

A recently introduced architecture, called SurfNoC [60], employs optimized TDM scheduling, also applied at the VC level, to minimize the latency overhead. However, the required hardware is expensive. Achieving low-cost implementations with SurfNoC would increase the latency overhead of static scheduling. The current state-of-the-art in TDM-based scheduling is PhaseNoC [61]. It improves the VC-level scheduling proposed by SurfNoC by pre-configuring the network in order to receive packets from the same domain at all the input ports at every cycle, and performing the arbitration in the next cycle of this incoming domain. Following this approach, PhaseNoC meets the first requirement, by minimizing the latency overhead. However, it lacks of flexibility, because PhaseNoC needs to modify the network (adding stages at the router) to support a higher number of domains. PhaseNoC proposes to divide the network into $x+y+$ and $x-y-$ to support a higher number of domains. However, following this approach, it can not guarantee the non-interference property any more, for which it would need input speedup similarly to SurfNoC.

8.4 The Strong Isolation Requirement in High-Assurance Systems

Current CMPs are designed following a tile-based approach, where a compute tile, typically including distributed L2 memory, is replicated inside the chip and connected by a network-on-chip (NoC) at the top-level of the hierarchy [81, 82]. In high-assurance systems, support for resource partitioning into isolated domains is an emerging requirement. For instance, Integrated Modular Avionics (IMA) architectures are currently mainstream avionic systems, constituting a logically-centralized and shared computing platform hosting a variety of avionics functions on a single computing platform (multi-function integration). IMA requires the use of software partitioning [83], which consists of achieving fault containment in software, independently of the underlying hardware platform. However, with the advent of parallel computing architectures, the partitioning concept should be enforced in hardware as well. The avionics safety standards dictate the enforcement of partitioning in space and time. Spatial partitioning ensures that an application in one partition is unable to change private data or use private devices of another one. Temporal partitioning guarantees that the timing characteristics of an application, such as worst-case execution time, are not affected by the execution of an application in another partition. While these concepts have been fundamentally driven by fault-tolerance and performance predictability considerations, protection from cyber attacks is getting increasing attention in the design of avionic systems [84]. Today, such systems should be safeguarded not only against basic DoS attacks, but also from other vulnerabilities such as timing channels [85]. Similar considerations hold for the automotive domain, where densely populated networks of electronic control units are being replaced by distributed central compute platforms, driven by cost, weight, complexity and security considerations. The picture becomes even more critical when considering the possible co-integration of safety-critical functions (e.g., the breaking system) with infotainment and even third-party applications. Also in this domain, numerous works are surveying cars' intercommunication technologies and possible threats [86, 87], and demonstrating different kinds of attacks [88].

Last but not least, computing platforms for space applications are following the same trend, extending the concern from failure cascading avoidance to the interference threats between system components in multicore architectures [89, 90].

Overall, the key challenge of applying time and space partitioning to CMP platforms lies in the on-chip interconnection network, where an additional layer of possible interaction arises for the system as a whole. Routers and links are shared among domains, and are thus subject to contention and congestion. This renders network performance unpredictable, prevents fault containment and exposes unprecedented security threats.

CMPs must isolate flows between different domains, preventing a packet being blocked by packets from other domains. This problem can be partially solved by using a different Virtual Channel (VC) for each required domain. This way, an attacker can only fill the VC buffers assigned to its domain.

However, separating the flows in VCs is not enough to guarantee the strictest notion of non-interference, that is, the latency of packets belonging to a domain should not depend on the traffic from others domains. This means that a packet cannot get delayed

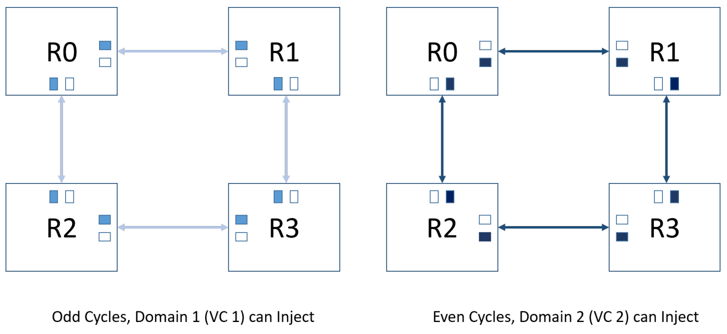

R0
R1
R2
R3
Odd Cycles, Domain 1 (VC 1) can Inject
Even Cycles, Domain 2 (VC 2) can Inject

FIGURE 8.1: Reference solution: VC partitioning coupled with time-division multiplexing.

because of conflicts with packets from other domains, therefore no timing channel is exposed in the NoC. This is the explicit target of this work.

8.4.1 Time Division Multiplexing

The reference solution to avoid any kind of domain interference consists of partitioning the virtual channels **and** time-multiplexing the physical channels and crossbars between different domains such that channels are only allowed to propagate packets from different domains on different cycles. This TDM partitioning scheme ensures that latency and throughput of each domain are completely independent of the other domain's load. However, this baseline scheme is heavily sub-optimal and non-scalable, since packets will have to wait as many cycles as the number of concurrent domains minus one at each hop. The incurred penalty grows significantly with the physical distance of the receiver end node.

To cope with this problem, we share the same intuition with previous work (SurfNoC, PhaseNoC), that is, we keep these time-varying partitions while changing the phase of their oscillations to cut down on network latency. However, our approach is based on the observation of the CDG, and is compared with state-of-the-art in the experimental results.

8.5 Target Architecture

Figure 8.2 shows the target CMP architecture, which is based on a tile-based approach. Each tile is composed by a core, a private L1 and a distributed L2 cache bank, in addition to a NoC router. All the elements are directly connected to the router. Routers from different tiles are connected by using a 2D mesh topology. Cores access main memory by sending a request to the memory controllers (MC). Without lack of generality, we assume one MC is attached to each network corner, hence 4 MCs are shared between

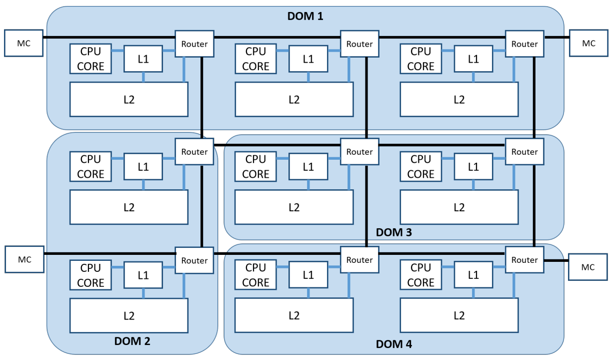


FIGURE 8.2: Target CMP architecture with a partitioning pattern into isolated domains.

all compute and memory tiles. Our work is complementary to previous work addressing isolation of off-chip memory accesses, resulting in augmented MC architectures [54].

The proposed architecture implements single-cycle routers, where link traversal and input buffer storage are performed in one cycle, whereas VC allocation (VA), switch allocation (SA) and crossbar switching (X) are performed in another cycle. Our design philosophy for strong isolation is not limited to this architecture, and can be extended to any NoC.

The target architecture is partitioned into disjoint domains, mapped to a subset of contiguous tiles. Figure 8.2 shows an example of a partitioning pattern as well. As a result, two kinds of communication requirements arise: between tiles of the same partition/domain, and between such tiles and the memory controllers. We target cost-effective implementations, therefore we do NOT replicate the NoC to serve both kinds of communications, which are thus consolidated onto the same NoC.

The NoC has a VC (or a set of VCs) reserved for packets of each domain. Packet routing is performed via logic-based distributed routing (LBDR), which avoids the access delay and area overhead of look-up tables [91]. LBDR consists of combinational logic at each input port, which processes the coordinates of the packet destination and determines the target output port at each switch. The logic accounts for forbidden turns by the routing algorithm and for topology boundaries, which are coded in a configuration register programmed with the so-called *LBDR bits*. We assume a unique, global minimal-path routing algorithm programmed through these bits, and we do not restrict the shapes of the partitions that can be inferred on top of the NoC. The routing algorithm is deadlock-free, hence the CDG is acyclic.

8.5.1 Partition-Aware TDM

Previous work has never considered the spatial locality of partitions, or has only considered single-tile partitions. The awareness of the partitioning pattern suggests a straightforward extension of TDM, which we also consider as a baseline reference solution.

If we assume that intra-partition traffic will never collide, then we can think of enabling such intra-partition traffic in parallel. In this article, we consider a TDM schedule where all domains can transmit local traffic during the same time slot, while a distinct time slot is available for each domain to send or receive packets to/from memory controllers.

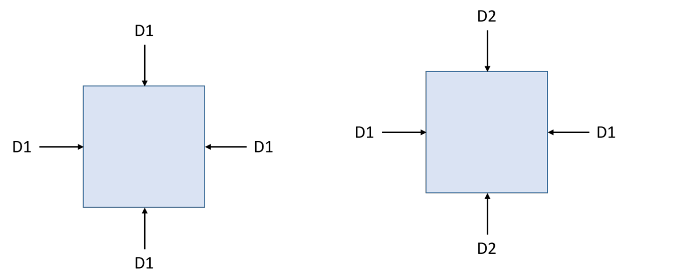
In practice, while strong isolation of global traffic (to/from MCs) is delivered by the TDM mechanism, local traffic can be easily kept non-interfering by exploiting the spatial locality of these communications. There are two possible solutions. First, we may restrict to regular partition shapes such as squares or rectangles. No two packets from neighboring partitions would collide on the same route, since there would be no intersection between such routes with minimal path routing. Second, we may allow arbitrary partition shape while limiting packet routes within the boundaries of the partition the sender-receiver pair belongs to. This can be achieved by enforcing the connectivity bits (a subset of the LBDR bits denoting topology connectivity) to zero, so that the LBDR logic never routes a packet outside the partition. This would however give rise to a concern: with a given routing algorithm, some partition shapes would not be fully connected, hence should be avoided. Without lack of generality, we validate several TDM approaches on regular partition shapes, so that partition-aware TDM can be part of the evaluation framework.

8.6 CDG-driven Strong Isolation

8.6.1 Router-Level Strong Isolation

Token-Based TDM enforces non-interference between traffic logically belonging to different partitions. The latter includes flows between sender-receiver pairs within the same partition, and flows from/to tiles of a partition to/from any MC. Even cycle-level variations are prevented in order to avoid timing channel attacks.

To achieve such property, the network relies on a token propagation scheme. Tokens contain scheduling orders to local router-level domain schedulers. For this purpose, tokens carry a domain identifier (DI), which identifies the domain whose packets can be forwarded from a specific router input upon arrival of the token. *In order to deliver strong isolation between domains, we need to synchronize the timing of token propagation throughout the network in such a way that every router gets homogeneous DIs at each*



(A) Requirement for router-level strong isolation. (B) Scenario where domain interference may cause latency variations.

FIGURE 8.3: Scenarios for router-level slot allocation to domains D_i.

input port on every cycle (see Fig.8.3). This means that the router will arbitrate and forward messages belonging to the same domain/partition, and messages from different domains will never compete.

The idea of this paper is to create and propagate as many kinds of tokens as the number of running domains, each one with its own Domain Identifier (DI). All different kinds of tokens are triggered back-to-back in sequence from a source node in the network, and propagated through it in the order of the CDG. The proposal triggers one new kind of token at each cycle, and repeats the domain sequence every D cycles, where D is the number of domains. As tokens are received in sequence at switch input ports, they command the local scheduler to serve packets with a specific identifier from those ports.

8.6.2 Synchronized Token Propagation

Since tokens traverse router ports and links in the order of the CDG, synchronized same-ID token arrival at all router input ports depends on the CDG and nominal router and link latencies.

Assuming an horizontal Segment-based Routing algorithm [73] and single-cycle routers and links, tokens would be triggered from the bottom right corner, and would be propagated throughout the network as illustrated in Figure 8.4. With this routing algorithm, there are two token propagation phases, a scroll-up one (left) and a scroll-down one (right), which occur one after the other. Their combined effect is the traversal of all router ports and NoC links. Numbers in the figure indicate token propagation latencies since initial injection. The diagonal arrows represent routing restrictions, that is, directions that packets can NOT take as dictated by the routing algorithm at hand. They are set in order to prevent cyclic dependencies, that is, deadlock from occurring.

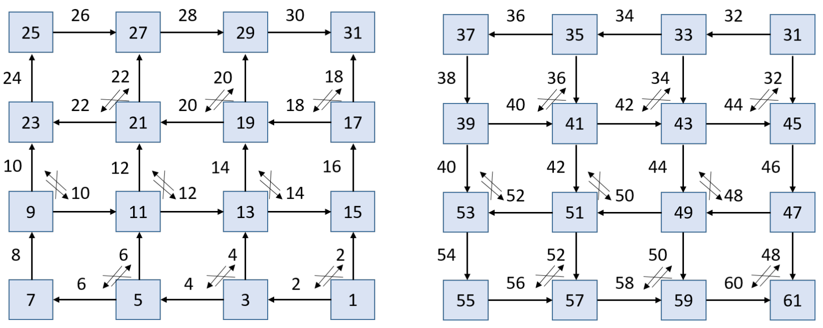


FIGURE 8.4: Network-level token propagation, with latency annotated, in the order of the CDG with horizontal segment-based routing and single-cycle routers and links.

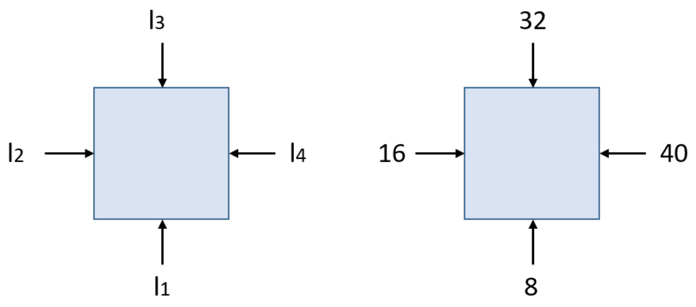


FIGURE 8.5: Arrival times (in clock cycles) of same-ID tokens at router input ports.

If we focus on a single router and a single token crossing the network as specified in the CDG, the token will reach different input ports of the same router at different timestamps t1-t2-t3 and t4, as seen in Figure 8.5.

Let us define *relative latencies* as the time periods elapsed between any two consecutive timestamps. *Router-level operation with strong domain isolation requires that the result of the modulo operation between any of these relative latencies and the number of domains is always 0*:

$\forall (y, x) \in A \rightarrow (ATy - ATx)\ mod\ D = 0,\ where\ ATy > ATx$

where A is the set of router input ports, D is the number of running domains, and ATi is the arrival time of same-ID tokens at input port i. In fact, as the proposed architecture injects the same domain identifier token into the network every D cycles, this ensures that a token with the same identifier will reach a specified port every D cycles.

Therefore, if the previous condition is met, at regime all the router input ports will receive tokens with the same domain identifier exactly at the same time, and can thus work in strong isolation mode. The number of domains D that enables this operating condition is the Maximum Common Divisor (MCD) of all relative latencies between consecutive pairs of arrival times (in increasing order).

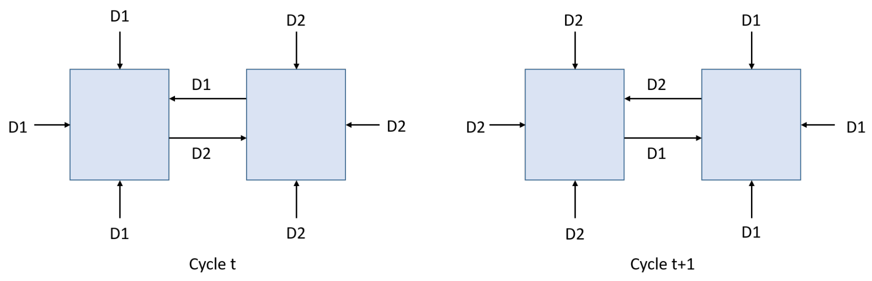


FIGURE 8.6: Perfect scheduling at network level for minimum latency.

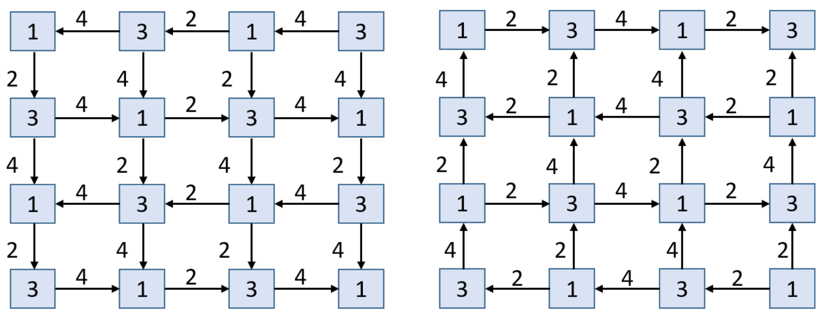


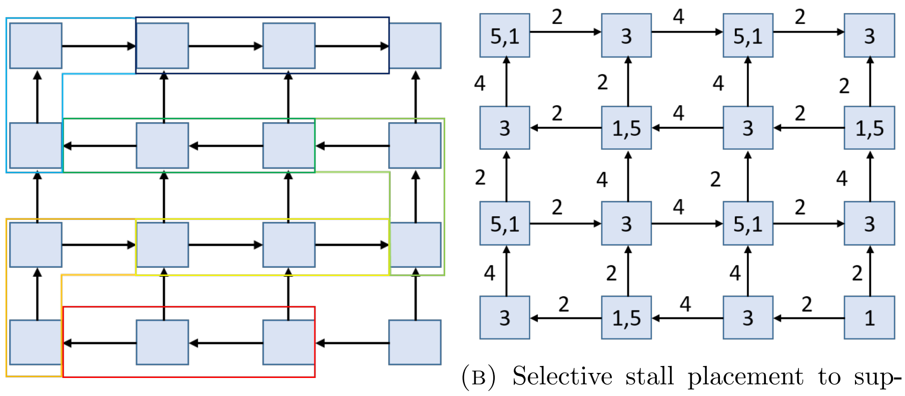
FIGURE 8.7: Token propagation flow at regime in a specific time slot. Numbers denote the token ID served on a specific NoC resource at that clock cycle.

Following the example in Figure 8.5, relative latencies from initial token arrival to its presence on all input ports amount to 4, 32, and 8 cycles. Hence, the router delivers strong isolation with 4 domains..

To extend this property to the whole network, we first calculate the maximum number of domains for every router as the MCD illustrated above (if any). If such an MCD can be computed for each router, then the topology, coupled with the target routing algorithm, supports strong isolation of domains. In this case, *the MCD of all router-level computed domains is the ideal number of domains which delivers strong isolation for the network as a whole.* In the 2D mesh example in Fig.8.4, strong isolation is achieved with 4 domains.

Notice that the ideal number of domains for strong isolation can be directly determined by the smallest relative latency inside the network. In our case, the smallest relative latency of the NoC in Fig.8.4 is exactly the one reported in Fig.8.5 (which is 4 cycles), and corresponds to the latency of the smallest cyclic path spanned by a token in the network to reach two different ports of the same router. Clearly, this cyclic latency depends on router and link latency, and is equal to: $(R-2)*(P+L)$ where R is the number of switches in the smallest cycle, P is the router latency and L is the link latency.

A relevant side effect of this theory is that the composition of strongly-isolated router-level operations at network level through the dependencies of the CDG gives rise to a



(A) Building segments out of the shortest token cycle.

(B) Selective stall placement to support 5 domains with strong isolation. Only scroll-up phase shown.

FIGURE 8.8: Extending the number of domains under strong isolation.

Perfect Schedule, which consists of the onset of unstopped propagating waves of same-ID tokens throughout the network (see Fig.8.6). This guarantees minimum-latency operation of the NoC.

Figure 8.7 shows token propagation flow over the network at regime, which is established once the first token comes back to the injecting router. Each router works in strong isolation mode, and globally a perfect schedule takes place.

This methodology generalizes the constraints for perfect scheduling identified in [61] from local scheduling loops, which we instead base on the observation of the CDG. This generalization is at the core of the new approach proposed in this paper, featuring enhanced flexibility as hereafter illustrated.

8.6.3 Supporting a Higher Number of Domains

In order to handle a larger number of domains than the ideal one, PhaseNoC requires deep modifications of the router architecture, thus proving unsuitable for runtime reconfigurability. In particular, the pipeline depth of all routers needs to be increased, which would preserve strong isolation and perfect scheduling. As we will see in the experimental results, for some configurations this leads to suboptimal performance. Alternatively, the NoC can be split into communicating subnetworks, similar to SurfNoC. In this case, the strong isolation property is lost (i.e., domains can affect timing of packet propagation in other domains), unless area-expensive input speedup is implemented to avoid VC contention at crossbar inputs.

Our CDG-inspired approach is less invasive and flexible, and aims at preserving strong isolation in router-level operation while giving up the perfect schedule globally.

As we have seen previously, in a regular 2D mesh network all relative latencies are multiples of the latency of the shortest cycle SCL (i.e., of $SCL = (R - 2) * (P + L)$). Therefore, we can split the critical path of the tokens throughout the CDG into segments of length SCL cycles (see Figure 8.8a). With the ideal number SCL of domains, all inputs to these segments will be in the same domain at a given time slot.

In order to support a higher number of domains D, our intuition is that only SCL domains should be in flight at any given point in time. The remaining $D - SCL$ domains should be stalled. This would enable to preserve the strong isolation property, while breaking the perfect scheduling assumption. As we will show in the experimental results, preserving perfect scheduling at all costs may not be the best performing solution.

In order to implement this concept, we need to place domain propagation stalls selectively within segments. The constraint to be met for correct operation is to place these stalls at the same positions within segments (e.g., either in the first router, or in the second one). Figure 8.8b shows an example of stall placement to support 5 domains. As can be observed, the stalls allow the network to receive the same domain identifier at all the input ports at every router.

Depending the target number of domains D and the ideal number of domains SCL, the number of stalls in each segment can be calculated as follows:

```
**if** D ¿ SCL **then**
    $Stalls = D - SCL$;
**else**
    $Stall = 0$;
**end if**
```

8.6.4 Supporting a Lower Number of Domains

If at any given point in time, a lower number of domains than the ideal one needs to be enforced, then *strong isolation and perfect scheduling can be preserved provided the target number of domains is an integer divider of the ideal number.* The reason is because this way the latency of the shortest cycle spanned by tokens to bridge two consecutive ports of the same router is a multiple of the repetition period of domain identifiers. Therefore, previous conclusions are still valid. As an example, with ideally 12 domains in a perfect schedule with strong isolation guarantees, the same property can be delivered with 2, 3, 4 and 6 domains.

With a different number of domains (e.g., 5 or 7), the same stall-based methodology explained before can be applied here, which preserves strong isolation but not perfect scheduling. In particular, the procedure that follows should be used to re-align domain identifiers token at segment inputs with $d < SCL$ domains:

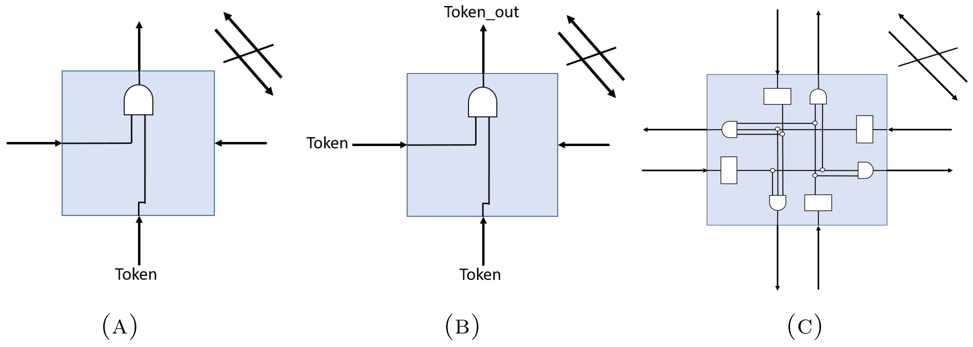


Figure 8.9: Token propagation mechanism.

1- compute an integer n such that $n \times d > SCL$;
2- compute the number of stalls per segment as $stalls = n \times d - SCL$;
3- enforce this number of stalls at the same position within each segment.

8.6.5 Handling the Transient to Regime

Before getting to regime, tokens start propagating in the network by following the CDG. However, not all input ports have received a token at a given point in time.

Figure 8.9 shows an example of initial token propagation through a router. For the sake of simplification, the figure only shows output port north, and input ports east, west and south. Arrows denote routing restrictions between the east and the north ports. Then, based on this CDG, only packets from the south and west input ports can be forwarded to the north output port. Figure 8.9a shows that a token has been received from south. However, this token cannot be forwarded downstream, since not all the input ports with routing dependencies with the north output port have received such a token. Therefore, the south token is dropped. When the token reaches the west input port as well (Figure 8.9b), it will appear again also in the south input port (see Section 8.6.2), then a new token can be forwarded to the north output port.

8.6.6 Router Architecture

Figure 8.10 shows a simplified version of the Token-based TDM router architecture. Only one input port is shown in order facilitate understanding, assuming one VC per domain. Single-cycle routers and links are considered.

Tokens are in charge to set the domain processed within the router. The incoming token ($TOKEN\_IN$) is used as a selector in the first demultiplexer stage to define in which domain buffer the incoming data flit will be stored. The $TOKEN\_IN$ is stored in a

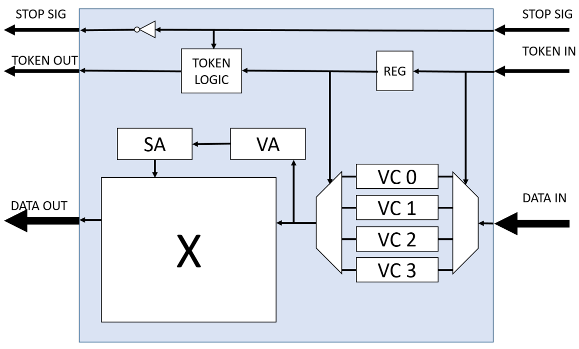


FIGURE 8.10: Proposed router architecture: the concept.

register for the sake of retiming, so that in the next clock cycle it can select the active domain within the router. Following this approach, only one domain can access to the VA/SA/X per cycle, then no conflicts between different domains can occur.

The token advances until the token logic, which is active only during the initial configuration transient, till the regime is reached. Until then, the logic checks whether tokens are available at all input ports with routing dependencies with the target output port. If not, tokens are dropped. At regime, all such tokens will be available, and an output token ($TOKEN\_OUT$) will be fired.

This initialization procedure is slightly more complex in case the network is initialized with a higher number of domains than the ideal one for perfect scheduling. In this case, the first domain identifier token carries the required number of domains. At the same time, the STOP signal is set to one. The token logic then computes the number of stalls to be enforced. Negation of the STOP signal enables to enforce stalls selectively once every two routers. After this, domain identifiers are injected in sequence at every cycle as usual. As an incoming domain identifier token arrives at the input port, it has to wait in a token buffer (of size $Dmax - SCL$) until it reaches the first position in the queue.

The proposed architecture can be inherently reprogrammed to support any number of running domains, since the adaptation is not in the architecture itself, but in the scheduling commands sent through the tokens. In the most complex case, the token logic has to compute, for the initialization procedure, the number of stops for tokens in transit. While in this paper we are experimentally characterizing the offered architectural flexibility, the precise reconfiguration protocol to exploit it at runtime (i.e., to safely transition from one network configuration to another) is left for future work.

Finally, in a traditional router with M-ports, D domains and v VCs per domain, there exist M x D x v input and outputs VCs. Then, virtual channel allocation maps M x D x v inputs to M x D x v outputs VCs. However, our proposal only allows one domain to access the VA arbiters at a time, then only M x v VCs perform VA at every cycle.

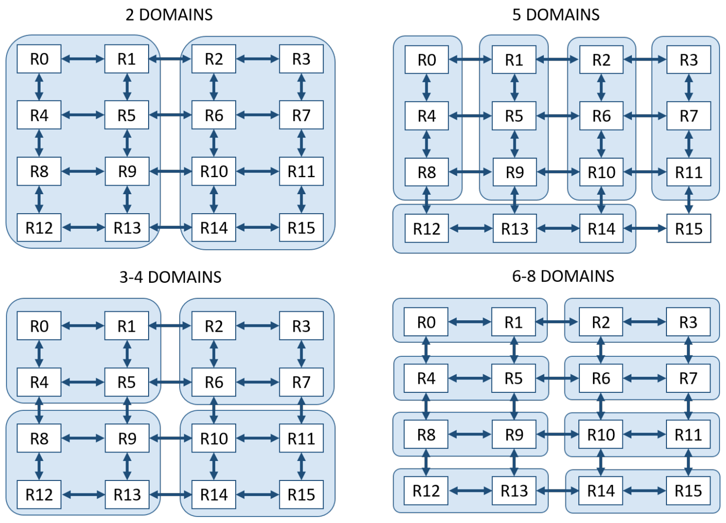


FIGURE 8.11: Different number of domains considered on a 16-tile 4x4 network.

Similarly, the M x D x v to M x D x v switch allocator in traditional routers is reduced in our implementation to a M x v to M x v allocator.

8.7 Experimental Results

For the experiments, we use the VirtualSocLite virtual platform [74], targeting the full-system simulation of massively parallel heterogeneous SoCs. It is coded in SystemC and models operation of a 2D mesh topology of any size with RTL-equivalent accuracy. The platform has been extended to model the proposed NoC architecture, and synthetic traffic generators have been instantiated and linked to the NoC.

For all the results, we inject a mix of read/write transactions from the tiles to the distributed L2 banks of a partition (intra-domain traffic). Inter-domain traffic is obtained by injecting additional a mix of read/write transactions from domain tiles to memory controller nodes (MC traffic). Two topologies sizes are simulated: 4×4 and 8×8 2D-mesh topologies. In both cases, there is one core per tile.

Different configurations of domains are used. In all cases, domains are of the same size and geometry. Figure 8.11 shows the domains for the 16-tile network.

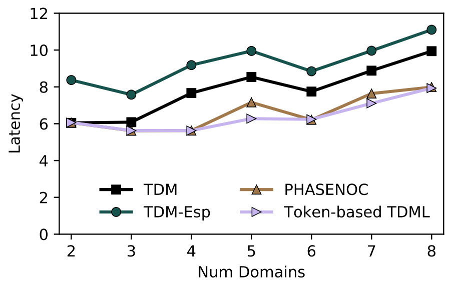
In the 8x8 2D mesh, in case of 2 domains, the network is divided into 2 domains of 32 tiles each. For 3 and 4 domains, the domain size is 16 tiles. For 5 domains, the network is split into domains of 12 tiles; if the networks requires between 6 and 8 domains, these domains are composed by 8 tiles. In case more than 8 domains are required, the domain size is set to 4 tiles.

We evaluate four mechanisms across a different number of running domains, so test the flexibility claim:

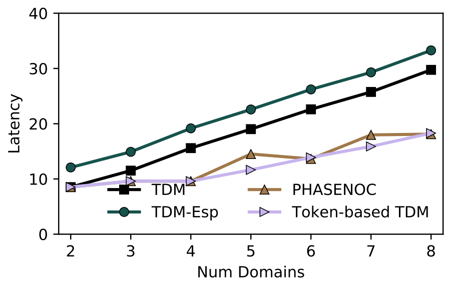
- TDM. A baseline TDM network where at each time slot the whole network is used for a specific domain. All domains are served in consecutive order. At each time slot, both intra- and inter-domain traffic for the associated partition is served.
- Partition-aware TDM, described in Section 8.5.1, where the TDM scheme is made aware of spatial partitioning. In practice, a dedicated time slot is concurrently used for intra-partition traffic of all domains. Partition-specific local traffic is kept segregated by the regularity of partition shapes. Then, every partition has one additional reserved slot to transmit or receive its own MC traffic. This scheme will be briefly referred to as *TDM-esp* in the results.
- PhaseNoC, which delivers strong isolation and minimum communication latency under specific configuration options (no. of domains). For each tested configuration, we tune the PhaseNoC router with the proper number of pipeline stages for perfect schedule and strong isolation (although this would be problematic to apply at runtime). However, in some cases PhaseNoC ends up in a suboptimal configuration. For instance, PhaseNoC with two pipeline stages per router can support 6 fully-isolated domains in perfect schedule, but if the required number of domains is 5, one domain would go unused. In contrast, if we revert back to 1 stage per router, strong isolation cannot be provided with 5 domains. For a fair comparison, we allocate such an unused time slot to the active domains in a round robin fashion. With PhaseNoC, both intra-domain traffic and MC traffic are served when a domain is active. PhaseNoC would also enable the partitioning of the network into subdomains to handle the critical cases. However, we verified that in this case the strong isolation property is lost, unless input speedup is implemented. We do not consider this case here, since it would amplify the implementation complexity gap between PhaseNoC and our approach.
- Token-based TDM. This is our approach, which serves both inter- and intra-domain traffic when a domain is active.

8.7.1 Zero-Load Latency

Figure 8.12a shows the zero-load latency results for local intra-domain traffic. The x-axis represents the number of domains and the y-axis the zero-load latency. As can be observed, both PhaseNoC and Token-based TDM improve upon the baseline TDM variants. However, our proposal improves upon PhaseNoC in scenarios where PhaseNoC's pipeline has to be oversized for strong isolation (5 and 7 domains). The improvement on the network latency is 13% and 9% for 5 and 7 domains, respectively. In these scenarios, we claim "generalized perfect scheduling" through selective placement of stalls. In scenarios where PhaseNoC achieves perfect scheduling by increasing the switch pipeline depth (with 4, 6 and 8 domains), our proposal provides the same perfect scheduling

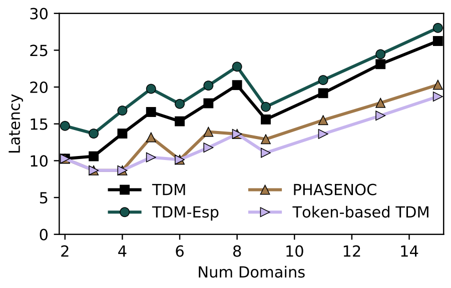


(A) Local traffic.

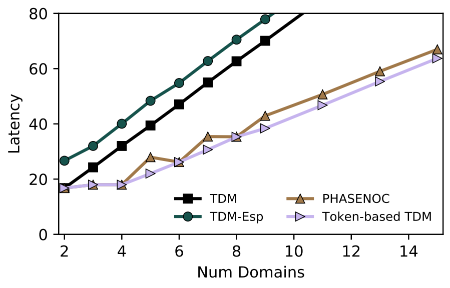


(B) MC traffic.

Figure 8.12: Zero-load traffic for the 4x4 2D-mesh.



(A) Local traffic.



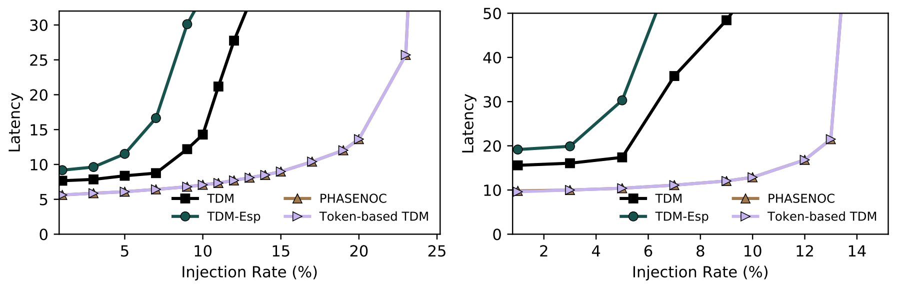
(B) MC traffic.

Figure 8.13: Zero-load traffic for the 8x8 2D-mesh.

as PhaseNoC. Thus, our approach provides increased flexibility without wasting performance. In order to properly read the plot, it should be noted that the higher the number of domains the smaller the partition size (Figure 8.11). However, latency tends to increase because there is a higher waiting time to pay in the network interfaces to wait for the suitable injection time slot.

Figure 8.12b plots the zero-load latency results for MC traffic. The performance reached with this type of traffic is similar to the local traffic one. Again, we appreciate the capability of Token-based TDM to keep up with PhaseNoC whenever the latter delivers perfect scheduling, while improving the network latency to access the MCs by about 20% for 5 domains and by 12% for a 7-domain configuration. Thus, our approach proves capable of generalizing the high performance efficiency of PhaseNoC to the whole configuration space, while exposing inherent reprogramm ability of the number of domains without pipeline modifications. We instead just change the scheduling commands we give to the switches. Another architectural difference explains the above results: while our approach tends to introduce a configurable number of stops for domain identifier tokens in specific points of the NoC in order to preserve the non-interference property, PhaseNoC tends to spread the latency overhead everywhere in order to achieve the same goal.

Figure 8.13a shows the zero-load latency results for local intra-domain traffic in a 64-tile scenario. As in the 16-tile network, Token-based TDM improves upon PhaseNoC in scenarios where the pipeline is oversized. Even the reallocation of the unused slot



(A) Local traffic. (B) MC traffic.

FIGURE 8.14: Uniform traffic for 4 Domains

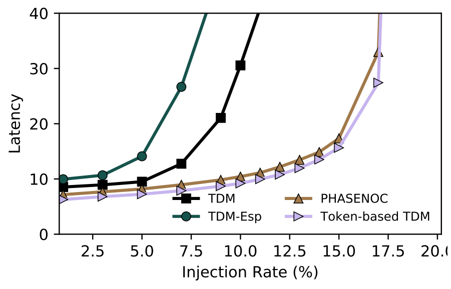
is not able to compensate for this inefficiency. The improvement of Token-based TDM oscillates between 20% (5 domains) and 9% (15 domains). For MC traffic (Figure 8.13b), Token-Based TDM reaches the best performance improving up to 30% the PhaseNoC network latency. For a number of domains higher than 8, the cases where PhaseNoC gets perfect scheduling are not shown because in those cases Token-based TDM performs the same, as we have shown previously. Notice that, the higher the number of domains the smaller the benefit achieved by Token-based TDM. This is due to the fact that as the number of domains gets larger, the bandwidth underutilization by PhaseNoC decreases.

8.7.2 Network Performance

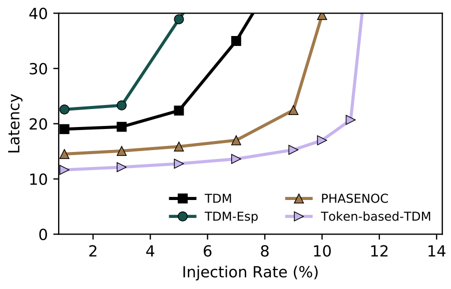
Figure 8.14a shows the network saturation curve for local traffic under uniform random traffic. The characterized scenario consists of four domains. If our approach is correct, we expect both PhaseNoC and our approach to deliver perfect scheduling. As shown in the Figure, the Token-based TDM exactly matches the performance of PhaseNoC, thus validating the claim. Figure 8.14b shows the performance for MC traffic with 4 MCs, in the same characterized scenario. These plots further validate the same conclusion, hence validating the efficiency of our approach in *matching the best case for PhaseNoC, while delivering extended flexibility.*

Next we analyze the network saturation curves in those cases where our proposal can set a "generalized" perfect scheduling while PhaseNoC can not. The unused slot domain is assigned in round robin fashion to the active domains. For this study we analyze four scenarios: one where the whole traffic is local, another with only MC traffic, and two scenarios with mixed traffic, one with fifty percent of each type of traffic and another one with 75% of local traffic and 25% of MC traffic.

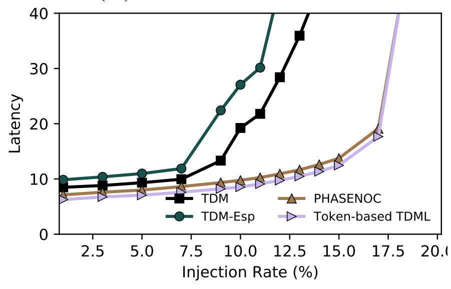
Figure 8.15a shows the results for local traffic for the 5-domain scenario. As can be seen, PhaseNoC and Token-based TDM improve the network performance of TDM. In addition, Token-based TDM reduces the network latency against PhaseNoC by 10% along the complete network saturation curve, reaching the saturation point at similar injection



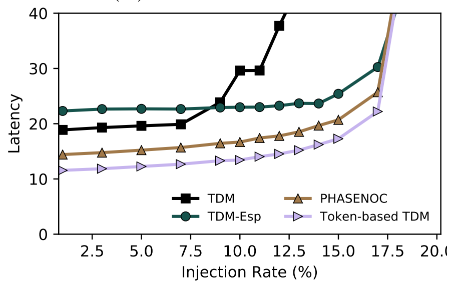
(A) 100 % Local traffic



(B) 100 %MC traffic



(C) 75 % Local traffic



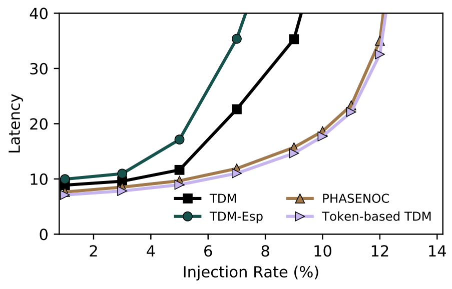
(D) 25 % MC traffic

FIGURE 8.15: Uniform traffic for 5 Domains.

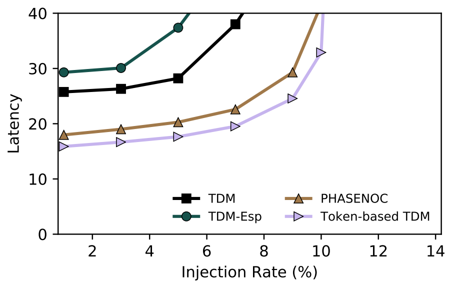
rates. Figure 8.15a plots the results for MC traffic. Similar as the previous case, Token-based TDM outperforms PhaseNoC by up to 20% before reaching the saturation point. Moreover, our approach increases the network capacity. When a mixed-flows scenario is presented, Token-based TDM continues to reach the best performance. Figures 8.15c and 8.15d show the results for local and MC traffic for a scenario with 75% of local traffic and 25% of MC traffic, respectively. Similarly to previous cases, Token-based TDM is able to reduce the PhaseNoC latency by roughly 10% and 20% for local and MC traffic, respectively. Moreover, in this scenario the customized TDM scheme (TDM-esp) has higher network latency compared with baseline TDM. However, for MC traffic TDM-esp is able to match the accepted traffic before reaching the saturation point. This behavior occurs because for a given domain with mixed flows, TDM-esp provides higher bandwidth than baseline TDM, because each domain has two time slots, one for intra-domain traffic and one for inter-domain traffic (for access to the MC).

Focusing on a 7-domain scenario, Figure 8.16 plots the latency results. Similarly to the 5-domain scenario, Token-based TDM reduces network latency by 8% in intra-partition traffic scenario and 15% in MC traffic scenario. Figures 8.16c and 8.16d plot the results for the mixed-flows scenario, where the traffic is divided in equal parts between both types of traffic. Token-based TDM gets the minimum network latency, improving over PhaseNoC by 7% on local traffic and 12% on MC traffic. However, similarly to what happens in the mixed-flows scenario for 5 domains, TDM-esp is able to accept a higher injection rate of traffic.

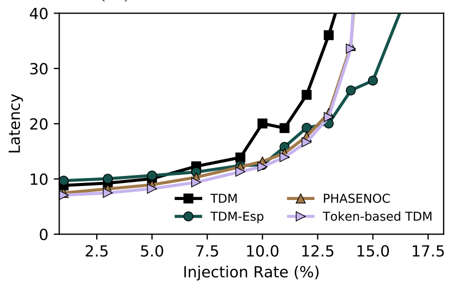
Due to lack of space, the results of the network saturation curves for 64 nodes are not shown. However, the performed analysis shows that the results obtained for the network



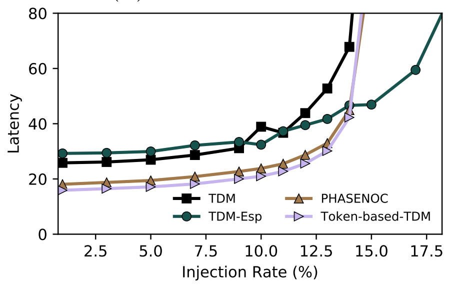
(A) 100 % Local traffic



(B) 100 %MC traffic



(C) 50 % Local traffic



(D) 25 % MC traffic

FIGURE 8.16: Uniform traffic for 7 Domains.

of 16 nodes can be directly extrapolated to the network of 64 nodes.

8.8 Conclusions

In this paper we propose Token-based TDM, a novel approach to sustain domain isolation in high-assurance systems running on top of CMP computing platforms. Special emphasis has been devoted to providing protection against subtle security threats such as timing channels. This requires the enforcement of strong isolation in its strictest sense, that is, injection of packets in one domain cannot affect the timing of packet delivery of other domains.

Our approach leverages the unique understanding of the problem enabled by the channel dependency graph abstraction of an on-chip interconnection network. This has enabled to understand previous work in the context of the new unified design framework, to generalize its validity and to bring performance and implementation efficiency to the next step.

Overall, Token-based TDM is proven to match the performance best cases of PhaseNoC (the most competitive approach presented so far in literature), while generalizing the performance efficiency to the remaining configurations. Performance speedups range from 10% for intra-domain traffic up to 30% for MC traffic.

At the same time, our approach enables a flexible NoC architecture, that potentially goes beyond the design-time tunability of PhaseNoC. In fact, our philosophy is not to update the switch architecture to the execution scenario, but rather to change the scheduling commands sent to the switches via a token-based notification mechanism. This provides an unprecedented level of flexibility at an overly small implementation complexity.

Token-based TDM turns out to be a more latency-efficient solution than any incremental attempt to customize the baseline TDM scheme for spatial partitioning. Such custom-tailored TDM is only able to extend saturation bandwidth with balanced local and MC traffic. It should be seen only as a first step toward full (and more expensive) NoC replication, where a TDM NoC is used for MC traffic, and local traffic can be propagated concurrently within partitions at each time slot through a dedicated local NoC with segregation capability of routing paths.

In future work, we will address the runtime reconfiguration protocol enabling to take advantage of the provided architectural flexibility.

Chapter 9

Conclusions

9.1 Thesis Conclusions

Networks-on-chip are emerging as the key solution in the multicore/manycore era to provide connectivity between tens, hundreds or even thousands of nodes, due to its effectivity and scalability. However, as higher the number of cores is, the connectivity gets higher importance inside the chip. The use of parallel charges increases communications within the chip. Therefore, the effectiveness of the transmission of the packets along the NoC.

In this thesis we have proposed some techniques to improve the network latency, making communication more effective in the NoC. All the contributions have been developed and implemented on a cycle-accurate simulator. TBFC+SUR, ECP, and PROSA are developed and implemented on GNoCsim simulator. Memory access patterns from real applications are obtained from SPLASH and PARSEC. These patterns are used as an input to provide GNoCsim with real traffic applications. The memory coherence protocol has been modified to support PROSA improvements adding some transients states. Moreover, these contributions are also implemented on Verilog to study the real implementation overhead. The last contribution of this thesis, Token-based TDM, has been developed on an international internship in Ferrara. This contribution is implemented on VirtualNoCsim. Building on this work, there we show some conclusions extracted from the current work.

The first proposal, termed TBFC and SUR, contributes to the field by designing a new flow control mechanism and complementary routing algorithm. The aim is to attain a perfect balance of buffer resources usage within the NoC, thus achieving optimal performance. The combination of TBFC and SUR allows us to reduce the number of VCs required to implement fully adaptive routing algorithms. The improvement in the VC utilization allows our proposal to improve previous proposals, allowing a higher injection rate in the network before to reach the saturation point. The results show a similar performance when our proposal is implemented with the minimum resources (VCs) to work. However, when the the number or resources is matched, our proposal outperforms the Fully Adaptive routing algorithm up to 20 % on network throughput.

The second proposal of this thesis is the EPC filter. It can help to manage and prevent spreading congestion within the network when adaptive routing is used. This proposal has demonstrated that by filtering and limiting the available output ports to the packets with the same destination than another has recently sent, the network latency for non-congested destinations is not affected by the congestion. Then, a switch with EPC improves the non-congested traffic reducing network latency at least four times and outperforming FA at least by 28% on throughput under a congested scenario.

As is well known, in a CMP scenario packet switching is commonly implemented, this switching architecture reaches a good utilization and network performance. However, a good network performance can be obtained with circuit switching if the setup latency can be relaxed or avoided. In this thesis we introduce PROSA, a circuit-switched enabled NoC architecture which allows the coherence protocol to steer circuit connections for future predictable MC transactions between the MC and L2 cache, and speculative connections between, L2 cache banks and L1 caches. Connection establishment is performed as single-packet based and established for the required time period for L2 to L1 an MC to L2. On a miss predict the circuit is silently removed. PROSA builds a separate infrastructure to manage circuits, thus the main NoC network is not affected by the additional control traffic. PROSA relies on SMART technologies in order to improve more the network latency. SMART is a multihop technology, it allows the flit to perform multiples hops in a single cycle. Using this technology and getting benefit from the protocol coherence information to setup the circuits before they are needed, PROSA improves on average the network latency by 34%, and this improvement is transformed into a high execution runtime reduction, on average PROSA reduces execution runtime by 33%.

Also, further enhancements of PROSA are presented: 1) the coherence protocol allows to send all messages via circuits, 2) modifications at arbiters to set priorities, 3) a new threshold mechanism to limit the distance of circuits, and 4) removal of NACK messages with implicit teardown circuits logic implemented. These enhancements allow to PROSA to reduce network latency compared with baseline PROSA by 35%, but this improvement is not reflected on the execution runtime. Moreover, with these enhancements the required area and power consumption are reduced by 2% and 5% compared with the standard PROSA.

The last proposal is Token-based TDM, a novel approach to sustain domain isolation in high-assurance systems. Special emphasis has been devoted to providing protection against subtle security threats such as timing channels. This requires the enforcement of strong isolation in its strictest sense, that is, injection of packets in one domain cannot affect the timing of packet delivery of other domains. Our approach leverages the unique understanding of the problem enabled by the channel dependency graph abstraction of an on-chip interconnection network. Overall, Token-based TDM is proven to match the performance best cases of PhaseNoC (the most competitive approach presented so far in literature), while generalizing the performance efficiency to the remaining configurations. Performance speedups range from 10% for intra-domain traffic up to 30% for MC traffic. At the same time, our approach enables a flexible NoC architecture, that potentially goes beyond the design-time tunability of PhaseNoC. In fact, our philosophy is not to update the switch architecture to the execution scenario, but rather to change the scheduling commands sent to the switches via a token-based notification mechanism. This provides an unprecedented level of flexibility at an overly small implementation complexity.

9.2 Publications

Conferences:

- M. Gorgues, D. Xiang, J. Flich, Z. Yu and J. Duato, ”Achieving balanced buffer utilization with a proper co-design of flow control and routing algorithm,” 2014 Eighth IEEE/ACM International Symposium on Networks-on-Chip (NoCS), Ferrara, 2014, pp. 25-32. doi: 10.1109/NOCS.2014.7008758
- M. Gorgues and J. Flich, ”End-Point Congestion Filter for Adaptive Routing with Congestion-Insensitive Performance,” in IEEE Computer Architecture Letters, vol. 15, no. 1, pp. 9-12, Jan.-June 1 2016. doi: 10.1109/LCA.2015.2429130
- M. Gorgues and J. Flich, ”PROSA: protocol-driven NoC architecture,” 2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS), Nara, 2016, pp. 1-8. doi: 10.1109/NOCS.2016.7579320
- M. Gorgues and J. Flich, ”PROSA: Protocol-Driven Network on Chip Architecture,” in IEEE Transactions on Parallel and Distributed Systems, vol. PP, no. 99, pp. 1-1. doi: 10.1109/TPDS.2017.2784422
- M. Gorgues , D. Bertozzi and J. Flich, ”A Low-Latency Reconfigurable Time-Division Multiplexed Network-on-chip for High-Assurance Systems,” submitted in 2018 12th IEEE/ACM International Symposium on Networks-on-Chip (NOCS) pending of acceptance.

In addition, other related papers have been published in national conferences:

- M. Gorgues, D. Xiang, J. Flich, Z. Yu and J. Duato, "Codiseño de un Mecanismo de Control Flujo y un Algoritmo de Encaminamiento," XXV Jornadas de Paralelismo, Valladolid, 2014, pp. 455-462. ISBN: 978-84-697-0329-8
- M. Gorgues and J. Flich, "End-Point Congestion Filter for highly-Loaded Network with Adaptive routing," XXVI Jornadas de Paralelismo, XXVI Jornadas de Paralelismo. Jornadas SARTECO 2015, Cordoba, 2015, pp. 213-217. ISBN: 978-84-16017-52-2
- M. G. Alonso and J. F. Cardo, "Arquitectura NoC Híbrida con Conmutación de Paquete-Circuito Dirigida por el Protocolo de Coherencia," XXVII Jornadas de Paralelismo. Jornadas SARTECO 2016, Salamanca, 2016, pp. 421-429. ISBN: 978-84-9012-626-4

References

[1] Wikipedia. Intel 80286. Available at https://es.wikipedia.org/wiki/Intel_80286.

[2] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of 51st Design Automation Conference (DAC)*, pages 684–689, 2001. doi: 10.1109/DAC.2001.156225.

[3] J. Flich and D. Bertozzi. *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC, 2010. ISBN 1439837104, 9781439837108.

[4] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z. T. Deniz, D. Wendel, and M. Ziegler. 5.1 power8tm: A 12-core server-class processor in 22nm soi with 7.6tb/s off-chip bandwidth. In *Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 96–97, Feb 2014. doi: 10.1109/ISSCC.2014.6757353.

[5] Tilera Corp. Tilera tile multicore processors. Available at http://www.mellanox.com/page/products_dyn?product_family=238&mtag=tile_gx72.

[6] J. Duato, I. Johnson, J. Flich, F. Naven, P. García, and T. N. Frinós. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In *Proceedings of 11th International Conference on High-Performance Computer Architecture (HPCA)*, pages 108–119, 2005. doi: 10.1109/HPCA.2005.1. URL https://doi.org/10.1109/HPCA.2005.1.

[7] T. Krishna, C. H. O. Chen, W. C. Kwon, and L. S. Peh. Breaking the on-chip latency barrier using SMART. In *Proceedings of 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 378–389, 2013. doi: 10.1109/HPCA.2013.6522334. URL https://doi.org/10.1109/HPCA.2013.6522334.

[8] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *In IBM Journal of Research and Development*, 49(4.5):589–604, July 2005. ISSN 0018-8646. doi: 10.1147/rd.494.0589.

[9] A. Kumar, L. S. Peh, P. Kundu, and N. K. Jha. Express virtual channels: towards the ideal interconnection fabric. In *Proceedings of 34th International Symposium on Computer Architecture (ISCA)*, pages 150–161, 2007. doi: 10.1145/1250662.1250681. URL `http://doi.acm.org/10.1145/1250662.1250681`.

[10] N. D. E. Jerger, L. S. Peh, and M. H. Lipasti. Circuit-switched coherence. In *Proceedings of 2nd ACM/IEEE International Symposium on Networks-on-Chip (NOCS)*, pages 193–202, April 2008. doi: 10.1109/NOCS.2008.4492738.

[11] J. Yin, P. Zhou, S. S. Sapatnekar, and A. Zhai. Energy-efficient time-division multiplexed hybrid-switched noc for heterogeneous multicore systems. In *Proceedings of IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 293–303, 2014. doi: 10.1109/IPDPS.2014.40. URL `https://doi.org/10.1109/IPDPS.2014.40`.

[12] A. Mazloumi and M. Modarressi. A hybrid packet/circuit-switched router to accelerate memory access in noc-based chip multiprocessors. In *Proceedings of 18th Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 908–911, 2015. URL `http://dl.acm.org/citation.cfm?id=2757023`.

[13] G. Chen, M. A. Anders, H. Kaul, S. K. Satpathy, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, V. De, and S. Borkar. A 340 mv-to-0.9 v 20.2 tb/s source-synchronous hybrid packet/circuit-switched 16x16 network-on-chip in 22 nm tri-gate cmos. *In IEEE Journal of Solid-State Circuits*, 50(1):59–67, Jan 2015. ISSN 0018-9200. doi: 10.1109/JSSC.2014.2369508.

[14] A. Abousamra, R. G. Melhem, and A. K. Jones. Déjà vu switching for multiplane nocs. In *Proceedings of 6th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 11–18, 2012. doi: 10.1109/NOCS.2012.9. URL `https://doi.org/10.1109/NOCS.2012.9`.

[15] A. Abousamra, A. K. Jones, and R. G. Melhem. Proactive circuit allocation in multiplane nocs. In *Proceedings of 50th Annual Design Automation Conference (DAC)*, pages 35:1–35:10, 2013. doi: 10.1145/2463209.2488778. URL `http://doi.acm.org/10.1145/2463209.2488778`.

[16] A. Van Laer, C. Ellawala, M. R. Madarbux, P. M. Watts, and T. M. Jones. Coherence based message prediction for optically interconnected chip multiprocessors. In *Proceedings of 18th Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 613–616, 2015. URL `http://dl.acm.org/citation.cfm?id=2755892`.

[17] A. Shacham, K. Bergman, and L. P. Carloni. Photonic networks-on-chip for future generations of chip multiprocessors. *In IEEE Transactions on Computers*, 57(9): 1246–1260, Sept 2008. ISSN 0018-9340. doi: 10.1109/TC.2008.78.

[18] L. S. Peh and W. J. Dally. Flit-reservation flow control. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 73–84, 2000. doi: 10.1109/HPCA.2000.824340. URL `https://doi.org/10.1109/HPCA.2000.824340`.

[19] S. Liu, A. Jantsch, and Z. Lu. Parallel probing: Dynamic and constant time setup procedure in circuit switching noc. In *Proceedings of the 15th Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1289–1294, 2012. doi: 10.1109/DATE.2012.6176691. URL `https://doi.org/10.1109/DATE.2012.6176691`.

[20] Y. Xue and P. Bogdan. Improving noc performance under spatio-temporal variability by runtime reconfiguration: a general mathematical framework. In *Proceedings of the 10th IEEE/ACM International Symposium on Networks-on-Chip, (NOCS)*, pages 1–8, 2016. doi: 10.1109/NOCS.2016.7579322. URL `https://doi.org/10.1109/NOCS.2016.7579322`.

[21] S. J. Hollis, C. Jackson, P. Bogdan, and R. Marculescu. Exploiting emergence in on-chip interconnects. *In IEEE Transactions on Computers*, 63(3):570–582, 2014. doi: 10.1109/TC.2012.273. URL `https://doi.org/10.1109/TC.2012.273`.

[22] Y. Xue and P. Bogdan. User cooperation network coding approach for noc performance improvement. In *Proceedings of the 9th International Symposium on Networks-on-Chip, (NOCS)*, pages 17:1–17:8, 2015. doi: 10.1145/2786572.2786575. URL `http://doi.acm.org/10.1145/2786572.2786575`.

[23] Z. Qian, P. Bogdan, G. Wei, C. Tsui, and R. Marculescu. A traffic-aware adaptive routing algorithm on a highly reconfigurable network-on-chip architecture. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis, (CODES+ISSS)*, pages 161–170, 2012. doi: 10.1145/2380445.2380475. URL `http://doi.acm.org/10.1145/2380445.2380475`.

[24] M. Tang and X. Lin. A new adaptive flow control for mesh-based network-on-chip (noc). In *Proceedings of the 8th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, pages 255–260, June 2009. doi: 10.1109/ICIS.2009.113.

[25] I. Nousias and T. Arslan. Wormhole routing with virtual channels using adaptive rate control for network-on-chip (noc). In *Proceedings of the 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 420–423, June 2006. doi: 10.1109/AHS.2006.79.

[26] P. Avasare, V. Nollet, J. Mignolet, D. Verkest, and H. Corporaal. Centralized end-to-end flow control in a best-effort network-on-chip. In *Proceedings of the 5th ACMInternational Conference On Embedded Software (EMSOFT)*, pages 17–20, 2005. doi: 10.1145/1086228.1086232. URL `http://doi.acm.org/10.1145/1086228.1086232`.

[27] S. Ma, N. D. Enright, and Z. Wang. Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 467–478, 2012. doi: 10.1109/HPCA.2012.6169049. URL `https://doi.org/10.1109/HPCA.2012.6169049`.

[28] S. Ma, Z. Wang, Z. Liu, and N. D. Enright. Leaving one slot empty: Flit bubble flow control for torus cache-coherent nocs. *In IEEE Transaction on Computers*, 64(3):763–777, 2015. doi: 10.1109/TC.2013.2295523. URL `https://doi.org/10.1109/TC.2013.2295523`.

[29] L. Chen and T. M. Pinkston. Worm-bubble flow control. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 366–377, 2013. doi: 10.1109/HPCA.2013.6522333. URL `https://doi.org/10.1109/HPCA.2013.6522333`.

[30] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *In IEEE Transaction on Computers*, 36(5):547–553, 1987. doi: 10.1109/TC.1987.1676939. URL `https://doi.org/10.1109/TC.1987.1676939`.

[31] C. J. Glass and L. M. Ni. The turn model for adaptive routing. volume 41, pages 874–902, 1994. doi: 10.1145/185675.185682. URL `http://doi.acm.org/10.1145/185675.185682`.

[32] G. M. Chiu. The odd-even turn model for adaptive routing. *In IEEE Transactions on Parallel and Distributed Systems*, 11(7):729–738, 2000. doi: 10.1109/71.877831. URL `https://doi.org/10.1109/71.877831`.

[33] J. Wu. A fault-tolerant and deadlock-free routing protocol in 2d meshes based on odd-even turn model. *In IEEE Transactions on Computers*, 52(9):1154–1169, 2003. doi: 10.1109/TC.2003.1228511. URL `https://doi.org/10.1109/TC.2003.1228511`.

[34] D. Xiang and W. Luo. An efficient adaptive deadlock-free routing algorithm for torus networks. *In IEEE Transactions on Parallel and Distributed Systems*, 23(5):800–808, 2012. doi: 10.1109/TPDS.2011.145. URL `https://doi.org/10.1109/TPDS.2011.145`.

[35] A. Singh, W. J. Dally, A. K. Gupta, and B. Towles. GOAL: A load-balanced adaptive routing algorithm for torus networks. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 194–205, 2003. doi: 10.1109/ISCA.2003.1207000. URL `https://doi.org/10.1109/ISCA.2003.1207000`.

[36] A. Singh, W. J. Dally, A. K. Gupta, and B. Towles. Adaptive channel queue routing on k-ary n-cubes. In *Proceedings of the 16th Annual ACM Symposium on*

Parallelism in Algorithms and Architectures (SPAA), pages 11–19, 2004. doi: 10.1145/1007912.1007915. URL `http://doi.acm.org/10.1145/1007912.1007915`.

[37] L. Gravano, G. D. Pifarré, P. E. Berman, and J. L. C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. *In IEEE Transactions on Parallel and Distributed Systems*, 5(12):1233–1251, 1994. doi: 10.1109/71.334898. URL `https://doi.org/10.1109/71.334898`.

[38] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. *In IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067, 1995. doi: 10.1109/71.473515. URL `https://doi.org/10.1109/71.473515`.

[39] J. Duato and T. M. Pinkston. A general theory for deadlock-free adaptive routing using a mixed set of resources. *In IEEE Transactions on Parallel and Distributed Systems*, 12(12):1219–1235, 2001. doi: 10.1109/71.970556. URL `https://doi.org/10.1109/71.970556`.

[40] V. Puente, R. Beivide, J. A. Gregorio, J. M. Prellezo, J. Duato, and C. Izu. Adaptive bubble router: a design to improve performance in torus networks. In *Proceedings of the 1999 International Conference on Parallel Processing (ICPP)*, pages 58–67, 1999. doi: 10.1109/ICPP.1999.797388.

[41] J. Hu and R. Marculescu. Dyad: smart routing for networks-on-chip. In *Proceedings of the 41th Design Automation Conference, (DAC)*, pages 260–263, 2004. doi: 10.1145/996566.996638. URL `http://doi.acm.org/10.1145/996566.996638`.

[42] M. Ebrahimi, X. Chang, M. Daneshtalab, J. Plosila, P. Liljeberg, and H. Tenhunen. Dyxyz: Fully adaptive routing algorithm for 3d nocs. In *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, (PDP)*, pages 499–503, 2013. doi: 10.1109/PDP.2013.80. URL `https://doi.org/10.1109/PDP.2013.80`.

[43] A. Mejia, J. Flich, J. Duato, S. A. Reinemo, and T. Skeie. Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In *Proceedings of 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, April 2006. doi: 10.1109/IPDPS.2006.1639341.

[44] P. Yew, N. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *In IEEE Transactions on Computers*, 36(4):388–395, 1987. doi: 10.1109/TC.1987.1676921. URL `https://doi.org/10.1109/TC.1987.1676921`.

[45] J. L. Ferrer, E. Baydal, A. Robles, P. López, and J. Duato. Progressive congestion management based on packet marking and validation techniques. *In IEEE Transactions on Computers*, 61(9):1296–1310, 2012. doi: 10.1109/TC.2011.146. URL `https://doi.org/10.1109/TC.2011.146`.

[46] Y. Xu, B. Zhao, Y. Zhang, and J. Yang. Simple virtual channel allocation for high-throughput and high-frequency on-chip routers. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–11, Jan 2010. doi: 10.1109/HPCA.2010.5416640.

[47] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally. Network congestion avoidance through speculative reservation. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 443–454, 2012. doi: 10.1109/HPCA.2012.6169047. URL `https://doi.org/10.1109/HPCA.2012.6169047`.

[48] N. McKeown, A. Mekkittikul, V. Anantharam, and J. C. Walrand. Achieving 100% throughput in an input-queued switch. *In IEEE Transactions on Communications*, 47(8):1260–1267, 1999. doi: 10.1109/26.780463. URL `https://doi.org/10.1109/26.780463`.

[49] P. Gratz, B. Grot, and S. W. Keckler. Regional congestion awareness for load balance in networks-on-chip. In *Proceedings of the 14th International Conference on High-Performance Computer Architecture (HPCA)*, pages 203–214, 2008. doi: 10.1109/HPCA.2008.4658640. URL `https://doi.org/10.1109/HPCA.2008.4658640`.

[50] M. Ramakrishna, V. K. Kodati, P. V. Gratz, and A. Sprintson. Gca:global congestion awareness for load balance in networks-on-chip. *In IEEE Transactions on Parallel and Distributed Systems*, 27(7):2022–2035, July 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2477840.

[51] X. Chang, M. Ebrahimi, M. Daneshtalab, T. Westerlund, and J. Plosila. Pars; an efficient congestion-aware routing method for networks-on-chip. In *Proceedings of the 16th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 166–171, May 2012. doi: 10.1109/CADS.2012.6316439.

[52] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 414–423, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press. ISBN 0-8186-0719-X. URL `http://dl.acm.org/citation.cfm?id=17407.17404`.

[53] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011. doi: 10.2200/S00346ED1V01Y201104CAC016. URL `https://doi.org/10.2200/S00346ED1V01Y201104CAC016`.

[54] Y. Wang and G. E. Suh. Efficient timing channel protection for on-chip networks. In *Proceedings of the IEEE/ACM 6th International Symposium on Networks-on-Chip (NOCS)*, pages 142–151, May 2012. doi: 10.1109/NOCS.2012.24.

[55] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *In IEEE Design Test of Computers*, 22(5): 414–421, Sept 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.99.

[56] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A flit-synchronous network on chip with composable and predictable services. In *Proceedings of the 12th Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 250–255, April 2009. doi: 10.1109/DATE.2009.5090666.

[57] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient gals implementation. *In IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24 (2):479–492, Feb 2016. ISSN 1063-8210. doi: 10.1109/TVLSI.2015.2405614.

[58] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki. A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pages 152–160, May 2012. doi: 10.1109/NOCS.2012.25.

[59] T. Marescaux and H. Corporaal. Introducing the supergt network-on-chip; supergt qos: more than just gt. In *2007 44th ACM/IEEE Design Automation Conference*, pages 116–121, June 2007.

[60] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood. Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 583–594, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485972. URL `http://doi.acm.org/10.1145/2485922.2485972`.

[61] A. Psarras, I. Seitanidis, C. Nicopoulos, and G. Dimitrakopoulos. Phasenoc: Tdm scheduling at the virtual-channel level for efficient network traffic isolation. In *Preceedings of the 18th Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1090–1095, March 2015. doi: 10.7873/DATE.2015.0418.

[62] A. Kumar, P. Kundu, A. P. Singh, L. Peh, and N. K. Jha. A 4.6tbits/s 3.6ghz single-cycle noc router with a novel switch allocator in 65nm CMOS. In *Proceedings of th 25th International Conference on Computer Design, (ICCD)*, pages 63–70, 2007. doi: 10.1109/ICCD.2007.4601881. URL `https://doi.org/10.1109/ICCD.2007.4601881`.

[63] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *In IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, 1993. doi: 10.1109/71.250114. URL `https://doi.org/10.1109/71.250114`.

[64] W. Dally and B. Towles. *Principles and Practices of InterconnectionNetwork*. Morgan Kaufmann, San Mateo, 2nd edition, 2004.

[65] J. V. Escamilla, J. Flich, and P. J. García. ICARO: congestion isolation in networks-on-chip. In *Proceedings of the 8th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 159–166, 2014. doi: 10.1109/NOCS.2014.7008775. URL `https://doi.org/10.1109/NOCS.2014.7008775`.

[66] DISCA UPV. urlhttp://wiki.mnit.ac.in/mediawiki/index.php/GMemNoCsim.

[67] S. Qi, J. Li, T. Zhao, X. Jia, and M. Zhang. The design and implementation of two-cycle noc router. In *Proceedings of the 10th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 233–235, Nov 2010. doi: 10.1109/ICSICT.2010.5667782.

[68] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995. doi: 10.1145/223982.223990. URL `http://doi.acm.org/10.1145/223982.223990`.

[69] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 72–81, 2008. doi: 10.1145/1454115.1454128. URL `http://doi.acm.org/10.1145/1454115.1454128`.

[70] ”nangate freepdk45 generic open cell library ver 1.0.”. `http://www.si2.org/openeda.si2.org/projects/nangatelib/`.

[71] A. B. Kahng, B. Lin, and S. Nath. ORION3.0: A comprehensive noc router estimation tool. *In Embedded Systems Letters*, 7(2):41–45, 2015. doi: 10.1109/LES.2015.2402197. URL `https://doi.org/10.1109/LES.2015.2402197`.

[72] G. Dhiman, k. Pusukuri, and T. Rosing. Analysis of dynamic voltage scaling for system level energy management. In *Proceedings of the 2008 Conference on Power Aware Computing and Systemsv (HotPower)*, pages 9–9, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855610.1855619`.

[73] A. Mejia, J. Flich, and J. Duato. On the potentials of segment-based routing for nocs. In *Proceedings of th e 37th International Conference on Parallel Processing (IPDPS)*, pages 594–603, Sept 2008. doi: 10.1109/ICPP.2008.56.

[74] D. Bortolotti, A. Marongiu, and L. Benini. Virtualsoc: A research tool for modern mpsocs. *ACM Transactions on Embedded Computing Systems*, 16(1):

3:1–3:27, October 2016. ISSN 1539-9087. doi: 10.1145/2930665. URL `http://doi.acm.org/10.1145/2930665`.

[75] M. Gorgues, D. Xiang, J. Flich, Z. Yu, and J. Duato. Achieving balanced buffer utilization with a proper co-design of flow control and routing algorithm. In *Proceedings of the 8th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 25–32, Sept 2014. doi: 10.1109/NOCS.2014.7008758.

[76] M. Gorgues and J. Flich. End-point congestion filter for adaptive routing with congestion-insensitive performance. *In IEEE Computer Architecture Letters*, 15(1): 9–12, Jan 2016. ISSN 1556-6056. doi: 10.1109/LCA.2015.2429130.

[77] M. G. Alonso and J. F. Cardo. Prosa: protocol-driven noc architecture. In *Proceedings of the 10th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, Aug 2016. doi: 10.1109/NOCS.2016.7579320.

[78] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *IEEE Computer*, 35(1):70–78, 2002. doi: 10.1109/2.976921. URL `https://doi.org/10.1109/2.976921`.

[79] M. Gorgues Alonso and J. Flich. Prosa: Protocol-driven network on chip architecture. *In IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2017. ISSN 1045-9219. doi: 10.1109/TPDS.2017.2784422.

[80] T. Pimpalkhute and S. Pasricha. Noc scheduling for improved application-aware and memory-aware transfers in multi-core systems. In *Proceedings of th 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 234–239, 2014. doi: 10.1109/VLSID.2014.47. URL `https://doi.org/10.1109/VLSID.2014.47`.

[81] Tilera Corp. Tilera TILE Multicore Processors.

[82] Kalray. Kalray, MPPA-256 Bostan.

[83] R. Hilbrich and J. R. Kampenhout. Partitioning and task transfer on noc-based many-core processors in the avionics domain. *In Softwaretechnik-Trends*, 31(3), 2011. URL `http://pi.informatik.uni-siegen.de/stt/31_3/01_Fachgruppenberichte/ada/1-Hilbrich-many-core.pdf`.

[84] O. Stan, Y. Elovici, A. Shabtai, G. Shugol, R. Tikochinski, and S. Kur. Protecting military avionics platforms from attacks on mil-std-1553 communication bus. *In CoRR*, abs/1707.05032, 2017.

[85] T. D. Nguyen. Towards mil-std-1553b covert channel analysis, 2015. URL `https://calhoun.nps.edu/handle/10945/44707`.

[86] P. Kleberger, T. Olovsson, and E. Jonsson. Security aspects of the in-vehicle network in the connected car. In *Proceedings of the 4th IEEE Intelligent Vehicles Symposium (IVS)*, pages 528–533, 2011. doi: 10.1109/IVS.2011.5940525. URL `https://doi.org/10.1109/IVS.2011.5940525`.

[87] M. Wolf, A. Weimerskirch, and C. Paar. Security in automotive bus systems. In *Proceedings of the Workshop on Embedded Security in Cars(ESCAR)*, 04 2004.

[88] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks - practical examples and selected short-term countermeasures. *In Reliability Engineering and System Safety*, 96(1):11–25, 2011. doi: 10.1016/j.ress.2010.06.026. URL `https://doi.org/10.1016/j.ress.2010.06.026`.

[89] W. Otte, A. Dubey, S. Pradhan, P. Patil, A. S. Gokhale, G. Karsai, and J. Willemsen. F6COM: A component model for resource-constrained and dynamic space-based computing environments. In *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC)*, pages 1–8, 2013. doi: 10.1109/ISORC.2013.6913199. URL `https://doi.org/10.1109/ISORC.2013.6913199`.

[90] E. Ong and J. Losinski O. Brown. System F6: Progress to Date. In *Small Satellite Constellations: Strength in Numbers,*, page 7, 2012.

[91] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato. Cost-efficient on-chip routing implementations for cmp and mpsoc systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):534–547, April 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2011.2119150.